# Leanexe's numerical library: proved accuracy for compiled floating-point code

exaspec@morphism.com

27 November 2026

**Abstract**

Leanexe compiles programs written in a first-order subset of Lean and proves that the compiled code computes the Lean definition. Lean's kernel checks the proof for each program. The compiled code runs on AArch64 and x86-64 processors, on an Arm Cortex-M7 microcontroller, on GPUs, and in WebAssembly engines. This report describes leanexe's library of binary64 floating-point functions and the theorems that bound their accuracy.

For each elementary and special function, Lean's kernel checks a theorem that bounds the compiled code's result against a real function. For the elementary functions and the error and gamma functions, the result lies within a stated number of steps of the correctly rounded value. A step separates adjacent binary64 values, and the bound is one step for most of these functions and two or three steps for the others. For the Bessel functions, the bounds are absolute errors, except for $Y_0$ and $Y_1$ below $2^{-1022}$. Mathlib defines the reference functions, except the error function and the Bessel functions, which leanexe's library of shared lemmas defines from Mathlib's integrals and power series.

The theorem describes the WebAssembly module that the verified compiler produces. Leanexe translates that module to C code and to bare-metal AArch64 Images, and theorems state that the translations compute the same Lean definitions. By composition with those theorems, the bounds hold for the results of the C code and the Images.

The fused multiply-add and the decimal parser return correctly rounded results, and the decimal printer returns the shortest decimal that reads back to the same value. Sums, means, dot products, matrix products, and discrete Fourier transforms have proved error bounds against the exact real results. Intervals with outward-rounded endpoints give enclosures of exact real quantities, and one command derives an interval program and its enclosure theorem from a real definition. The theorems hold under WebAssembly's deterministic floating-point profile, which browsers do not implement, and they bound neither the width of an interval nor the running time. The report compares the library with Flocq, Gappa, FPTaylor, CRlibm, CORE-MATH, Arb, CompCert, FloatLib, and floating-point proofs in HOL Light, Isabelle, and CakeML.

## 1 Introduction

A floating-point library computes approximations of real functions, and a program that uses it needs a bound on the distance between each result and the exact value. The GNU C Library, for example, aims for results within a few units in the last place of the exact value, states that some of its functions miss that aim, and its test suite flags only errors above 9 units [17, §19.7]. A test measures the error at the tested arguments and bounds nothing at the others. Machine-checked proofs in Coq and HOL Light bound the errors of floating-point algorithms in formal models of IEEE 754 arithmetic (Section 8). The code that runs is a compiled program, and a theorem about the algorithm states nothing about that program unless a further proof relates the two.

Leanexe compiles programs written in a first-order subset of Lean and proves that the compiled code computes the Lean definition [28]. The compiled code runs on AArch64 and x86-64 processors, on an Arm Cortex-M7 microcontroller, on GPUs, and in WebAssembly engines. The verified compiler produces a WebAssembly module, which serves as the intermediate representation: the C and machine-code translations start from that module, and WebAssembly engines run the module itself. Companion reports describe the verified compiler [14] and the targets [15], and this report does not repeat them.

Leanexe's numerical library writes its functions in the dialect, the first-order subset of Lean that leanexe compiles. The verified compiler therefore compiles each function with a theorem that the compiled code computes the function's Lean definition. For each elementary and special function, a Lean proof bounds the definition's result against a real function. A master theorem combines that bound with the compiler's theorem for the function's export, so the bound becomes a statement about the compiled module.

Mathlib, Lean's library of formalized mathematics [41], defines most of the reference functions, and leanexe's library of shared lemmas defines the error function and the Bessel functions. Most bounds count steps between the result and the correctly rounded value, on an ordering of the binary64 values that Section 2 defines. The Bessel functions' bounds are absolute errors (Section 3).

The library adds the following to the verified compiler and its targets.

1. Theorems that bound each elementary and special function's export against a real function, in steps from the correctly rounded value or, for the Bessel functions, as an absolute error (bounds and domains in Table 1, Section 3).
2. Proofs of those bounds that follow each algorithm's stages, with constants and polynomial bounds that Lean's kernel checks in exact rational arithmetic (Section 4).
3. Error bounds for sums, means, sums of squares, dot products, matrix products, and discrete Fourier transforms (Section 3.3).
4. A software fused multiply-add and decimal conversions, with theorems that state their exact results (Section 5).
5. Interval arithmetic with outward rounding, and a command that writes an interval program and its enclosure theorem from a real definition (Section 6).

This report describes leanexe at commit `4556fa84` of 26 November 2026, with Lean 4.34.1 [25], Mathlib at `85e3a25`, and the WebAssembly interpreter Talos at `87e3aa5e` [5]. Every link to a leanexe file points to that commit. Where leanexe's documents at that commit contradict its sources, this report states what the sources show.

Leanexe's index calls the error bounds of binary64 sums, statistics, and linear algebra open (`index.md:162`, `index.md:173`, `index.md:238`). The guides of the reductions, the mean, and linear algebra, which are the README files of their directories, also call those bounds open. Master theorems state those bounds (Section 3.3), and the pull request "State the proved error bounds in the numerics READMEs" corrects the documents [4].

# 2 Binary64 words, rounding to nearest, and steps

## 2.1 One bit-level semantics for Lean's floats and WebAssembly

ProofKit, leanexe's library of shared lemmas, proves that Lean's float operations equal Talos's definitions of the corresponding WebAssembly instructions on all words, NaN included, as `toBits_add` does for addition (`LeanExe/ProofKit/F64Bits.lean:46`). Lean 4.34 defines its types `Float` and `Float32` through `Float.Model` and `Float32.Model`, models of IEEE binary64 and binary32 with

rounding to nearest, ties to even (docs/design.md:189). Talos, the WebAssembly interpreter written in Lean that gives the module's theorem its semantics [5, 14], defines each `f64` instruction as a function on 64-bit words. A binary64 word has a sign bit, an 11-bit exponent field, and a 52-bit fraction field. Talos's addition, multiplication, division, and square root compute the exact result from the operands' values as integers in units of $2^{-1074}$, the smallest subnormal value, and round it once (interpreter/Interpreter/Wasm/IEEE64.lean:105).

Lean's model of `Float` has one NaN, the positive quiet NaN `0x7FF8000000000000`, and `Float.ofBits` maps every NaN word to it (docs/design.md:189). A theorem about a function with a `Float` parameter therefore covers only the canonical NaN among the NaN words, and a theorem about a function of a `UInt64` word covers every word (docs/design.md:252). WebAssembly lets an engine choose the sign of a NaN that an instruction generates, and its deterministic profile requires every such NaN to be canonical and positive [6]. Talos implements that profile, leanexe's host for Wasmtime enables the canonicalization of NaN results, and browsers do not implement the profile (docs/design.md:198). The compiler removes the other differences: negation compiles to `-0.0 - x`, `Float.ofBits` to a reinterpretation followed by an addition of $-0$, and `min` and `max` to a comparison and a selection (docs/design.md:200).

## 2.2 The correctly rounded reference and the count of steps

The accuracy theorems compare each result with a reference word that ProofKit defines from a real number. `value w` is the real number that a finite word `w` denotes (codelib/CodeLib/IEEE64/Roundoff.lean:20). `roundNearestEven y` is the binary64 word nearest to the real number `y`, the even one at a tie (LeanExe/ProofKit/F64SignedRounding.lean:18). When $|y|$ is at least the overflow threshold $2^{1024} - 2^{970}$, `roundNearestEven y` is the infinity of `y`'s sign (LeanExe/ProofKit/F64PositiveRounding.lean:10). The threshold is the midpoint between the largest finite value and $2^{1024}$, and the exponential's guide cites clause 4.3.1 of IEEE 754-2019 [22] for that rule (Library/Numerics/Float64/Exp/README.md:7).

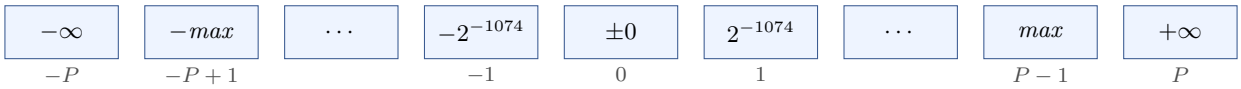
The step count uses `position`, which places every word except a NaN on an ordered line (LeanExe/ProofKit/F64SignedRounding.lean:23). The position of a word with a clear sign bit is the word as an unsigned integer, and the position of a word with the sign bit set is the negation of its magnitude's position. Both zeros therefore lie at position 0, and each infinity lies next to the largest finite value of its sign. `WithinSteps n y w` states that `w` is not a NaN and that the positions of `w` and `roundNearestEven y` differ by at most `n` (LeanExe/ProofKit/F64SignedRounding.lean:28). Figure 1 shows both definitions.

A step has no fixed size, since the spacing of binary64 values doubles at each power of two and the steps count words on both sides of that boundary alike. A bound of zero steps would state correct rounding except for the sign of a zero result, since both zeros share position 0 (Library/Numerics/Arithmetic/Fma/README.md:7). A bound of one step admits the correctly rounded word and the word next to it on each side.

## 2.3 Roundoff of the basic operations over the finite range

ProofKit bounds the roundoff of each basic operation for exact results up to $2^{1022}$ or $2^{1023}$ in magnitude. Addition and subtraction are within $2^{-53}$ times an exact result below $2^{1023}$ in magnitude, and multiplication and division add an absolute term for underflow below $2^{1022}$ (LeanExe/ProofKit/F64AddBounds.lean:23, LeanExe/ProofKit/F64MulBounds.lean:97, LeanExe/ProofKit/F64DivBounds.lean:51). The square root is within $2^{-53}$ times the exact root (LeanExe/ProofKit/F64SqrtBounds.lean:11). The predicate `FApprox` carries such bounds through a computation, with

(a) Positions of words on the ordered line, with $P$ the position of $+\infty$, the word `0x7FF0000000000000`




(b) Words near a power of two $2^q$, the reference $r$ of a real $y$, and the words within one and two steps of $r$

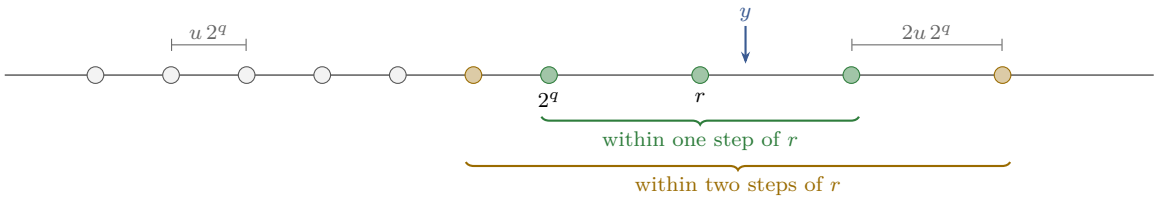



Figure 1: The ordered line of binary64 words on which `WithinSteps` counts steps (`LeanExe/ProofKit/F64SignedRounding.lean:23`). In part (a), blue cells are words with their positions below them. Both zeros share position 0, the smallest subnormal values lie next to them, and the infinities lie next to the largest finite values $\pm max$. In part (b), dots are adjacent words, with $u = 2^{-53}$: the spacing doubles at $2^q$, and a step counts one word whatever the spacing. The blue arrow marks the real $y$, $r$ is `roundNearestEven y`, green dots are the words within one step of $r$, and green and orange dots together are the words within two steps.

one lemma for each operation (`LeanExe/ProofKit/F64Approx.lean:240`). Talos's roundoff lemmas assume operands of magnitude at most 1 (`codelib/CodeLib/IEEE64/Roundoff.lean:624`), and the library's proofs use ProofKit's lemmas in their place.

# 3 Accuracy theorems for the compiled module

For each elementary and special function, Lean's kernel checks a theorem that bounds the compiled code's result against a real function, in steps from the correctly rounded value or, for the Bessel functions, as an absolute error. The theorem describes the WebAssembly module that the verified compiler produces. The bound also holds for the C code and the AArch64 Images that leanexe translates from the module, because their theorems state that they compute the same Lean definitions. Figure 2 shows that composition for the exponential (Section 3.4). Table 1 lists the bounds, the reference functions, and the domains.

## 3.1 The form of a master theorem

`Verified.Master m tables name B D R` states that the module `m` exports a function under `name`, and it describes every call of that function on an argument `x` with `D x` (`Verified/Master.lean:32`). Called from a ready store with room for `B x` bytes, the function returns a value `y` with `R x y` and does not trap. A store is ready when the allocator's invariant holds, the module's tables lie at their addresses, and the memory's cap is at most 65,535 pages (`Library/Numerics/Float64/Exp/README.md:71`). The function that the module computes appears only under an existential quantifier, so a reader of a master theorem needs the definitions in `R`, `D`, and `B` and none of the program (`Verified/Master.lean:12–16`). A registry file imports a master theorem for every function of the library (`Library/Masters.lean`).

The master theorem of the exponential, in Listing 1, states `WithinSteps 1` against Mathlib's `Real.exp` of the argument's value for every finite argument. `Master.of_trapFree` proves such a

```
theorem exp_master :
    Verified.Master compiled.module compiled.tables "exp" compiled.exp.bound
      (fun x : Float => Finite x.toBits)
      (fun x (y : Float) =>
        F64SignedRounding.WithinSteps 1 (Real.exp (value x.toBits)) y.toBits)
```

Listing 1: The master theorem of the exponential (Library/Numerics/Float64/Exp/Master.lean:11), without its proof. `value` is the real value of a word, and `Real.exp` is Mathlib's exponential.

statement from two theorems (Verified/Master.lean:54). The first is the compiler's theorem `compiled.exp.trapFree`, which states that the module's export returns the bits of `Exp.exp x` without a trap [14]. The second is `exp_one_step`, which bounds the result of the Lean definition `Exp.exp` against `Real.exp` for every finite argument (Library/Numerics/Float64/Exp/ExpTheorem.lean:212). Separate theorems give the exponential's exact results for zeros, infinities, and NaN, and the master theorems of most other functions include such exact results in their relations (Library/Numerics/Float64/Exp/README.md:11–18).

### 3.2 Accuracy of the elementary and special functions

Mathlib defines every reference of Table 1 except the error function and the Bessel functions, which ProofKit defines from Mathlib's integrals and power series (Section 4.3). The Bessel functions' bounds are absolute errors in units of $u\,M_n(x)$, with $u = 2^{-53}$ and the modulus $M_n = \sqrt{J_n^2 + Y_n^2}$, which stays positive at the zeros of $J_n$ and $Y_n$ (index.md:223). The theorems of `jn` and `yn` exclude $2 \le |x| < 2^{302}$ with $|x| < 2|n|$. A proof for that range needs a bound on $M_k(x)$ for every order $k$ through the turning point $k = x$ (index.md:234). The theorem of `yn` also allows the infinity of $Y_n(x)$'s sign when $|Y_n(x)| > 2^{1023}$.

The step bounds of Table 1 are weaker than correct rounding. The exponential's guide notes that IEEE 754-2019 recommends a correctly rounded exponential (Library/Numerics/Float64/Exp/README.md:20). A test compares the exponential with a reference that has at least 400 fractional bits, and every result lies within one step of the reference's rounding (Library/Numerics/Float64/Exp/README.md:145–147). Some results differ from the correctly rounded value, so the exponential is not correctly rounded and the measured distance reaches the proved bound. The test of the sine and cosine also finds results that differ from the correctly rounded value (Library/Numerics/Float64/Trig/README.md:19).

### 3.3 Error bounds of sums, means, and linear algebra

Master theorems bound the error of the library's sums, means, sums of squares, dot products, matrix products, and discrete Fourier transforms against the exact real results (Library/Numerics/Reductions/Float64/Master.lean:65, Library/Numerics/LinearAlgebra/Float64/Master.lean:89, Library/Numerics/Fourier/Master.lean:107). The bounds have the form $\gamma_k \sum_i |t_i|$ for the terms $t_i$, with $\gamma_k = k/(2^{53} - k)$, which equals $ku/(1 - ku)$ (Library/Numerics/Reductions/SumError.lean:164). The bounds of products add an absolute term for results below the normal range. The binary32 functions use $\gamma_k = k/(2^{24} - k)$, as `axpyArray32_master` and `matVec32_master` state (Library/Numerics/LinearAlgebra/Binary32/Master.lean:113, Library/Numerics/LinearAlgebra/Binary32/Master.lean:187). Each bound holds for the order of operations that the program fixes, and the compiled module keeps that order, since the compiler's theorem states the Lean definition's exact result.

Table 1: The bounds of the master theorems, with $u = 2^{-53}$ and $M_n = \sqrt{J_n^2 + Y_n^2}$. A bound in steps is `WithinSteps` against the reference. The export `lgammaR` returns $\ln|\Gamma(x)|$ and the sign of $\Gamma(x)$. The Source column gives the origin of each function's code: fdlibm in FreeBSD's msun [39, 40], Arm's optimized routines [7], CORE-MATH [2], or leanexe. The rows come from `index.md`:182–234 and the master theorems.

| Function | Reference | Proved bound | Domain of the bound | Source |
|---|---|---|---|---|
| `exp`, `exp2`, `exp10` | $e^x$, $2^x$, $10^x$ | 1 step | finite $x$ | Arm |
| `expm1` | $e^x - 1$ | 1 step | finite $x$ | fdlibm |
| `log`, `log2` | $\ln x$, $\log_2 x$ | 1 step; `log2` exact at $2^k$ | $x > 0$ | Arm |
| `log10` | $\log_{10} x$ | 1 step; exact at $10^n$, $0 \le n \le 22$ | $x > 0$ | fdlibm |
| `log1p` | $\ln(1+x)$ | 1 step | $x > -1$ | fdlibm |
| `pow` | $x^y$ | 1 step | $x > 0$; $x < 0$, integer $y$; $x = 0$, $y \ge 0$ | Arm |
| `sin`, `cos`, `tan` | $\sin x$, $\cos x$, $\tan x$ | 1 step | finite $x$ | fdlibm |
| `sinpi`, `cospi`, `tanpi` | $\sin \pi x$, $\cos \pi x$, $\tan \pi x$ | 1 step | finite $x$; `tanpi`: no half-integer | msun |
| `atan`, `asin`, `acos` | $\arctan x$, $\arcsin x$, $\arccos x$ | 1 step | finite $x$; $\lvert x\rvert \le 1$ for `asin`, `acos` | fdlibm |
| `atan2` $y$ $x$ | $\arg(x + iy)$ | 2 steps | finite $x$, finite $y \ne 0$ | fdlibm |
| `sinh`, `tanh`; `cosh` | $\sinh x$, $\tanh x$; $\cosh x$ | 3 steps; 1 step | finite $x$ | fdlibm |
| `asinh`, `acosh`, `atanh` | inverse hyperbolic functions | 2 steps | finite $x$; $x \ge 1$; $\lvert x\rvert < 1$ | fdlibm |
| `erf`, `erfc` | ProofKit's erf, $1 - \text{erf}$ | 1 step | finite $x$ | fdlibm; CORE-MATH |
| `tgamma`, `lgamma` | $\Gamma(x)$, $\ln\lvert\Gamma(x)\rvert$ | 1 step | finite $x$, not 0 or a negative integer | CORE-MATH |
| `j0`, `j1` | ProofKit's $J_0$, $J_1$ | $10u\,M_0(\lvert x\rvert)$, $12u\,M_1(\lvert x\rvert)$ | finite $x$ | fdlibm |
| `y0`, `y1` | ProofKit's $Y_0$, $Y_1$ | $11u\,M_0$; $12u\,M_1$ for $x \ge 2^{-1022}$, 1 step below | $x > 0$ | fdlibm |
| `jn`, `yn` | ProofKit's $J_n$, $Y_n$ | $(23 + 12\lvert n\rvert)u\,M_{\lvert n\rvert}$, $(250 + 12\lvert n\rvert)u\,M_{\lvert n\rvert}$ | finite $x$ outside $2 \le \lvert x\rvert < \min(2^{302}, 2\lvert n\rvert)$; `yn`: $x > 0$ | fdlibm |
| `fma` $a$ $b$ $c$ | $ab + c$ | correctly rounded, with IEEE 754-2019's special cases | every triple | leanexe |
| `parse64`, `parse32` | the string's number | correctly rounded, or rejection | strings under $2^{32}$ bytes | leanexe |
| `print64`, `print32` | the word's value | shortest decimal that rounds to the word, nearest among the shortest | every word | leanexe |
| `total` | $\sum_i x_i$, $n$ terms | $\gamma_{n-1} \sum_i \lvert x_i\rvert$ | $\sum_i \lvert x_i\rvert \le 2^{1022}$ | leanexe |
| `mean` | $\frac{1}{n}\sum_i x_i$ | $\gamma_n \frac{1}{n} \sum_i \lvert x_i\rvert + 2^{-1075}$; NaN for $n = 0$ | $\sum_i \lvert x_i\rvert \le 2^{1021}$ | leanexe |
| `sumSquares` | $\sum_i x_i^2$ | $\gamma_n \sum_i x_i^2 + n\,2^{-1074}$ | $\sum_i x_i^2 \le 2^{1021}$ | leanexe |
| `dot`, `matVec`, `matMul` | $\sum_i x_i y_i$ per entry | $\gamma_n \sum_i \lvert x_i y_i\rvert + n\,2^{-1074}$ | $\sum_i \lvert x_i y_i\rvert \le 2^{1021}$ | leanexe |
| `matVec32` | $\sum_c m_{rc} v_c$, $k$ columns | $\gamma_k \sum_c \lvert m_{rc} v_c\rvert + k\,2^{-149}$ | $k < 2^{22}$, sums at most $2^{125}$ | leanexe |
| `dft` | $X_k = \sum_j x_j e^{-i\theta_{jk}}$ | $(\gamma_{n-1} + 2^{-46}) \sum_j (\lvert a_j\rvert + \lvert b_j\rvert) + n\,2^{-1071}$ | $x_j = a_j + ib_j$, sum at most $2^{1000}$ | leanexe |

### 3.4 Bounds on the WebAssembly, C, and AArch64 targets

The WebAssembly target receives each bound through a byte theorem, a theorem about the module's bytes. `exp_bytes_accuracy` states that the module's bytes decode to the module (`Library/Numerics/Float64/Exp/ExpTheorem.lean:309`), and a file theorem states that the emitted `exp.wasm` holds those bytes (`Library/Numerics/Float64/Exp/ExpFile.lean`). `exp_bytes_accuracy` also states that every call of `exp` from a ready store returns the bits of `Exp.exp x`, a word within one step of the reference, and leaves a ready store. The theorem therefore covers every call in a sequence of calls to `exp` from the initial store. Applied to an engine, the theorems assume that Talos describes the engine under the deterministic profile [14].

For each function `f` of a program `p`, the C translation has a theorem `p.f.native`, stated in the Lean copy of CompCert's Clight (`Verified/Reflect/Command.lean:518`). Like the module's theorem `p.f.implements`, it states that the function computes the Lean definition `f`. Leanexe's index lists a C form for every elementary and special function (`index.md:182–234`), so `compiled.exp.native` with `exp_one_step` gives the one-step bound for the C function's result. No named theorem of leanexe states that combination.

The C function's bound assumes the trusted base of the C target, which includes CompCert and the copy's agreement with CompCert's Clight [15]. The copy computes floating-point operations with Talos's functions and leaves each NaN result to the architecture's choice. The C translation replaces such a NaN result by the canonical NaN (`LeanExe/Native/Clight/Floats.lean`, `LeanExe/Native/Proof/Floats.lean`).

The AArch64 back end writes bare-metal boot Images, each of which computes one function on fixed arguments. `exp1_accuracy` states that, from boot, the Image of `exp` on 1 writes to the UART a word within one step of the correctly rounded value of $e$ (`tests/TestSupport/A64/ImageFilesExp.lean:98`). Further theorems state the same for `exp` on $-740$, whose result is subnormal, and for `sin` on 10 and $10^{22}$ (`tests/TestSupport/A64/ImageFilesSin.lean:84`). A test script, outside leanexe's build, checks these theorems. The AArch64 model's check against Arm's specification assumes equations between Arm's floating-point functions and Talos's [15].

The library's bounds cover no code for the Cortex-M7 or for GPUs. The Cortex-M7 back end rejects floating point, so no program with a float has a flash image (`docs/m7-bare-metal.md:571`). The GPU kernels come from a separate kernel language with binary32 arithmetic [14], and leanexe's index lists no GPU form of the library's functions.

## 4 Accuracy proofs: stage bounds, kernel-checked constants, and Mathlib's analysis

### 4.1 Stage bounds and the one-step criterion

Each accuracy proof follows the stages of its algorithm and bounds each stage's error against the exact function. The exponential, a port of Arm's code without fused multiply-add, writes $x = k \ln 2/128 + r$, takes $2^{j/128}$ from a table, and approximates $e^r - 1$ by a polynomial of degree 5 (`Library/Numerics/Float64/Exp/README.md:3`). Its proof bounds the polynomial within $2^{-56}$ of $e^r - 1$ for $|r| \le 1/256$ and the evaluation in floating point within $2^{-56}$ (`Library/Numerics/Float64/Exp/README.md:97–107`). It bounds the reconstruction $s + s \cdot tmp$ within $\frac{7}{5}us$ of the exact value. In that expression, the scale $s = 2^q H_j$ multiplies a table value $H_j$ by a power of two, and $tmp$ is the computed sum of a second table value and the polynomial (`Library/Numerics/Float64/Exp/Program.lean`).

The last lemma of each proof turns an error bound into a step bound. A word within one and a half spacings of the exact value lies within one step of its correctly rounded value, since rounding

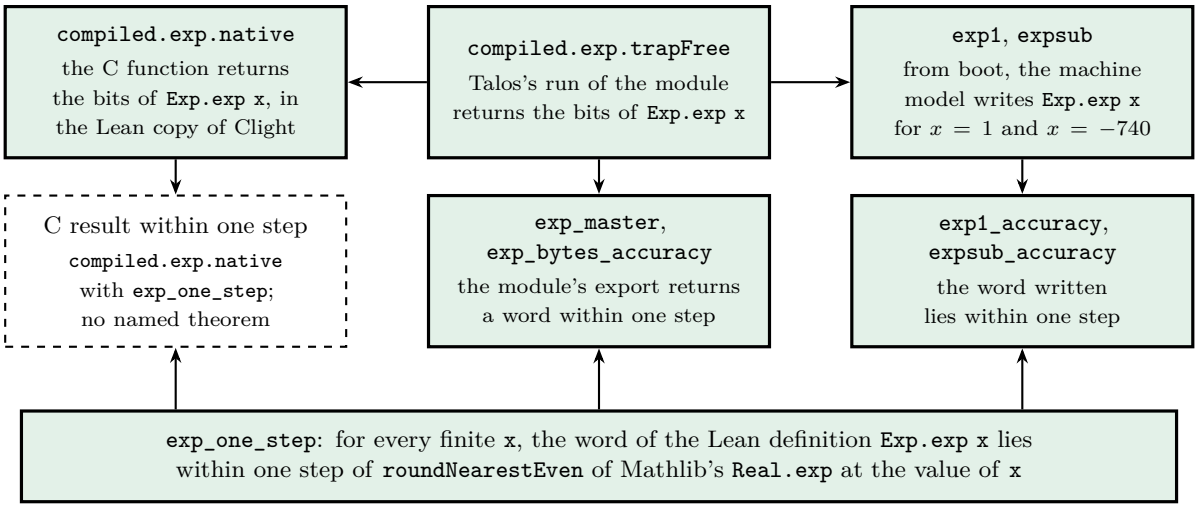



Figure 2: The exponential's bound on three targets. Boxes with heavy borders are theorems that Lean's kernel checks, and the dashed box is a combination that no named theorem of leanexe states. The arrows from the module's theorem in the upper row are the compositions that the companion report on leanexe's targets describes [15]: `translate_correct` for C and `program_sim` for AArch64. The other arrows are combinations with `exp_one_step`, the bound of the Lean definition. The Image theorems cover the arguments 1 and −740 only.

moves the exact value by at most half a spacing. `one_step_binades` proves this criterion across binade boundaries and the step to infinity (`LeanExe/ProofKit/F64FullStep.lean:164`). For the exponential's main path, the error is at most $2.79u\,2^q$ against an allowance of $3u\,2^q$, and at most $1.4u\,2^q$ against $1.5u\,2^q$ where the result may fall below $2^q$ (`Library/Numerics/Float64/Exp/README.md:109`).

## 4.2 Facts about constants checked by kernel evaluation

Lean's kernel checks facts about specific numbers in exact rational arithmetic, with the tactic `decide +kernel`. `PolyBound.check` bounds a polynomial with rational coefficients on a rational interval from its coefficients at the interval's center, and it bisects the interval when that bound is too large (`LeanExe/ProofKit/PolyBound.lean:57`). `check_sound` turns an accepted check into a bound at every real point of the interval (`LeanExe/ProofKit/PolyBound.lean:150`). The accuracy proofs compare each approximating polynomial with a Taylor polynomial by such checks. `checked_table_bounds` checks that each entry of the exponential's table lies within a relative $2^{-100}$ of $2^{j/128}$ (`Library/Numerics/Float64/Exp/ExpTableBounds.lean:26`).

The trigonometric reduction needs a lower bound on the distance of every binary64 value from the multiples of $\pi/2$. The guide of the trigonometric functions reports that Kahan and McDonald found the closest such value by a continued-fraction search, which K. C. Ng's report describes [30] (`Library/Numerics/Float64/Trig/README.md:29`). `quadrant_distance` proves that $x \cdot 2/\pi$ lies more than $2^{-63}$ from every integer for every binary64 $x$ from $1/2$ on, from one rational approximation per binade that the kernel checks (`Library/Numerics/Float64/Trig/TrigDistance.lean:1156`).

A Python program computes the approximations as convergents of continued fractions, and because the kernel checks each approximation, the proof does not trust that program (Library/Numerics/Float64/Trig/README.md:104).

### 4.3 Mathlib's functions and the definitions that Mathlib lacks

Mathlib supplies the reference functions and the analysis that turns a check on rationals into a bound on a real function. The exponential's proof specializes `Real.exp_bound`, Mathlib's bound on the Taylor remainder of the exponential (Library/Numerics/Float64/Exp/README.md:125). The gamma functions use Mathlib's `Real.Gamma`, and `lgamma_master` states its bound against `Real.log` of the absolute value of `Real.Gamma` (Library/Numerics/Float64/Lgamma/Master.lean:60).

Mathlib defines neither the error function nor the Bessel functions, and ProofKit defines both from Mathlib's primitives. `RealErf.erf x` is $(2/\sqrt{\pi}) \int_0^x e^{-t^2}\,dt$, an interval integral of Mathlib (LeanExe/ProofKit/RealErf.lean:25). ProofKit defines `Bessel.J0` and the other functions of orders 0 and 1 by their power series, following DLMF 10.2.2, 10.8.1, and 10.8.2 [29], with Mathlib's Euler–Mascheroni constant and harmonic numbers (LeanExe/ProofKit/Bessel.lean:51). These definitions are part of what the theorems state, so a reader must accept them as the intended functions.

## 5 Correctly rounded fused multiply-add and decimal conversion

The fused multiply-add, the parsers, and the printers have theorems that state their exact results, with no error bound. The proofs of the fused multiply-add and of the parsers count in units in which every operand is an integer, so that rounding becomes a statement about natural numbers. Each proof then shows that the code's inexact intermediate value rounds as the exact value does. An entry of leanexe's public verification knowledge base records this method (VKB/entries/correct-rounding/README.md) [16].

### 5.1 The software fused multiply-add

WebAssembly has no scalar fused multiply-add, and the CORE-MATH algorithms that the library ports for `erfc`, `tgamma`, and `lgamma` call one (Library/Numerics/Arithmetic/Fma/README.md:3). The library therefore computes $ab + c$ in 64-bit word arithmetic: an exact 106-bit product in a pair of words, the aligned addend with a sticky bit, and one rounding to nearest, ties to even. `fma_toBits` proves that the result equals `fmaReference` for every triple of `Float` operands (Library/Numerics/Arithmetic/Fma/FmaTheorem.lean:144). `fmaReference` rounds the real $ab + c$ with `roundNearestEven` and applies IEEE 754-2019's rules for zeros, infinities, and invalid operations, as the guide cites them clause by clause (Library/Numerics/Arithmetic/Fma/FmaTheorem.lean:23, Library/Numerics/Arithmetic/Fma/README.md:9–19). The word-level function returns the first NaN operand with its quiet bit set, and the `Float` wrapper returns the canonical NaN.

### 5.2 Decimal parsing and printing

`parse64` and `parse32` read a string in the number grammar of RFC 8259 or in a superset in the manner of C's `strtod` (Library/Numerics/Decimal/ParseFloat/README.md:11). `parse64_accept` states that, for every string of fewer than $2^{32}$ bytes that the grammar derives, the parser returns the binary64 word of the number rounded to nearest, ties to even. `parse64_reject` states that the parser rejects every other string (Library/Numerics/Decimal/ParseFloat/ParseFloat

Theorem.lean:497). The implementation uses Clinger's fast path for short inputs and otherwise Cox's unrounded scaling with exact multi-word integers, as its guide states with citations (Library/Numerics/Decimal/ParseFloat/README.md:5). It keeps at most 768 significant digits, and the proof shows that the digits it drops cannot change the rounding, since $(2^{54}+1)\cdot 5^{1075} < 10^{768}$ (Library/Numerics/Decimal/Shared/StrconvTruncate.lean:12).

`print64` and `print32` write a word as the shortest decimal that rounds to the word, the one nearest to the word's value among the shortest, with an even last digit at a tie (Library/Numerics/Decimal/PrintFloat/README.md:3). `Shortest64` states that property (Library/Numerics/Decimal/Shared/StrconvRound.lean:72). According to the printer's guide, the property is ECMA-262's rule for Number::toString with the choice that the rule's Note 2 recommends (Library/Numerics/Decimal/PrintFloat/README.md:7). `print64_bytes_accuracy` states the printer's theorems for the compiled module and every word (Library/Numerics/Decimal/PrintFloat/PrintFloatBytes.lean:140). The printer finds the largest decimal exponent at which a multiple of a power of ten lies in the word's rounding interval, with exact integers of any size (Library/Numerics/Decimal/PrintFloat/README.md:3).

# 6 Interval arithmetic with outward rounding

## 6.1 Outward endpoints from rounding to nearest

WebAssembly arithmetic rounds to nearest only, so the library obtains a lower or an upper bound by rounding to nearest and moving the result to the adjacent word below or above. `endpoint` performs that move, and it rejects a result or an endpoint that is not finite (LeanExe/ProofKit/F64Outward.lean:17). The library's checked operations are arithmetic operations that return a status and a word (Library/Numerics/Bounds/README.md, LeanExe/ProofKit/F64Outward.lean:40). A checked operation also rejects a nonfinite operand, a zero divisor, or a negative radicand, so a program tests one status at the end of a chain of operations.

Rump, Zimmermann, Boldo, and Melquiond give algorithms that bound or compute the neighbors of a floating-point number with rounding to nearest only [36]. They note that the methods may be used to simulate interval operations. Leanexe moves to the neighbor with an operation on the word's bits. Its endpoints can lie one step farther out than a directed rounding would put them.

## 6.2 Intervals and their soundness theorems

An `Interval` holds a status and two binary64 endpoints. The soundness theorems state `Contains i x`: the interval is accepted, its endpoints are finite, and the real number `x` lies between their values (Library/Numerics/Interval/Proof.lean:21). `add_sound` states that an accepted sum contains $x + y$ for every $x$ in the first operand and $y$ in the second, and the other arithmetic operations have theorems of the same form (Library/Numerics/Interval/Proof.lean:225). `mul` and `div` take the smallest of the four corner results rounded down and the largest rounded up, and they reject when any corner overflows (Library/Numerics/Interval/README.md).

The functions `increasing` and `decreasing` turn a one-step accuracy theorem of a library function into an enclosure of the function's image, by moving the computed values at the two ends two steps outward (Library/Numerics/Interval/Proof.lean:486). Two steps are the least widening that a one-step theorem supports (Library/Numerics/Interval/README.md:98). `two_up` proves that two steps suffice (Library/Numerics/Interval/Steps.lean:182).

Table 2: Programs on the exaspec examples page whose theorems enclose an exact quantity. Each name links its section of the page.

| Program | Quantity that the theorem encloses or bounds |
|---|---|
| Satellite collision probability | The exact value of the model's collision integral for the real inputs |
| Ground-station visibility | The satellite's elevation above a minimum angle for every position in a supplied box |
| Orbit tubes | Every solution of the equations of motion from the initial box, under every disturbance within its bound |
| Drone trajectories | Distances between drones and positions in corridors over whole intervals of time |
| Poses of a positioning stage | Every pose of the platform that fits the measured leg lengths |
| Covariance-intersection fusion | The exact fused covariance and center |
| Certified expected shortfall | The expected shortfall of every loss vector within the scenario intervals |
| Quadratic stress-limit certificates | The quadratic loss on the whole stress set |
| Compressible flow | The change of each conserved total: boundary fluxes plus a rounding residual with a computed bound |
| Certified residual reduction | The residual reduction of a binary64 run, through Crouzeix's inequality |
| Bound-state energies | The summed magnitudes of the bound-state energies, through the sharp Lieb–Thirring inequality |
| Capacity of a decaying qubit | The channel's classical capacity, in an interval no wider than the requested tolerance |
| Least energy of point arrangements | The energy of the triangular lattice, whose lower end bounds every arrangement of density one |

### 6.3 One definition for a real function and its interval program

The command `enclosure_def` reads a chain of `let` bindings over real expressions (`Library/Numerics/Interval/Enclose.lean:273`). It declares the real function, its interval function in the dialect, and a theorem that the interval function's result encloses the real function's result when each interval argument encloses the real argument. The tactic `enclose` proves that theorem by traversing the two chains of bindings together and proving the enclosure of each pair of bound values with the theorems labeled `enclose` (`Library/Numerics/Interval/Enclose.lean:243`). Leanexe's SGP4 demonstration defines the near-Earth part of the SGP4 satellite model with `enclosure_def` (`Demos/Sgp4/README.md`). Its program returns an interval for each coordinate of position and velocity, with a proof that the interval contains the model's value.

## 7 Certified enclosures in the public examples

The exaspec examples page [3] describes programs whose theorems state that a returned interval or bound contains an exact quantity of a mathematical model. Table 2 names those programs and the quantity that each encloses, and the page gives their details. According to the page, the proofs cover the WebAssembly code that leanexe's verified compiler produces. Several programs also run as C functions and AArch64 programs whose theorems state the same claims.

The satellite collision probability and the compressible-flow solver have public sources in leanexe. The collision program's theorem `pc_enclosure` states that an accepted interval contains the probability, an integral over the reals (`Demos/Conjunction/Soundness.lean:750`). According to the examples page, the following programs use theorems that OpenAI proved in Lean in 2026 and published, with papers, in its mathematics repository [35]. The program for residual reduction uses Crouzeix's inequality with the constant 2 [32] (Lean proof), and the program for bound-state energies uses the sharp one-dimensional Lieb–Thirring inequality [33] (Lean proof). The program for the qubit's capacity uses a theorem on the generalized amplitude-damping channel [31] (Lean

Table 3: Statements and trusted bases of leanexe's numerical library and related systems. The language or prover is in parentheses. Trusted bases follow each primary source, and italics mark this report's reading where the source states none.

| System | Statement | Trusted base |
|---|---|---|
| leanexe (Lean) | The compiled module's result lies within stated steps of the correctly rounded value of a real function or within a stated absolute error, or encloses a real value | Lean's kernel; Talos under the deterministic profile; the statements' definitions |
| Flocq [10] (Coq) | Formats and rounding over the reals | *Coq's kernel; the definitions* |
| Gappa [13] (C++, Coq) | Enclosures of expressions with rounded operators, with generated proofs | Coq and a proved library, or HOL Light with axioms |
| Coq Interval [27] (Coq) | Inequalities over the reals with elementary functions, by reflection | Coq's kernel and its reduction |
| FPTaylor [38] (OCaml, HOL Light) | Roundoff of a straight-line expression on a box, in a rounding model without overflow | The tool, or HOL Light for a certificate |
| CRlibm [12] (C) | Correctly rounded binary64 functions where worst cases are known | C99 and IEEE 754 compliance; paper and Gappa proofs; worst-case searches |
| CORE-MATH [21, 37] (C, Coq) | Correctly rounded functions, tested exhaustively or on worst cases; Coq for one phase of `pow` | *The compiler, the hardware, the tests' reference* |
| Arb [24] (C) | Ball arithmetic at arbitrary precision; inclusion by design and tests | *Arb and the libraries it uses* |
| Harrison [19, 20] (HOL Light) | Error bounds of Tang's exponential and IA-64 sine and cosine in a formal model | HOL Light's kernel; the model; the link to code |
| CompCert [11] (Coq) | Compiled C preserves the source's floating-point semantics | IEEE 754 hardware; CompCert's trusted parts |
| Isabelle [23, 42] | IEEE model; verified ODE enclosures with software floats | Isabelle; the code generator and PolyML for runs |
| RealCake, FloVer [8, 9] (HOL4, Coq) | Optimized kernels' machine code within a checked error bound of the real semantics | *HOL4 and CakeML's trusted base* |
| FloatLib [18] (Lean) | IEEE formats of arbitrary width; backends equal to encoded specifications | Lean's kernel; Lean's compiler for runs |

proof), and the program for the lattice's energy uses a theorem that the triangular lattice has the least energy among arrangements of density one [34] (Lean proof).

Each program's theorem on the examples page assumes that its inputs, such as boxes, intervals, or covariances, contain the true values of the modeled quantities. Each theorem also assumes, for the WebAssembly code, that the engine follows Talos's semantics. The page states a width bound only for the capacity program: a result marked successful is no wider than the requested tolerance.

## 8 Related work

Table 3 compares leanexe's numerical library with formalizations of floating-point arithmetic, tools that bound roundoff, correctly rounded libraries, ball arithmetic, and floating-point work in verified compilers. Leanexe's theorems differ from these works in two respects: they apply to the compiled module under Talos's semantics, and they compare its results with Mathlib's real functions. Trusted bases in the table are as the primary sources state them, and italics mark this report's reading where a source states none.

The tools that bound roundoff differ in what their certificates prove. Gappa generates a theorem and its proof for each bound, with Coq certificates that use a proved companion library and HOL Light certificates that use axioms [13, §2]. FPTaylor's rounding model $x(1 + e) + d$ with $ed = 0$ holds in the absence of overflow and invalid operations, and its optional HOL Light certificates prove

bounds relative to that model [38, §2, §6.2]. FloVer checks roundoff certificates for straight-line kernels without conditionals or loops [8, §VIII], and RealCake uses it to relate optimized CakeML machine code to the real-number semantics of the source [9, §2.4]. Leanexe's theorems cover programs with loops and tables, and they bound the error of the emitted code, whose operations the compiler does not rewrite.

CRlibm and CORE-MATH return correctly rounded results and support that claim with paper proofs, worst-case searches, and tests. CRlibm's report uses "proving" a function mostly for proving a bound on the relative error of its evaluation, with Gappa for some functions [12, §0.4, §1.3.5]. CORE-MATH checks univariate binary32 functions exhaustively and other functions against known worst cases [37, §IV-A], [1]. Hubrecht et al. verified in Coq the algorithms of the first phase of CORE-MATH's binary64 power function [21, §VII]. Leanexe ports CORE-MATH's algorithms for `erfc`, `tgamma`, and `lgamma`, and its theorems prove a bound of one step from the correctly rounded value for every input of the domain.

Harrison verified Tang's single-precision exponential in HOL Light, with an error below 0.54 units in the last place for normal results [19, §7]. He also verified Intel's double-extended sine and cosine for IA-64, within 0.07341 units in the last place before the final rounding [20, §6]. The exponential's paper states that its theorem concerns an algorithm and not an implementation [19, §1].

Yu's Isabelle model of IEEE arithmetic returns one unspecified NaN [42]. Immler's verified ODE solver in Isabelle certifies the computations of Tucker's proof about the Lorenz attractor with affine arithmetic [23, §8.3]. The certified computations run as generated code that PolyML executes.

Arb computes midpoint-radius enclosures at arbitrary precision for elementary and special functions, among them gamma and Bessel functions [24, §1]. Its paper describes inclusion as the defining property of interval arithmetic and tests that compare two computations of the same quantity, and it describes no machine-checked proof [24, §2, §5]. Leanexe's intervals have binary64 endpoints and soundness theorems that Lean's kernel checks, while Arb's balls reach arbitrary precision.

CompCert maps `float` and `double` to binary32 and binary64, prohibits reassociation and the contraction into fused multiply-adds, and parametrizes NaN results by the target architecture [11, §3.3, §5.2]. Its theorem preserves the floating-point semantics of the source program, and its targets' semantics assume that the hardware implements IEEE 754 correctly [11, §4, §6]. Leanexe's compiler also preserves the floating-point results of its source bit for bit, and the library's accuracy theorems bound those results against real functions.

FloatLib formalizes IEEE binary and decimal formats of arbitrary width in Lean, proves each backend equal to its encoded specification, and tests IEEE conformance [18, §1, §5.2]. Its paper states that its binary elementary-function kernels compute approximations whose certificates do not establish correct rounding [18, §4.2]. FloatLib's compiled code relies on Lean's compiler and runtime [18, §3.7], while leanexe's theorems cover the code that its own compiler emits.

## 9 Proofs, tests, and assumptions

Table 4 separates what Lean's kernel checks from what tests measure and what the theorems take without proof. The axioms of every theorem are `propext`, `Classical.choice`, and `Quot.sound`, and the build of leanexe's full check fails when a file reports another (`docs/design.md:115`). The author read the statements and documents at the pinned commit and did not rebuild the proofs.

Table 4: Proved, tested, and assumed items of the numerical library.

| Kind | Item | Source |
|---|---|---|
| Proved | Lean's float operations equal Talos's word functions on every input | `LeanExe/ProofKit/F64Bits.lean`:46 |
| | Roundoff of the basic operations below $2^{1022}$ or $2^{1023}$ | `LeanExe/ProofKit/F64AddBounds.lean`:23 |
| | A master theorem for each export: no trap under its bound, and the bound of Table 1 on its domain | `Library/Masters.lean` |
| | The fused multiply-add's, the parsers', and the printers' exact results | `Library/Numerics/Arithmetic/Fma/FmaTheorem.lean`:144 |
| | Soundness of each checked and interval operation and of each `enclosure_def` theorem | `Library/Numerics/Interval/Proof.lean` |
| | Byte and file theorems of the emitted modules | `Library/Numerics/Float64/Exp/ExpTheorem.lean`:309 |
| Tested | The exponential against a reference with at least 400 fractional bits: every result within one step | `Library/Numerics/Float64/Exp/README.md`:145 |
| | The fused multiply-add and the parsers against exact references, and the printers against an independent search | `Library/Numerics/Arithmetic/Fma/README.md`:3, `Library/Numerics/Decimal/ParseFloat/README.md`:5, `Library/Numerics/Decimal/PrintFloat/README.md`:3 |
| | Talos against the WebAssembly test suite [14] | `Library/Numerics/Float64/Exp/README.md`:151 |
| | The AArch64 floating-point forms against processors [15] | [15] |
| Assumed | Talos's functions are the WebAssembly instructions under the deterministic profile, and the decoder gives the bytes their meaning | `docs/design.md`:250 |
| | Lean's `Float.Model` gives `Float` its meaning | `docs/design.md`:189 |
| | Mathlib's definitions and ProofKit's error and Bessel functions are the intended functions | `LeanExe/ProofKit/Bessel.lean` |
| | A `Float` argument that is a NaN is the canonical NaN | `docs/design.md`:252 |
| | An enclosure program's inputs contain the true values | Section 7 |
| | For C and AArch64, the trusted bases in the companion report on leanexe's targets | [15] |

## 10 Limitations

The theorems concern Talos's semantics of WebAssembly under the deterministic profile. They do not describe a browser or another engine outside that profile, such as one on which `0/0` gives a negative NaN (`docs/design.md`:198). The accuracy theorems for `Float` arguments cover only the canonical NaN among the NaN words. No theorem covers conversions between `Float` and `Float32` (`docs/design.md`:200).

The integer-order Bessel functions have no bound for $2 \le |x| < 2^{302}$ with $|x| < 2|n|$. The interval theorems bound no width. The outward endpoints can lie one step farther out than a directed rounding would put them.

Lean 4.34.1's `min` and `max` on `Float` choose an operand by the comparison $\le$ [25], and the compiled code keeps that definition (`docs/design.md`:200). IEEE 754-2019's `minimum` and `maximum`, which return a NaN for a NaN operand, first appear in Lean 4.36.0 [26]. The interval library's `fmin` and `fmax` have theorems for finite arguments only (`Library/Numerics/Interval/Proof.lean`:64).

The AArch64 Image theorems cover fixed arguments of two functions, and a test script, outside the build, checks them. The C results' bounds follow from two theorems that leanexe does not

combine in a named theorem. The tests' references state their truncation errors in docstrings, and no theorem proves those errors (Library/Numerics/Float64/Exp/README.md:147). This report makes no claim about speed, and it describes the examples whose sources are private only as the examples page does.

## References


[1] CORE-MATH: correctly rounded mathematical functions. Project page, 2026. URL https://core-math.gitlabpages.inria.fr/. Read 27 November 2026.

[2] CORE-MATH. Git repository, commit ff2f71f6 of 10 October, 2026. URL https://gitlab.inria.fr/core-math/core-math/-/tree/ff2f71f635e95d395a6646b7e7b2c72e7b03a5d8.

[3] exaspec examples. Web page, 2026. URL https://exaspec.com/examples. Read 27 November 2026.

[4] State the proved error bounds in the numerics READMEs. Leanexe pull request, opened 27 November, 2026. URL https://github.com/jsmorph/leanexe/pull/36.

[5] Talos: a WebAssembly interpreter and code library in Lean. Git repository `jsmorph/talos`, commit 87e3aa5e, 2026. URL https://github.com/jsmorph/talos/tree/87e3aa5e8f6e6f3b3eb5e7e4c5aba43071002d47.

[6] WebAssembly core specification. Living specification, 2026. URL https://webassembly.github.io/spec/core/.

[7] Arm Limited. Optimized routines. Git repository, commit 503fafe3 of 2 October, 2026. URL https://github.com/ARM-software/optimized-routines/tree/503fafe311c177de0e571c458c7c337b1ca5f522.

[8] Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O. Myreen, and Anthony Fox. A verified certificate checker for finite-precision error bounds in Coq and HOL4. In *2018 Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2018. https://doi.org/10.23919/FMCAD.2018.8603019.

[9] Heiko Becker, Robert Rabe, Eva Darulova, Magnus O. Myreen, Zachary Tatlock, Ramana Kumar, Yong Kiam Tan, and Anthony Fox. Verified compilation and optimization of floating-point programs in CakeML. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*, volume 222 of *LIPIcs*, pages 1:1–1:28, 2022. https://doi.org/10.4230/LIPIcs.ECOOP.2022.1.

[10] Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic (ARITH)*, pages 243–252, 2011. https://doi.org/10.1109/ARITH.2011.40.

[11] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015. https://doi.org/10.1007/s10817-014-9317-x.

[12] Catherine Daramy-Loirat, David Defour, Florent de Dinechin, Matthieu Gallet, Nicolas Gast, Christoph Quirin Lauter, and Jean-Michel Muller. CR-LIBM: A library of correctly rounded elementary functions in double-precision. Research report, Laboratoire de l'Informatique du Parallélisme, École Normale Supérieure de Lyon, 2006. URL https://hal.science/ensl-01529804. HAL ensl-01529804.

[13] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):1–20, 2010. https://doi.org/10.1145/1644001.1644003.

[14] exaspec. Core leanexe: a verified compiler for a subset of Lean. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-core.pdf.

[15] exaspec. Leanexe's targets: composed theorems in stated semantics, with exact trusted bases. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-targets.pdf.

[16] exaspec. Leanexe's verification knowledge bases: reviewed entries whose cited declarations a checker resolves. Technical report, Morphism, 2026. URL https://exaspec.com/leanexedocs/leanexe-vkbs.pdf.

[17] *The GNU C Library Reference Manual*. Free Software Foundation, version 2.44 edition, 2026. URL https://sourceware.org/glibc/manual/2.44/html_node/Errors-in-Math-Functions.html.

[18] Robert Joseph George, Will Adkisson, and Anima Anandkumar. FloatLib: Verified floating-point arithmetic in Lean. Preprint arXiv:2609.19352v1, 16 September; no venue stated, 2026.

[19] John Harrison. Floating point verification in HOL Light: The exponential function. *Formal Methods in System Design*, 16(3):271–305, 2000. https://doi.org/10.1023/A:1008712907154.

[20] John Harrison. Formal verification of floating point trigonometric functions. In *Formal Methods in Computer-Aided Design (FMCAD 2000)*, volume 1954 of *Lecture Notes in Computer Science*, pages 254–270. Springer, 2000. https://doi.org/10.1007/3-540-40922-X_14.

[21] Tom Hubrecht, Claude-Pierre Jeannerod, Paul Zimmermann, Laurence Rideau, and Laurent Théry. Towards a correctly-rounded and fast power function in binary64 arithmetic. Preprint hal-04159652v2, 8 February; extended version of a paper in ARITH 2023, 2024. URL https://hal.science/hal-04159652v2.

[22] IEEE. IEEE standard for floating-point arithmetic. IEEE Std 754-2019, 2019. URL https://doi.org/10.1109/IEEESTD.2019.8766229.

[23] Fabian Immler. A verified ODE solver and the Lorenz attractor. *Journal of Automated Reasoning*, 61:73–111, 2018. https://doi.org/10.1007/s10817-017-9448-y.

[24] Fredrik Johansson. Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic. *IEEE Transactions on Computers*, 66(8):1281–1292, 2017. https://doi.org/10.1109/TC.2017.2690633.

[25] Lean FRO. Lean 4, tag v4.34.1. Git repository, 2026. URL https://github.com/leanprover/lean4/tree/v4.34.1.

[26] Lean FRO. Lean 4, tag v4.36.0. Git repository, 2026. URL https://github.com/leanprover/lean4/tree/v4.36.0.

[27] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016. https://doi.org/10.1007/s10817-015-9350-4.

[28] Morphism LLC. LeanExe. Git repository, commit 4556fa84 of 26 November, 2026. URL https://github.com/jsmorph/leanexe/tree/4556fa84205eee51b989e365a8519a33655833a2.

[29] National Institute of Standards and Technology. NIST digital library of mathematical functions. Release 1.2.8, Chapter 10, Bessel Functions, 2026. URL https://dlmf.nist.gov/10.

[30] K. C. Ng and the members of the FP group of SunPro. Argument reduction for huge arguments: Good to the last bit. Technical report, SunPro, 1992. URL https://userpages.cs.umbc.edu/phatak/645/supl/Ng-ArgReduction.pdf. Typeset version by Derek O'Connor, 2006.

[31] OpenAI. Classical capacity and entropy inequalities for generalized amplitude damping. Preprint dated 24 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/Classical-capacity-and-entropy-inequalities-for-generalized-amplitude-damping-September-24-2026/paper.pdf.

[32] OpenAI. A direct proof of the complete Crouzeix inequality. Preprint dated 26 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/A-direct-proof-of-the-complete-Crouzeix-inequality-September-26-2026/paper.pdf.

[33] OpenAI. Sharp one-dimensional Lieb–Thirring constants. Preprint dated 23 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/Sharp-One-Dimensional-Lieb-Thirring-Constants-September-23-2026/paper.pdf.

[34] OpenAI. An atomic certificate for triangular-lattice universal optimality. Preprint dated 26 September, 2026. URL https://github.com/openai/math/blob/3344da59991ed94f9ba711217b29c1d760badfa6/preprints/An-atomic-certificate-for-triangular-lattice-universal-optimality-September-26-2026/paper.pdf.

[35] OpenAI. Mathematics repository. Git repository, 2026. URL https://github.com/openai/math/tree/3344da59991ed94f9ba711217b29c1d760badfa6.

[36] Siegfried M. Rump, Paul Zimmermann, Sylvie Boldo, and Guillaume Melquiond. Computing predecessor and successor in rounding to nearest. *BIT Numerical Mathematics*, 49(2):419–431, 2009. https://doi.org/10.1007/s10543-009-0218-z.

[37] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondu. The CORE-MATH project. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 26–34, 2022. https://doi.org/10.1109/ARITH54963.2022.00014.

[38] Alexey Solovyev, Marek S. Baranowski, Ian Briggs, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. *ACM Transactions on Programming Languages and Systems*, 41(1):Article 2, 2018. https://doi.org/10.1145/3230733.

[39] Sun Microsystems. FDLIBM, version 5.3. Netlib, 2026. URL https://netlib.org/fdlibm/. Read 27 November 2026.

[40] The FreeBSD Project. msun, the FreeBSD math library. `lib/msun` in the FreeBSD source tree, commit 20381bce of 8 October, 2026. URL https://github.com/freebsd/freebsd-src/tree/20381bce4b63975494a2f4bc84257f6932e6d379/lib/msun.

[41] The Mathlib Community. Mathlib. Git repository, commit 85e3a25e, 2026. URL https://github.com/leanprover-community/mathlib4/tree/85e3a25e006c35636f0e53b0e9296caca2685bc0.

[42] Lei Yu. A formal model of IEEE floating point arithmetic. Archive of Formal Proofs, 2013. URL https://www.isa-afp.org/entries/IEEE_Floating_Point.html.