STAG: Smart Tools and Applications in Graphics (2025)
M. Comino Trinidad, C. Mancinelli, D. Cabiddu, and D. Giorgi (Editors)

TIGHT Intervals for Provably Correct Geometric Computation

Federico Sichetti[1], Marco Attene[2], Enrico Puppo[1]

[1] Università di Genova, Genoa, Italy
[2]CNR IMATI, Genoa, Italy

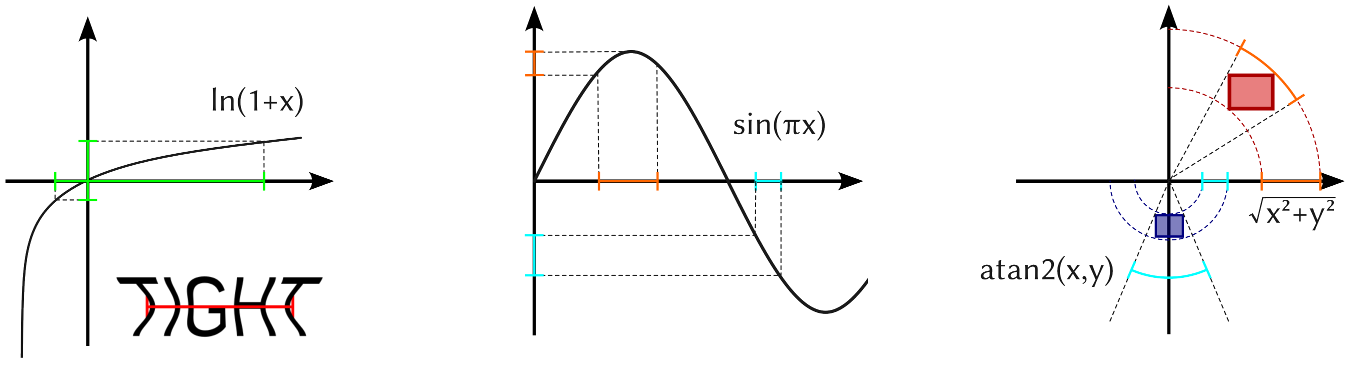

Figure 1: *Conservative evaluation of transcendental functions on a floating-point interval: (left) for a monotonic function, it is sufficient to properly round – one up and the other down – the values computed at the two endpoints of the interval; (center) for a non-monotonic function, we need to know the values of the maxima and minima within the interval (center); we also support common functions with two arguments, such as the distance of a 2D point from the origin, and the atan2 function (right). All floating-point evaluations are correctly rounded to warrant the tightest interval about the real value.*

Abstract

Interval arithmetic is a practical method for robust computation, bridging the gap between fast, but inexact, floating-point arithmetic and slow, exact arithmetic, such as rational or arbitrary-precision. In this system, numbers are represented as intervals bounded by floating-point numbers, and operations are performed conservatively, guaranteeing that the resulting interval contains the exact mathematical result. We extend a fast C++ library for interval arithmetic by adding support for several transcendental functions. A key feature of our library is that all operations are correctly rounded, ensuring the resulting interval is the smallest floating-point interval that contains the true result. We demonstrate the library's effectiveness by applying it to complex non-polynomial problems, including surface-surface intersection and continuous collision detection for geometric primitives undergoing roto-translational motion.

CCS Concepts

• ***Mathematics of computing*** → ***Interval arithmetic;*** *Mathematical software performance;* • ***Theory of computation*** → ***Rounding techniques;***

1. Introduction

Robust geometric computation is a critical component of many applications in graphics, such as collision detection, minimum distance computation, element inversion, and Boolean operations [SPH*25]. Although floating-point arithmetic is fast, it can easily generate inaccurate results that may lead to unpredictable, often catastrophic outcomes [Gol91, BB15]. In contrast, exact computation, using methods like rational arithmetic or arbitrary precision floating-point arithmetic, is extremely slow and often impractical. Moreover, such methods are exact only for algebraic operations.

Interval arithmetic bridges this gap by providing a viable alternative. It offers a balance between performance and accuracy, giving conservative estimates of exact computations at a moderate speed penalty compared to standard floating-point arithmetic. In this model, all real values are represented as intervals bounded by floating-point numbers. Expressions on these intervals are computed in a way that guarantees the resulting interval will contain the true mathematical result (Figure 1).

As the width of an interval grows during computation, it is crucial to minimize this propagation of uncertainty to maintain precision. This is achieved through *correctly rounded* (CR) operations, where each elementary operation returns the tightest possible floating-point interval that contains the exact result. Apart from CR operations yielding improved precision, correct rounding can ensure *bit-by-bit reproducibility*: because the result to be returned is well-defined, an expression evaluated with correct rounding will return the same result on any machine, and with any correctly-rounded implementations. Without this property, brittle code is subject to unexpected bugs that are hard to replicate, and results that are not consistent over time due to improvements to the underlying mathematical libraries.

The 2019 revision of the IEEE 754 standard for floating point arithmetic requires a compliant implementation of a function to round correctly for all inputs [iee19]. Indeed, required operations such as summation, subtraction, multiplication, division, and square roots produce the same, correctly rounded results on any IEEE 754-compliant machine. However, this is not true for *recommended* functions like sin or log: because they are not mandatory, mathematical libraries are allowed to implement fast, non-CR routines that are not IEEE 754-conforming, but the language implementation as a whole will be conforming as long as the mandatory operations are CR. As a result, most operations in existing mathematical libraries are not correctly rounded. We are not aware of any existing libraries that guarantee the creation of as-tight-as-possible intervals when the expressions involve this kind of operations. See Section 2 for further discussion.

In this paper, we describe the design principles of our TIGHT library for correctly rounded interval arithmetic, which always produces as-tight-as-possible intervals, is faster than any existing interval library, and supports transcendental functions. Our original contributions include:

1. We extend the NFG library [Att25] – which provides the most efficient implementation of interval arithmetic to date, but is limited to algebraic operations – with transcendental operations, based on the CORE-MATH floating-point correctly rounded implementation [SZG22]. In Section 3, we discuss the challenges involved in extending transcendental operators to intervals while guaranteeing correct rounding.
2. We integrate our library with the recent Domain Specific Language MiSo [SPH*25] that supports the fast prototyping of non-linear constraint solving and optimization. Extension of the language with transcendental functions largely broadens its spectrum of applicability.
3. We demonstrate the effectiveness and efficiency of our library by implementing surface-surface intersection between non-algebraic surfaces, and continuous collision detection between geometric primitives undergoing roto-translational motion. In Section 4, we also compare our library against the popular Filib++ library [LTG*06, LTG*11], achieving a faster performance.

We are committed to making TIGHT a usable piece of software: unlike a large share of other related libraries, TIGHT can be easily integrated into CMake projects. Its source code is available at https://gitlab.com/fsichetti/tight.

2. Background and state of the art

Interval arithmetic [HJVE01] provides a set of operations on real intervals $\mathbb{I}$ such that if $x \in a = [\underline{a}, \overline{a}] \in \mathbb{I}$ and $y \in b = [\underline{b}, \overline{b}] \in \mathbb{I}$, then $x \star y \in a \star b$, where $\star$ in the right-hand side is the interval version of operation $\star$ on reals. We are interested in intervals whose lower and upper bounds can be represented by FP numbers.

From now on, the set of representable FP numbers is denoted by $\mathbb{F}$. When a and b are in $\mathbb{F}$, the result $r = a \star b$ may not be in $\mathbb{F}$, hence not representable. In that case, $i = [fp^-(r), fp^+(r)]$, where $fp^-(r) = max(f : f \in \mathbb{F}, f \leq r)$ and $fp^+(r) = min(f : f \in \mathbb{F}, f \geq r)$, is the tightest representable interval containing r. As mentioned, IEEE 754 requires that when $\star$ is an algebraic operation (or the square root) an implementation of $\star$ must round the theoretically exact result r to either $fp^-(r)$ or $fp^+(r)$, depending on the current *rounding mode*. In most modern architectures, a particular register within the CPU controls the rounding mode, and specific system functions exist to set it. Therefore, a trivial approach to create a tight interval for the operation $a \star b$ is to (1) set the rounding mode to $towards - \infty$, (2) execute $a \star b$ to determine the interval's lower bound, (3) set the rounding mode to $towards + \infty$, (4) execute $a \star b$ to determine the interval's upper bound. Since setting the rounding mode is typically slower than executing arithmetic operations, a more efficient approach is (1) set the rounding mode to $towards + \infty$, (2) execute $(-a) \star b$ and switch the sign of the result to determine the interval's lower bound, (3) execute $a \star b$ to determine the interval's upper bound. Furthermore, if no other parts of the program require a different rounding, step (1) can be executed only once at the beginning. This approach is used by existing interval arithmetic libraries such as Boost [BMP06] and CGAL [Pro21]. However, note that this approach works for the arithmetic operations, because they are all odd functions (in the operand that changes its sign), but cannot be extended immediately to cope with functions that are not odd.

Another possibility is to deconstruct the binary representation of the result r to directly modify the mantissa, exponent, and sign, and produce a reasonably small interval around r. This approach is used by Filib [LTG*06, LTG*11]. Alternatively, the error propagation can be analyzed to derive a bound ϵ on the rounding so that the interval $i = [r - \epsilon, r + \epsilon]$ is guaranteed to contain r. This is how libraries such as BIAS [Knü94] or GAOL [Gou16] work. Library Filib++ [LTG*06, LTG*11] offers several modes that implement the various strategies; according to the authors, the fastest is the *native_onesided_global* mode, which adopts the strategy based on a fixed rounding mode and change of sign described above.

The aforementioned existing libraries were comprehensively compared by Tang and colleagues [TFS*22] who evaluated diverse aspects, including their correctness, efficiency and precision (in terms of interval tightness). They conclude that only Filib and

Filib++ are always correct when transcendental functions are involved, although the intervals they produce might be larger than necessary. In contrast, Boost and BIAS may produce intervals that do not contain the exact result. Also, Tang's evaluation could verify that libraries that use the rounding mode produce tighter intervals. In our experiments, we compare against Filib++ with the fastest mode.

Except for rather old architectures, most existing CPUs provide SIMD registers and instructions that proved useful to accelerate interval arithmetic libraries [Lam08]. The basic idea is to store both bounds in a single 128bit-wide register and perform operations on them in parallel. This and other optimizations exploiting more recent AVX architectures were included in the NFG library [Att25] that, to the best of our knowledge, represents the fastest existing library at the time of writing. Since NFG exploits the rounding mode, it is also guaranteed to produce as-tight-as-possible intervals for all the algebraic operations and the square root. Our TIGHT library wraps around NFG while adding many other elementary and transcendental functions, while keeping the guarantee to produce tight intervals.

2.1. Correct rounding

Correct rounding (CR) refers to the property that an implementation of a mathematical function f has if, for any x that is representable and contained in the domain of f, it returns the same results one would get by rounding the exact result $f(x)$ to the target representation.

Producing efficient CR implementations of functions is a difficult task that has been actively researched for many years. The classical method involves performing a fast approximation of the function with a known error bound, followed by a correctness check, and a slower but higher-accuracy approximation if the check fails. Interestingly, implementing CR double-precision functions is more difficult than single-precision, because one can check correct rounding exhaustively on 32-bit floats, which is infeasible in the 64-bit case. For double precision, correctness must either be proved formally or tested on known hard-to-round cases.

The most notable libraries for CR mathematical functions are CRlibm [DLDdM03] and RLibm [LN22]. CRlibm implements several transcendental functions of one double-precision argument, correctly rounded in the four rounding modes *towards* $+\infty$, *towards* $-\infty$, *towards zero*, *to nearest*. However, instead of rounding according to the CPU setting, it provides separate functions for each rounding mode and assumes that the CPU is set to the default *to nearest* mode with ties-to-even (whereas interval arithmetic uses directed rounding). To the best of our knowledge, the project is no longer actively maintained. RLibm is a more recent project that proposes a new approach to correct rounding by polynomial approximants, but it is currently limited to single-precision inputs.

The CORE-MATH Project [SZG22] is an ongoing effort to build a complete collection of correctly rounded C implementations of mathematical functions to foster integration into existing mathematical libraries. CORE-MATH is actively developed and provides efficient CR routines for univariate and bivariate functions with double-precision arguments. For a thorough account of correct rounding we refer the interest reader to a recent survey [BHMZ25].

TIGHT uses CORE-MATH functions in its interval extensions for all those functions that are not CR in the C++ standard library.

3. Implementing elementary functions

TIGHT's interval class wraps the NFG interval library [Att25], which efficiently supports CR interval computation, limited to the arithmetic operations, the square, and the square root, i.e., those floating-point operations for which the IEEE 754 standard prescribes correct rounding. We extend the scope of the library to also support transcendental functions, exploiting the results of the CORE-MATH Project [SZG22], which provides CR floating-point implementation of the most common transcendental functions.

Our major contribution consists of implementing CR *interval* functions also for those transcendental functions, and providing a whole support to both polynomial and transcendental interval computation within a unified context. The functions currently supported by TIGHT intervals are:

- basic arithmetic: $x+y$, $x-y$, xy, x/y, $-x$, $|x|$, $\max(x,y)$, $\min(x,y)$;
- power functions: x^2, $\sqrt{x}$, $\sqrt[3]{x}$, $1/\sqrt{x}$, and the generic x^y;
- trigonometric functions and their inverses: $\sin(x)$, $\cos(x)$, $\tan(x)$, $\arcsin(x)$, $\arccos(x)$, $\arctan(x)$;
- trigonometric functions with scaled argument: $\sin(\pi x)$, $\cos(\pi x)$, $\tan(\pi x)$, $\arcsin(x)/\pi$, $\arccos(x)/\pi$, $\arctan(x)/\pi$;
- hyperbolic functions and their inverses: $\sinh(x)$, $\cosh(x)$, $\tanh(x)$, $\operatorname{arcsinh}(x)$, $\operatorname{arccosh}(x)$, $\operatorname{arctanh}(x)$;
- exponentials in base e, 2 and 10: e^x, 2^x, 10^x, e^x-1, 2^x-1, 10^x-1;
- logarithms in base e, 2 and 10: $\log(x)$, $\log_2(x)$, $\log_{10}(x)$, $\log(1+x)$, $\log_2(1+x)$, $\log_{10}(1+x)$;
- functions to convert to polar coordinates: $\sqrt{x^2+y^2}$, $\operatorname{arctan2}(x,y)$;
- the error functions $\operatorname{erf}(x)$, $\operatorname{erfc}(x)$.

3.1. Interval extension

Assuming the availability of a correctly rounded function f, we want to obtain a correctly rounded inclusion function for f, that is, an interval-valued function $\Box f$ such that the endpoints of the resulting interval are correctly rounded outward.

If an input contains NaNs, infinity, or points that are outside the domain of f, we return a NaN interval. Other libraries opt to provide some extended definition of operators to handle such inputs; we postpone this to future work. In the following, we limit our discussion to valid inputs.

The challenge of extending a function to intervals with correct rounding is twofold. First, we need an expression for the range of the function; this is obtained by enumerating the possible cases for a given function. Then, these expressions must be instantiated on the input datum and rounded correctly - downward for the lower bound and upward for the upper one.

3.1.1. Computing extensions

When f is monotonically increasing on $[\underline{x},\overline{x}]$, the range of the function on an interval is easily obtained as $\Box f([\underline{x},\overline{x}]) = [f(\underline{x}), f(\overline{x})]$ (Figure 1 left); if it is monotonically decreasing, the two endpoints are swapped. If f is not monotonic on $[\underline{x},\overline{x}]$, we need to know where f attains its extrema on the interval. As we will see for some functions, even deciding that $[\underline{x},\overline{x}]$ lies in a part of the domain where f is monotonic is tricky in floating-point arithmetic.

Once we know how to compute the range of the function in exact arithmetic, correct rounding amounts to rounding the left endpoint down and the right endpoint up. Given that we are operating in round-upward mode, the latter is free. To round down we can always change the rounding mode and reset it after the operation. However, changing rounding modes flushes the CPU pipeline, thus it is a relatively expensive operation. Fortunately, it can be avoided in most cases:

1. If f is odd, the result of $f(x)$ rounded down can be computed in upward rounding mode as $-f(-x)$, since negation is always exact (it only changes the sign bit). This is the same technique used by NFG for arithmetic operations.
2. For non-odd functions, if we know the values of x for which $f(x)$ is representable in floating-point, we can check *a priori* if (upward) rounding happens, and if it does, we take the next smallest floating-point value.
3. Only for the remaining functions, we do change the rounding mode to downwards rounding, and reset it to upwards after computing the lower bound.

3.1.2. Computing $\arcsin(x)$, $\arctan(x)$, $\arcsin(x)/\pi$, $\arctan(x)/\pi$, $\sinh(x)$, $\tanh(x)$, $\operatorname{arcsinh}(x)$, $\operatorname{arctanh}(x)$, $\sqrt[3]{x}$, and $\operatorname{erf}(x)$

All these functions are odd and monotonic. For an odd, monotonically increasing function f for which we have access to a CR implementation, the correctly rounded $\Box f$ is $\Box f([\underline{x},\overline{x}]) = [-f(-\underline{x}), f(\overline{x})]$.

3.1.3. Computing $\tan(x)$ and $\tan(\pi x)$

Because the tangent is an odd function, rounding down is not a problem. The only difficulty in this case is that x may be an interval that crosses one of the vertical asymptotes of the function, so additional checks are needed. Our logic works as follows:

1. if x has a width larger than a period, then it must contain a singularity and we stop;
2. otherwise, we compute the function at the endpoints, and because the two points differ by less than a period, x can only contain a singularity if $\tan(\underline{x}) > \tan(\overline{x})$, in which case we stop;
3. if the previous tests passed, we return the intervals with the endpoints we already computed.

The same technique applies to the scaled version.

3.1.4. Computing $\arccos(x)$, $\arccos(x)/\pi$, $\operatorname{arccosh}(x)$, $\operatorname{arccosh}(x)/\pi$, e^x, e^x-1, 2^x, 2^x-1, 10^x, 10^x-1, $\log(x)$, $\log(1+x)$, $\log_2(x)$, $\log_2(1+x)$, $\log_{10}(x)$, $\log_{10}(1+x)$ and $1/\sqrt(x)$

For these monotonic functions we can devise a fast test that checks whether the image of x is exactly representable Denote $R \subset \operatorname{Dom} f$ the set of FP values such that every element of $f(R)$ is a FP number. The lower and upper bound are both computed with upward correct rounding; if $x \in R$, the lower bound is left unchanged, otherwise it is changed to the next smallest representable value.

To get the next smallest representable value, we could use the `nextafter` function offered by the standard library, which operates directly on the bit representation of the number. However, because we are using upward rounding across our program, it is slightly faster to compute the floating point number immediately before y as $-(\epsilon - y)$ where ϵ is the smallest positive representable number.

The list of all representable values for each of the functions listed above is documented in the source code of the library and omitted here for brevity.

3.1.5. Computing $\cosh(x)$ and $\sqrt{x^2+y^2}$

The hyperbolic cosine is the only single-argument U-shaped function in our set (except x^2, which is supported by NFG already). To compute it, we take the absolute value of x (defined as the interval of all possible values of $|y|$ for all $y \in x$) and see if it contains 0. If it does, the lower bound is 1; otherwise it is $\cosh(\underline{|x|})$ rounded down as in the previous section (unconditionally, since $\overline{0}$ is the only number in R). The upper bound is simply $\cosh(\overline{|x|})$.

The Pythagorean sum (also known as `hypot`) is similar to $\cosh(x)$, but in two variables. It returns the distance of the closest and farthest point in the 2D box from the origin, rounded down and up respectively (Figure 1 right). We compute $|x|$ and $|y|$ and test if any of them contains zero; in such case, the minimum distance is simply $\max(\underline{|x|}, \underline{|y|})$, where at least one of these numbers is zero. Otherwise, since we have no fast and reliable way of knowing whether the distance is a representable number, we resort to changing the rounding mode to compute the lower bound. The upper bound is easily computed in any case with a call to the CORE-MATH function.

3.1.6. Computing $\sin(x)$, $\cos(x)$, $\sin(\pi x)$ and $\cos(\pi x)$

Perhaps surprisingly, sines and cosines are the hardest functions to implement.

Let us discuss rounding issues first. $\sin(x)$ and $\sin(\pi x)$ are odd and rounding down is trivial. We know that the only value of x for which $\cos(x)$ is a FP number is 0, so we can round down as in Section 3.1.4. For $\cos(\pi x)$, the set R consists of rational numbers with denominator 2, so we check if $2x$ is integer (multiplication by 2 is exact) and round down as before.

To compute the interval inclusion, we first check if x is larger than a period of f, in which case we know it must contain at least two consecutive extrema, and we can return $[-1,1]$. Otherwise, we need to know the sign of f' at the two endpoints of x. If the signs are equal, then x contains either 0 or 2 critical points; specifically, it contains 2 critical points if its width is at least π (and again we return $[-1,1]$), otherwise it contains none. If the signs are different, then x contains a single critical point (Figure 1 center). This leaves us with four cases to consider:

- x lies entirely within part of the domain where f is monotonically increasing, so we return $[f(\underline{x})_-, f(\overline{x})]$;

- x lies entirely within part of the domain where f is monotonically decreasing, so we return $[f(\overline{x})_-, f(\underline{x})]$;
- x contains a minimum, so we return $[-1, \max(f(\underline{x}), f(\overline{x}))]$;
- x contains a maximum, so we return $[\max(f(\underline{x})_-, f(\overline{x})_-), 1]$.

The cases where x has an endpoint at an extremum do not require special handling. This leaves us with the issue of how to efficiently compute the sign of f' at the endpoints of x.

For $\sin(\pi x)$ and $\cos(\pi x)$, the derivative changes sign at integer multiples of $1/2$, so we can check the value of $\lceil 2\underline{x} \rceil \bmod 4$ (and likewise for $\overline{x}$, respectively); depending on whether we are computing the sine or cosine, two of the possible values correspond to a positive derivative and the other two to a negative one. To correctly get this result, we compute the ceiling operator by converting the endpoints, multiplied by 2, to 64-bit integers; overflows are not a problem, since past the value of 2^{54} the distance between adjacent double-precision FP numbers is at least as large as a full period of the function, so non-singleton intervals will return $[-1, 1]$ anyway and can be handled as a special case.

For the regular sin and cos, we cannot apply the same strategy. One would like to know the consecutive integer multiples of $\pi/2$ that contain a floating point value, but since π is irrational we would need a correctly rounded function to compute x/π, or at least πx. While such methods have been researched in previous work [BM05], they are not part of CORE-MATH and are beyond the scope of this work. Instead, we compare $\underline{x}$ and $\overline{x}$ with precomputed, correctly rounded multiples of $\pi/2$ in the range $[-2\pi, 2\pi]$, and if an endpoint is beyond this limit, we call the correctly rounded function corresponding to the derivative of cos or sin, which is more expensive but gives correct results. For this reason, TIGHT's current implementation of sin and cos is relatively cheap if x lies in $[-2\pi, 2\pi]$ and somewhat slower otherwise.

3.1.7. Computing $\mathrm{erfc}(x)$

For erfc, we are not aware of an easy way to check if the image of a number is representable. The function is monotonically decreasing, so we implement it with a single change in rounding mode.

3.1.8. Computing x^y and $\mathrm{arctan2}(x, y)$

We provide two versions of the power function: one where x and y are both intervals, but x is not allowed to have negative values, and one where x can be any interval but y is an unsigned integer.

For the integer power function, we have a special case for $y = 2$ that calls NFG's fast square function. In other cases, we test the parity of the exponent and perform similar procedures to the other even/odd functions discussed previously.

For the general power function, we distinguish nine cases based on the position of the 2D box (x, y) on the $x \geq 0$ half-plane: whether x is above, below or contains 1, and whether y is above, below or contains 0. For eight out of nine cases, we only require two calls to CORE-MATH's power function to compute the result; in the worst case, i.e., when (x, y) contains $(1, 0)$, we need four.

The two-argument arctangent, i.e. the range of angles between the positive horizontal semi-axis and the line formed by connecting the origin with points in (x, y) (Figure 1 right); the values are in $(-\pi, \pi]$, requires a similar distinction of nine cases, only this time x can be negative and so we check whether each intervals is above, below or contain 0. Again, we do this to limit the number of calls to the mathematical library, and in eight out of nine cases, we perform two calls. The ninth case, where (x, y) contains $(0, 0)$, is degenerate, and we return NaN. However, one of the non-degenerate cases is very problematic, namely when x is in the negative half-space and y contains 0. In this case, the range of angles contains the points of angle π, and the function is discontinuous. Mathematically, one should return the whole range $[-\pi, \pi]$, but the meaningful result would be the disjoint union $[-\pi, a] \cup [b, \pi]$ for some values a, b. To give meaningful results, we opt for a different solution: TIGHT's arctan2 function returns both an interval and a boolean flag that signals the pathological case, and when the flag is on, we instead return the *complement* of the range of angles of (x, y), i.e. the ones that are not spanned by points in the box, with consequently inverted rounding. In the previous notation, we return $[a, b]$ with a rounded up and b rounded down. A caller that expects points in this range should then check the flag and decide how to use this result.

4. Results and comparison

We evaluate our library in several scenarios. Furthermore, because Filib/Filib++ is the only library which is always correct when transcendental functions are involved [TFS*22], we compare TIGHT with this representative of the state of the art.

All experiments were timed single-threaded on a server equipped with Intel Xeon Gold 6430 CPUs and 64GB of RAM. The compiler used is GCC version 13.3.0 on Ubuntu.

4.1. Benchmark comparison

We start by comparing the execution times and average interval width of single functions in TIGHT and Filib++. To compute the width, for each operation we select one or two singleton input intervals that lie inside the function domain, and compute the number of floating point values between the lower and upper ends of the computed interval (plus one); a width of 1 ULP (or 0, in the case of an exactly representable output) corresponds to a correctly rounded result. To get timings, we call each function one million times; to prevent the compiler from optimizing calls, and to evaluate the timing on non-singleton intervals, we perturb the input interval at each iteration by enlarging it by an ULP on both sides, and sum the upper and lower bound of each result to a dummy accumulator. Note that some operations are omitted for Filib++ as they are not supported in the library.

Estimated times for executing one interval operation with the two libraries are reported in Table 1. Times are in nanoseconds. Note that, for the simplest operations that cost about or even less than one nanosecond, estimates are not fully reliable and may change on different runs. The values reported in the table for those operations are averages over different runs. However, we consistently had at all runs faster times with TIGHT, for all those operations that are already supported in NFG. For the newly implemented transcendental functions, times are comparable, but sometimes they are slower with TIGHT. This is the cost of all additional

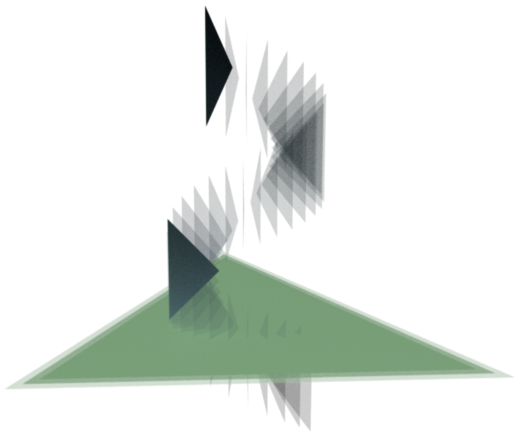

Figure 2: *Test problem: CCD of a triangle moving rigidly along a screw trajectory, and another deforming triangle. The triangle on top rotates and moves along an axis until the two primitives come into contact.*

operations that we undergo to guarantee correct rounding, which is not guaranteed by Filib++.

The latter issue is evident by looking at the two rightmost columns in Table 1, which report the width of the resulting interval in ULPs (number of contiguous floating-point values), when the input consists of a singleton $[x,x]$. While TIGHT always produces intervals with a width of either zero or one ULP, hence correctly rounded, Filib++ produces quite large intervals in many cases (it is still correctly rounded, but slower, for algebraic operations).

4.2. Comparison with Filib++ within MiSo

To assess the practical effectiveness of our library, we integrate it into the software MiSo [SPH*25], which makes intensive use of interval arithmetic. MiSo is a Python tool to generate interval-based C++ solvers with correctness guarantees, which uses NFG for its fast interval arithmetic, albeit with no support for non-algebraic operations. Since MiSo's architecture allows for easily switching the numeric backend, we were able to perform a side-by-side comparison of our library and Filib++ for several problems, while extending the software's support to transcendental operators.

The problems presented in the following involve heavy use of trigonometric functions that, as outlined above, are more expensive to compute in TIGHT than in Filib++. However, as we shall see, TIGHT's result in faster solution times overall, thanks to faster handling of basic arithmetic and its smaller intervals.

4.2.1. CCD along non-algebraic trajectories

We consider the classical problem of continuous collision detection (CCD): given two primitives in space with some prescribed trajectory, we seek the first time in $[0,1]$ for which the primitives first come into contact, and call it t^*. This type of test is essential in the simulation field to guarantee that physical objects do not interpenetrate. In practice, finding the exact value of t^* may be infeasible and often unnecessary. With interval methods, we instead seek an

Table 1: *Benchmark for single interval operations: times to execute a single operation (in nanoseconds) with TIGHT and Filib++ and the related width in ULPs of the interval returned when the operands are singleton intervals of the type* $[x,x]$.

Operation	Avg. ns		width in ULPs	
	TIGHT	Filib++	TIGHT	Filib++
`+`	0.28	1.08	1	1
`-`	0.34	0.71	0	0
`*`	0.66	1.13	1	1
`/`	0.89	1.30	1	1
`min`	0.30	-	0	-
`max`	0.30	-	0	-
`abs`	1.45	2.11	0	0
`sin`	98.20	17.34	1	23
`cos`	98.48	17.50	1	28
`tan`	128.41	18.59	1	52
`asin`	36.49	13.99	1	39
`acos`	20.15	13.44	1	36
`atan`	38.41	10.33	1	28
`sinpi`	30.12	-	1	-
`cospi`	31.37	-	1	-
`tanpi`	29.37	-	1	-
`asinpi`	40.24	-	1	-
`acospi`	44.97	-	1	-
`atanpi`	41.97	-	1	-
`sinh`	10.84	11.19	1	20
`cosh`	12.94	34.28	1	11
`tanh`	20.08	10.90	1	23
`asinh`	8.33	32.40	1	19
`acosh`	31.79	15.58	1	35
`atanh`	22.39	25.84	1	27
`exp`	14.84	17.68	1	9
`exp2`	17.90	18.59	1	9
`exp10`	19.61	18.28	1	10
`expm1`	15.06	10.51	1	14
`exp2m1`	66.32	-	1	-
`exp10m1`	78.21	-	1	-
`log`	39.91	10.89	1	9
`log2`	15.81	11.14	1	50
`log10`	58.41	13.76	1	44
`log1p`	15.36	14.59	1	12
`log2p1`	61.70	-	1	-
`log10p1`	69.79	-	1	-
`sqrt`	2.18	3.78	1	2
`cbrt`	18.78	-	1	-
`rsqrt`	21.32	-	1	-
`erf`	60.70	-	1	-
`erfc`	62.67	-	1	-
`pow`	136.80	36.31	1	15
`hypot`	39.20	-	1	-
`atan2`	28.07	-	1	

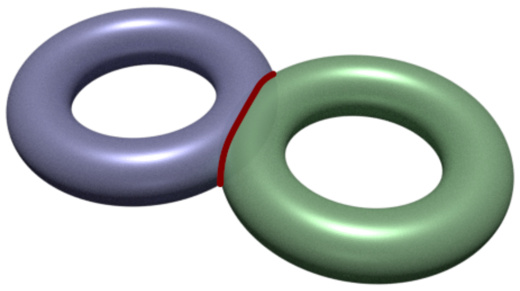

Figure 3: *Test problem: surface-surface intersection (SSI) of two parametric tori of inner radius* 0.3 *and outer radius* 1*, each shifted by* ± 1.1 *along the x axis.*

interval T^* that is guaranteed to contain t^* and is smaller than a user-specified precision $\delta > 0$. Then the lower bound of T^* tells us a moment in time until which we can safely move the objects without collisions, and the upper bound gives a moment when a collision has surely happened already. The difficulty of the query depends on the type of primitives, the type of trajectory, and the precision required.

Within MiSo, this is formulated as a MINIMIZE problem, that is, a constrained global optimization [SPH*25]. In this case the optimization variables are $\{\mathbf{u},\mathbf{v},t\}$, where $\mathbf{u} = (u_0,u_1)$ and $\mathbf{v} = (v_0,v_1)$ are the parametric coordinates of the two primitives, and t is time; the constraints are the domain constraints $(u_0,u_1,v_0,v_1,t) \in [0,1]^5, u_0+u_1 \leq 1, v_0+v_1 \leq 1$ (which are all implicit in MiSo), and the collision constraint $d(x,y) < \epsilon$ with a small but positive ϵ – i.e., we only consider pairs of points for which collision happens. The objective function is t – i.e., we want to find the collision that happens earliest.

In our test, we consider two moving linear (i.e., flat) triangles, where one is linearly deforming (i.e., each vertex follows a linear trajectory independent of the others), while the other vertex is undergoing a rigid roto-translation, following a spiral motion (Figure 2). More precisely, the center of the second triangle follows the spiral, and its normal remains aligned with the spiral's tangent. Of course, computing the roto-translation requires trigonometric functions, while all other computations involved are algebraic.

We test the same problem with different sets of parameters, changing the speed of the rotation and the position of the other triangle. The queries implemented with TIGHT take 27ms, 133ms and 2.5s respectively, versus the 127ms, 756ms and 13.2s of Filib++, for a speedup of approximately $5\times$.

4.2.2. Intersection of two parametric tori

Computing the intersection of parametric surfaces, also known as surface-surface intersection (SSI), is an important operation in CAD. In our example, we want to describe the intersection locus of two tori expressed parametrically (Figure 3). Again, the parametric representation of each torus requires computing trigonometric functions.

SSI can be formulated in MiSo as a SOLVE problem, that is, the problem of covering the set of all solutions of a constraint system within a certain tolerance. Being a conservative method, the algorithm returns a region that is larger than the true solution, but is guaranteed to contain every point of it.

Similarly to the previous example, the optimization variables are $\{\mathbf{u},\mathbf{v}\}$, where $\mathbf{u} = (u_0,u_1)$ and $\mathbf{v} = (v_0,v_1)$ are the parametric coordinates of the two tori; the constraints are the domain constraints $(u_0,u_1,v_0,v_1) \in [0,1]^4$ (implicit in MiSo), and the intersection constraint $d(x,y) < \epsilon$. This formulation gives solutions in *parameter* space rather than physical space. To be precise, this formulation solves a more general version of the problem: since the parameter space is 4-dimensional, the solution is a collection of 4D boxes that describe pairs of contacting regions, up to the specified tolerance.

In both cases, the solver performs 11105 iterations and found a solution composed of 6976 4D boxes. However, the result took 136ms to compute with TIGHT and 2186ms with Filib++, a speedup of about $16\times$.

4.3. Pathological cases

Not producing tight enough intervals can have unpredictable results for seemingly easy problems. We provide a very simple example where returning a slightly larger interval results in the program being unable to compute the result.

Suppose we have a function $f(x,y)$ defined on the plane that is inversely proportional to the distance of point (x,y) to a circle centered in the origin with radius 5 (Figure 4). Thus, we are evaluating $f(x,y) = 1/(\sqrt{x^2+y^2}-5)$. This type of function resembles contact potentials used in IPC physical simulations [LFS*20]. We want to evaluate f with interval arithmetic at point $P(3+10^{-15}, 4+10^{-15})$, so that P lies outside the sphere. Remember that, because we are using interval arithmetic, we need not fear that adding a small value leads to cancellation for very small ϵ: in those cases we

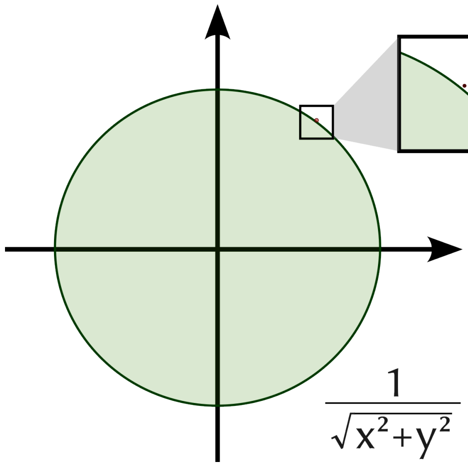

Figure 4: *An example where tighter intervals of CR functions prevent errors: when the inputs to the problem are a few ULPs away from producing pathological situations, CR minimizes error and is often able to avoid failure. Moreover, its results are machine-independent; a non-CR implementation may complete successfully on one machine but fail on another.*

simply get an interval that contains the true value. We compute this function in three ways:

- with TIGHT intervals, and using the function `hypot` to compute the Euclidean distance of P from the origin (which is absent in Filib++);
- with TIGHT intervals, using only operators which are also supported by Filib++ (all algebraic in this case);
- with Filib++ intervals, using the same mode as the rest of the paper.

In the first two cases, the denominator is correctly computed as positive with TIGHT and the operation returns a real interval (albeit larger in the second case, due to multiple operations being involved). When using Filib++, the square root produces an interval with lower bound equal to the radius, resulting in a division by zero.

While this example involves only a few operations, for more complex expressions, the propagation of error can be even more dramatic. Correct rounding does not eliminate the issue, but it reduces propagation to a minimum and makes it predictable, since each operation can introduce at most 1 ULP of error on each side.

5. Conclusions and future work

We have introduced TIGHT, an efficient C++ library for correctly rounded interval arithmetic built on top of the state-of-the art in both correctly rounded computations and interval arithmetic.

TIGHT is designed to be future-proof: if correctly rounded mathematical routines become the standard in the coming years, or more performant CR methodologies are developed, switching the underlying library requires minimal changes. Crucially, since the result of a CR operation is well defined, TIGHT will continue to return the same results *forever* on all machines, even in the event of a library change .

The library can still be improved in several ways:

1. NFG's arithmetic operations owe their speed to vectorization. Vectorized mathematical libraries exist [SP20] but we are not aware of a correctly-rounded solution. In any case, having non-CR, faster interval routines could still be useful for applications that can cope with error and inconsistencies across machines.
2. TIGHT currently does not conform with the IEEE 1788-2015 standard for interval arithmetic [iee15]. Adding compliance and testing with a framework such as [RBFZ22] would make the library more easily integrated. This includes adding conformant support for infinity and NaN values.
3. As mentioned in Section 3.1.6, integrating CR functions for multiplication with an irrational constant value as in [BM05] would bring a more consistent speed for sin and cos on arbitrary arguments.
4. CORE-MATH also contains a CR implementation of the gamma function (included in the C++ standard library), that could be used to build an interval implementation, though it is not clear whether this would be useful in practice.

References

[Att25] Attene M.: Nfg - numbers for geometry, 2025. URL: https://github.com/MarcoAttene/NFG. 2, 3

[BB15] Bailey D., Borwein J.: High-precision arithmetic in mathematical physics. *Mathematics 3*, 2 (2015), 337–367. 1

[BHMZ25] Brisebarre N., Hanrot G., Muller J.-M., Zimmermann P.: Correctly-rounded evaluation of a function: Why, how, and at what cost? *ACM Comput. Surv. 58*, 1 (2025). 3

[BM05] Brisebarre N., Muller J.-M.: Correctly rounded multiplication by arbitrary precision constants. In *17th IEEE Symposium on Computer Arithmetic (ARITH'05)* (2005), pp. 13–20. 5, 8

[BMP06] Brönnimann H., Melquiond S., Pion S.: The design of the boost interval arithmetic library. *Theoretical Computer Science 351* (2006), 111–118. doi:10.1016/j.tcs.2005.09.062. 2

[DLDdM03] Daramy-Loirat C., Defour D., de Dinechin F., Muller J.-M.: CR-LIBM: A correctly rounded elementary function library. In *Proceedings SPIE 48th Annual Meeting - Optical Science and Technology* (San Diego (CA) – USA, 2003). 3

[Gol91] Goldberg D.: What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv. 23*, 1 (1991), 5–48. 1

[Gou16] Goualard F.: Gaol: Not just another interval library, 2016. URL: https://github.com/goualard-f/GAOL. 2

[HJVE01] Hickey T., Ju Q., Van Emden M.: Interval arithmetic: from principles to implementation. *Journal of the ACM 48* (2001), 1038–1068. 2

[iee15] IEEE Computer Society: *IEEE Standard for Interval Arithmetic*. New York, NY, USA, 2015. doi:10.1109/IEEESTD.2015.7107164. 8

[iee19] IEEE Computer Society: *IEEE Standard for Floating-Point Arithmetic*. New York, NY, USA, 2019. doi:10.1109/IEEESTD.2019.8766229. 2

[Knü94] Knüppel O.: PROFIL/BIAS — a fast interval library. *Computing 53*, 3 (1994), 277–287. 2

[Lam08] Lambov B.: Interval arithmetic using sse-2. In *Reliable Implementation of Real Number Algorithms: Theory and Practice* (Berlin, Heidelberg, 2008), Hertling P., Hoffmann C. M., Luther W., Revol N., (Eds.), Springer Berlin Heidelberg, pp. 102–113. 3

[LFS*20] Li M., Ferguson Z., Schneider T., Langlois T., Zorin D., Panozzo D., Jiang C., Kaufman D. M.: Incremental potential contact: Intersection- and inversion-free large deformation dynamics. *ACM Trans. Graph. (SIGGRAPH) 39*, 4 (2020). 7

[LN22] Lim J., Nagarakatte S.: One polynomial approximation to produce correctly rounded results of an elementary function for multiple representations and rounding modes. *Proc. ACM Program. Lang. 6* (2022). 3

[LTG*06] Lerch M., Tischler G., Gudenberg J., Hofschuster W., Kramer W.: Filib++, a fast interval library supporting containment computations. *ACM Trans. Math. Softw 32*, 2 (2006), 299–324. 2

[LTG*11] Lerch M., Tischler G., Gudenberg J., Hofschuster W., Kramer W.: Fi_lib and filib++, 2011. URL: https://www2.math.uni-wuppertal.de/wrswt/software/filib.html. 2

[Pro21] Project T. C.: *CGAL User and Reference Manual*, 5.3 ed. CGAL Editorial Board, 2021. URL: https://doc.cgal.org/5.3/Manual/packages.html. 2

[RBFZ22] Revol N., Benet L., Ferranti L., Zhilin S.: Testing interval arithmetic libraries, including their ieee-1788 compliance. In *LNCS* (Gdansk, Poland, Sept. 2022), vol. 13827, Springer, pp. 428–440. URL: https://inria.hal.science/hal-03674743, doi:10.1007/978-3-031-30445-3_36. 8

[SP20] Shibata N., Petrogalli F.: Sleef: A portable vectorized library of c standard mathematical functions. *IEEE Transactions on Parallel and Distributed Systems 31*, 6 (2020), 1316–1327. doi:10.1109/TPDS.2019.2960333. 8

[SPH*25] SICHETTI F., PUPPO E., HUANG Z., ATTENE M., ZORIN D., PANOZZO D.: MiSo: a DSL for robust and efficient solve and minimize problems. *ACM Trans. Graph. 44*, 4 (July 2025). 1, 2, 6, 7

[SZG22] SIBIDANOV A., ZIMMERMANN P., GLONDU S.: The CORE-MATH project. In *29th IEEE Symposium on Computer Arithmetic (ARITH)* (2022), pp. 26–34. 2, 3

[TFS*22] TANG X., FERGUSON Z., SCHNEIDER T., ZORIN D., KAMIL S., PANOZZO D.: A cross-platform benchmark for interval computation libraries. In *Parallel Processing and Applied Mathematics: 14th International Conference, Revised Selected Papers, Part II* (Berlin, Heidelberg, 2022), Springer-Verlag, p. 415–427. 2, 5