# A Cross-Platform Benchmark for Interval Computation Libraries

Xuan Tang[1], Zachary Ferguson[1], Teseo Schneider[2], Denis Zorin[1], Shoaib Kamil[3], and Daniele Panozzo[1]

[1] New York University
[2] University of Victoria
[3] Adobe Research

**Abstract.** Interval computation is widely used in Computer Aided Design to certify computations that use floating point operations to avoid pitfalls related to rounding error introduced by inaccurate operations. Despite its popularity and practical benefits, support for interval arithmetic is not standardized nor available in mainstream programming languages.

We propose the first benchmark for interval computations, coupled with reference solutions computed with exact arithmetic, and compare popular C and C++ libraries over different architectures, operating systems, and compilers. The benchmark allows identifying limitations in existing implementations, and provides a reliable guide on which library to use on each system for different CAD applications. We believe that our benchmark will be useful for developers of future interval libraries, as a way to test the correctness and performance of their algorithms.

**Keywords:** Interval Arithmetic · Transcendental Functions · Certified Computations · Collision Detection · Robust Computation · Open-Source Library · Benchmark

| | BOOST | FILIB | NATIVE SWITCHED | MULTIPLICATIVE | PRED-SUCC | BIAS |
|---|---|---|---|---|---|---|
| CORRECTNESS (ARITHMETIC) | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| CORRECTNESS (TRANSCENDENTAL) | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ |
| CORRECTNESS (COMPOSITE) | ✗ | ✓ | ✗ | ✓ | ✓ | ✗ |
| INTERVAL WIDTH (ARITHMETIC) | 1 | 2 | 1 | 3 | 2 | 1 |
| INTERVAL WIDTH (TRANSCENDENTAL) | 1 | 2 | 2 | 2 | 2 | 1 |
| SPEED | 6 | 3 | 4 | 1 | 2 | 5 |
| CONSISTENCY | ✗ | ✓ | ✗ | ✓ | ✗ | ✗ |
| PORTABILITY | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ |

**Fig. 1.** We introduce a benchmark for interval arithmetic computation and test it on four C/C++ libraries: filib, filib++ (including the `native_switched`, `multiplicative`, and `pred_succ` methods), Boost, and BIAS. We evaluate each library for their correctness, output interval size, speed, consistency, and portability. The table shows a summary of our benchmark where the numbers indicate a ranking from best (small) to worst (large).

## 1 Introduction

Interval computation allows performing floating-point operations with certifiable correctness, by accounting for rounding errors. Every floating-point number is replaced by a pair of numbers, representing an interval that contains the exact result of the computation, independently from the rounding. While this approach increases the cost and memory usage of computations, it is a staple for many algorithms in computer aided design, geometric computing, image processing, computer graphics, and scientific computing. For example, they are used for Boolean computation [24], intersections between parametric patches [23], continuous collision detection [20], subdivision surfaces [28], and precision manufacturing [25]. More applications are discussed in the survey [15].

While the formal correctness of interval computation has been proven [22], ensuring that an implementation of interval arithmetic is correct is a daunting task, as the proof relies on assumptions on the order of operations (which can be altered by the compiler or the reordering buffers on the CPU) and on a set of hardware assumptions on the ALUs, which are architecture-dependent. At the same time, users rely on interval computation to certify the correctness of their algorithm, assuming that the interval computation library is correct, which, as we will show in this paper, is not always true for specific combinations of compilers, operating systems, and architectures.

Because a formal proof for every hardware and software combination is impractical (requiring to adapt the proof at every new software or hardware update), we propose an experimental approach: we introduce a large benchmark of test expressions and real-world algorithms for which the exact answer is computed using exact computation. The benchmark can then be used to test existing implementations, and identify issues. We note that a library passing our benchmark problems might still contain errors, as our benchmarks do not exhaustively test all possible combinations of operations and operands.

We use our benchmark to evaluate four popular C/C++ interval libraries (filib, filib++, Boost, BIAS) for correctness, interval size, speed, and consistency. The results are summarized in Table 1. The only library that is at the same time correct, consistent, portable, and has a reasonable speed is filib, which does not rely on using special hardware instructions to control the underlying rounding mode.

We provide the complete source code and scripts to run our benchmark, and in addition we provide a CMake build system for using filib on Windows, Linux, and macOS operating systems with both x86 and ARM architectures. We believe that our benchmark will be a useful tool to continue to assess the correctness of existing interval libraries as new compilers and architectures are developed, and also to provide a standardized set of tests for developers of interval libraries.

## 2 Background

In the past two decades, numerous interval arithmetic libraries have been developed in various languages. While the logic behind interval arithmetic has been explored in many works [12,3], the actual implementations vary from library to library and may produce different results.

### 2.1 Hardware Rounding Mode Control

Many modern programming languages comply with the IEEE 754 standard for implementing floating point datatypes, which supports different rounding rules (round to nearest, towards $\infty$, and towards $-\infty$) [14]. These rounding modes provide good lower and upper bounds on basic arithmetic operations. Libraries like Boost [21] or CGAL [24] use this functionality to build interval operations. Their implementation focuses on setting the correct rounding mode before calling the default math library [4]. Other libraries like Profil/BIAS [16] and filib++ [18] also use or include this implementation for basic algebraic operations.

Such strategies work well for basic arithmetic operations but require a lot of care when computing transcendental functions, where many rounding changes need to happen to evaluate a single transcendental function [10]. Some of them, like CGAL, sidestep the problem by not supporting transcendental operations.

### 2.2 Software Implementations

It is possible to avoid relying on hardware rounding mode support by using a pure software implementation. There are two main approaches.

*Multiplicative.* While it is hard to obtain the exact floating point error of an expression, the relative error of single operations can be generalized [17,11] since there are only a finite number of bits representing a number [7]. Hence, one can carefully analyze the error to generate a number $\epsilon$ such that if the true result is $\alpha$ and the computed result is $\beta$, $(1-\epsilon)\beta \leq \alpha \leq (1+\epsilon)\beta$ holds and $1-\epsilon$, $1+\epsilon$ can be exactly represented in floating-point. Filib++, BIAS, and GAOL [8] all provide such implementations, although the choice of $\epsilon$ varies depending on how the analysis is performed.

*Changing binary representation.* Since nowadays almost all floating-point numbers implementations follow the IEEE 754 standard, one can deconstruct the binary representation of a number and directly change the result to obtain an interval [1]. Filib and filib++ adopt this approach, by directly modifying the mantissa and exponent of a double, generating a reasonably-small interval without sacrificing performance.

### 2.3 Other Implementations

Some libraries rely on others as part of their implementation of interval arithmetic. For example, IBEX [5] and XSC [13] both use filib as the backend for interval computation. These libraries generally do not provide better performance or smaller interval width, but they focus on providing a more user-friendly interface. Other interval libraries exist in other programming languages. For example, IntervalArithmetic.jl [2] in Julia, interval-arithmetic [19] in Javascript. Since our goal is on C/C++ libraries we do not include such libraries for our study.

## 3 Methodology

A good interval library should maintain four traits: (1) correctness, (2) small interval widths, (3) efficiency, and (4) consistency across different architectures and compilers. We design our benchmark to test these four traits. We recognize that in many applications, an interval itself is initialized from a single number rather than an actual range since the goal is to compute an interval that includes the true value of an expression. Hence, the initialization of an interval in our benchmark is always from a single value. In our benchmark, we compare the following four popular open source libraries that complies with IEEE 754 standard: filib, filib++, Boost, and BIAS.

Filib++ supports three modes for interval computations: `native_switched` (uses system rounding modes), `pred_succ` (directly manipulates the bit representation of a double), and `multiplicative` (multiplies two numbers to generate an interval). BIAS includes three rounding modes (ROUND DOWN, ROUND UP, and ROUND NEAR) which can be set before an interval operation. Their documentation is unclear how an interval operation is affected by these rounding modes, thus we treat them as three different interval types.

### 3.1 Expressions

We list the expressions that will later be referred to in this paper here:

$$\frac{a(a+bc)}{(b+cd)} - \frac{d\,(e+f/g)}{(g+h)} - \frac{i}{j} \tag{1}$$

$$\cos\left(\left(\cos\left(\cos(f)+\exp\left(\frac{d}{c}\right)\right)\right)\left(\sin\left(\sqrt{e}+a+b-\sqrt{d+c}\right)\right)\right) \tag{2}$$

$$\exp\left(\sqrt{\exp\left(\sqrt{\exp\left(\sqrt{a}\right)}\right)}\right) \tag{3}$$

$$\exp\left(\frac{\sqrt{\exp\left(\cos\left(a/d\right)\right)/\exp\left(\cos\left(\sqrt{f}\right)\right)}}{\sqrt{\cos(\cos(\cos(c)))/\sqrt{\sin(\cos(b))}}}\right) \tag{4}$$

### 3.2 Correctness

While libraries can optimize interval operations for every single arithmetic or transcendental function, composite expressions that combine multiple operations can potentially cause the library to produce incorrect (interval does not include the true result) or empty (lower bound is greater than upper bound) intervals. In our benchmark, we test each library on 28 different basic expressions and 104 expressions from FPBench [6], a floating point accuracy benchmark that covers a variety of application domains. The basic benchmark is composed of: four basic arithmetic operations (addition, subtraction, multiplication, and division); four transcendental functions (sqrt, exp, sin, cos); ten composite expressions that only contain basic arithmetic operations; and ten composite expressions containing both arithmetic operations and transcendental functions. These expressions are randomly generated from a fixed seed and listed on the website.

For each expression, we generate one million valid inputs for evaluation. To ensure that the representation of the input and result are precise, and no additional floating point error is introduced during validation, we convert every input and output to rational format using GMP [9]. A typical query has the form

$$\frac{n_l}{d_l} \leq \text{expression}(\frac{n_1}{d_1}, \ldots) \leq \frac{n_u}{d_u}$$

for $n, d \in \mathbb{Z}$. Using this format, the queries can be evaluated later by an arbitrary precision software to get an exact answer. In our benchmark, we use Mathematica [27].

## 3.3 Interval width

To report the interval width we utilize a similar procedure to when checking correctness. Instead of outputting the actual query, we compute the interval width by using a rational subtraction (i.e., we convert the upper and lower bound to rational numbers).

## 3.4 Speed

To test the speed of an interval library, we measure the execution time for each expression: we generate 1,000 inputs for each expression and execute the expression 10,000 times for every input. Finally, we accumulate the total execution time for each library and expression to report the performance. It is important to execute different sets of inputs since input values may affect the performance of some operations due to range reduction.

## 3.5 Consistency and Portability

We deployed our benchmark on four different platforms with different compilers:

- Windows (Intel Core i7 8700k, x86-64, Windows 10, MSVC 14.27.29110)
- macOS Intel (2.4 GHz 8-Core Intel Core i9, x86-64, macOS Big Sur, Darwin Kernel Version 20.1.0, Apple clang version 12.0.0)
- macOS Arm (3.2 GHz 4-Core/2 + 2 GHz 4-Core Apple M1, arm64, macOS Big Sur, Darwin Kernel Version 20.1.0, Apple clang version 12.0.0)
- Linux (AMD EPYC 7452 32-Core Processor, x86-64, Ubuntu 19.10, GCC 9.2.1).

## 4 Results

We discuss in detail how each interval type perform over different platform and expressions.

### 4.1 Correctness

As discussed before, we test each library on 28 (constructed by us) and 104 (extracted from FPBench) expressions. We check for correctness by ensuring that the interval computation produces an interval containing the exact solution, evaluated with arbitrary precision with Mathematica [27].

We begin with the 28 expressions. All of the libraries produce correct results for basic arithmetic operations. However, when it comes to transcendental functions, Boost is not correct (since it deals with transcendental functions by setting rounding modes before calling the standard math library). Specifically, it fails for exp and trigonometry functions, where the implementation is based on Taylor expansion [10]. For composite expressions that only contain basic arithmetic operations, all libraries are correct. When transcendental functions are included in a composite expression, BIAS produces incorrect intervals for Expression (2).

Filib and filib++'s three interval modes are correct for the 28 expressions, the `native_switched` mode for filib++ is not correct on four of the expressions from FPBench. For example, "polarToCarthesian, x", that computes

$$r\cos(\theta \cdot (3.14159265359/180.0))$$

which contains transcendental function cos. Another example is the expression "sineOrder3"

$$(0.954929658551372x_0) - (0.12900613773279798((x_0\, x_0)\, x_0))$$

which only contains basic arithmetic operations, and is designed to find floating point problem caused by the order of evaluation.

We conclude that only filib and filib++'s `pred_succ` and `multiplicative` modes produce correct intervals for all tests.

### 4.2 interval width

Due to the large number of test expressions, we show only some of the most representative expressions. Specifically, we look at one expression that contains only arithmetic operations (Expression (1)), one expression that contains only transcendental functions (Expression (3)), and one that contains both (Expression (4)).

Across the different platforms, the distribution of interval width does not vary much. However, within each platform, the distribution of interval width can be quite different between libraries. The top row of Figure 2 shows that for expressions that contain only arithmetic operations, libraries that use system rounding

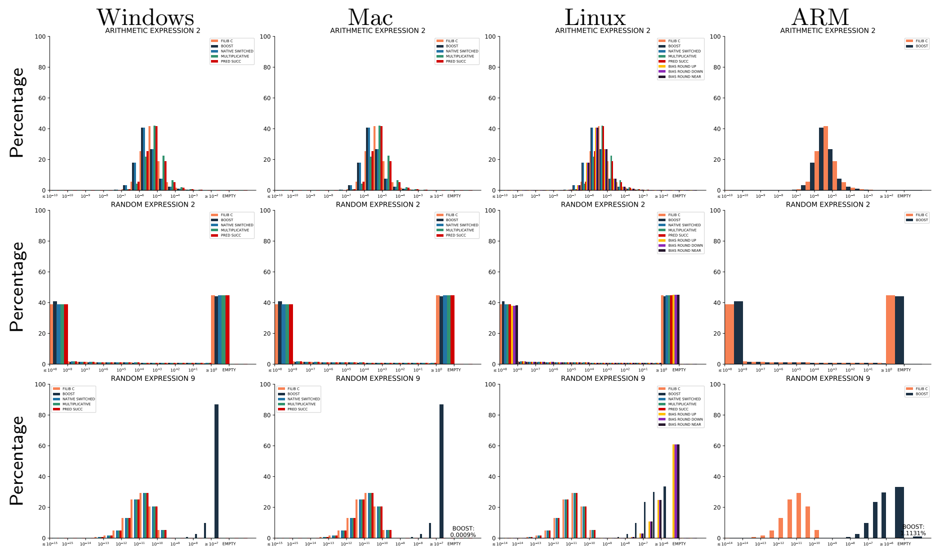

**Fig. 2.** Distribution of interval width. Top: Expression (1). Middle: Expression (3). Bottom: Expression (4)

modes (Boost, filib++ `native_switched`, BIAS) produce smaller interval widths compared to others. The `multiplicative` mode of filib++ produces the largest interval widths. However, the differences are small across libraries.

When transcendental functions are added into the expression, the interval widths can be unpredictable (Figure 2). The difference of overall distribution of the libraries can also be quite large depending on the expression, but within filib and filib++'s three interval modes, the interval widths are quite similar. We also see that Boost produces empty intervals, an indication that Boost's results are sometimes incorrect.

## 4.3 Performance

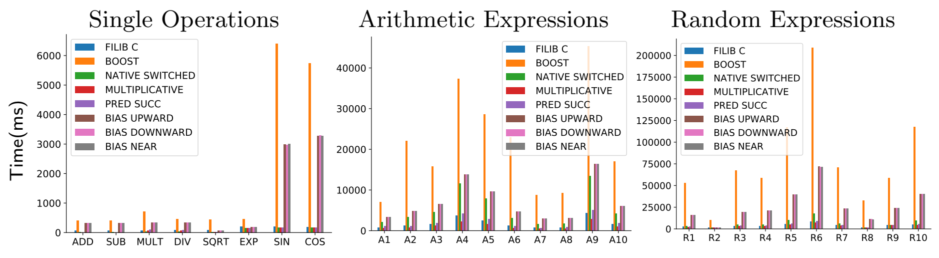

**Fig. 3.** Time for each expression (1,000 × 10,000 runs) in ms on Linux.

| | FILIB C | BOOST | NATIVE SWITCHED | MULTIPLICATIVE | PRED SUCC | BIAS UPWARD | BIAS DOWNWARD | BIAS NEAR |
|---|---|---|---|---|---|---|---|---|
| ADDITION | 66.52 | 405.78 | 18.49 | 3.94 | **3.92** | 327.93 | 328.69 | 327.94 |
| SUBTRACTION | 80.08 | 405.35 | 18.37 | 9.20 | **3.90** | 327.60 | 327.44 | 327.44 |
| MULTIPLICATION | 78.40 | 649.62 | 25.81 | **13.59** | 90.03 | 339.43 | 338.87 | 338.61 |
| DIVISION | 84.43 | 451.78 | **38.13** | 68.97 | 81.12 | 344.25 | 343.87 | 344.13 |
| SQUARE ROOT | 84.85 | 434.21 | 24.19 | 24.26 | **24.18** | 61.76 | 60.44 | 60.40 |
| EXPONENTIAL | 199.08 | 462.47 | **143.28** | 143.43 | 143.43 | 196.44 | 196.34 | 200.39 |
| SIN | 202.25 | 7115.77 | **171.54** | 172.79 | 172.94 | 2088.65 | 2088.34 | 2087.43 |
| COS | 190.21 | 6776.14 | 168.64 | 168.65 | **168.54** | 2425.19 | 2424.21 | 2423.90 |
| ARITHMETIC EXPRESSION 1 | 861.49 | 6848.71 | 1969.52 | **589.98** | 1053.79 | 4126.49 | 4097.03 | 4091.17 |
| ARITHMETIC EXPRESSION 2 | 1292.77 | 23227.45 | 3170.62 | **699.40** | 1040.81 | 5706.45 | 5672.61 | 5661.78 |
| ARITHMETIC EXPRESSION 3 | 1844.20 | 15082.05 | 4984.85 | **1318.50** | 1900.36 | 7790.34 | 7769.46 | 7757.53 |
| ARITHMETIC EXPRESSION 4 | 3758.37 | 30943.47 | 11407.83 | **2354.18** | 4077.55 | 16086.02 | 16062.78 | 16054.29 |
| ARITHMETIC EXPRESSION 5 | 2635.54 | 21860.23 | 7910.95 | **1626.65** | 2888.93 | 10891.99 | 10883.80 | 10871.13 |
| RANDOM EXPRESSION 1 | 2449.01 | 55822.93 | 3454.41 | **2112.71** | 2350.52 | 12849.99 | 12853.38 | 12858.24 |
| RANDOM EXPRESSION 2 | 1354.32 | 4737.32 | 1465.94 | 1441.29 | 1440.93 | **1323.64** | 1327.30 | 1324.32 |
| RANDOM EXPRESSION 3 | 3382.98 | 69212.48 | 5741.97 | **2945.31** | 3689.13 | 17524.99 | 17485.55 | 17498.33 |
| RANDOM EXPRESSION 4 | 3067.34 | 56868.36 | 4927.13 | **3063.33** | 3593.86 | 17996.21 | 17975.26 | 17973.13 |
| RANDOM EXPRESSION 5 | 5870.61 | 121944.99 | 9633.84 | **5146.98** | 6081.06 | 32546.81 | 32577.18 | 32540.89 |

**Table 1.** Time for each expression (1,000 × 10,000 runs) in ms on Linux. The relative timings are similar on different platforms and OS. The complete results can be found on our github page.

We show the accumulated time in milliseconds for each expression on the Linux platform since the relative performance across platforms is similar. We also highlight the fastest method for each expression. From Table 1, we see that Boost has the worst performance on all of the 28 expressions, followed by BIAS, then filib. While filib++'s `native_switched` mode also sets rounding mode for basic arithmetic operations, it is highly optimized and is significantly faster than the other two libraries.

Within filib++, the performance of the three modes on basic operations is comparable. However, for more complex arithmetic expressions, `native_switched` mode is consistently the slowest, likely because it changes the rounding mode for each operations. The multiplicative mode is always the fastest, while `pred_succ` method is between the two. Since filib++ ignores interval mode when computing transcendental functions, the performance on sqrt, exp, sin, cos are similar. As a result, when computing more complicated expressions, multiplicative mode remains the fastest one among three modes and among all the interval types, followed by `pred_succ` mode, then `native_switched` mode. Although the speed of filib++ can drop below filib or even BIAS on some expressions, the relative difference is minimal.

## 4.4 Consistency and Portability

While Boost can be deployed on all platforms we test on, it does not produce consistent results due to its system specific rounding modes.

While filib++ does well in terms of both correctness and speed, it does not produce the same result across different platforms: we found that it produces different results on the Linux platform. As seen in Figure 4, the `pred_succ`'s distribution of interval width differs from Linux to Mac. Additionally, its porta-

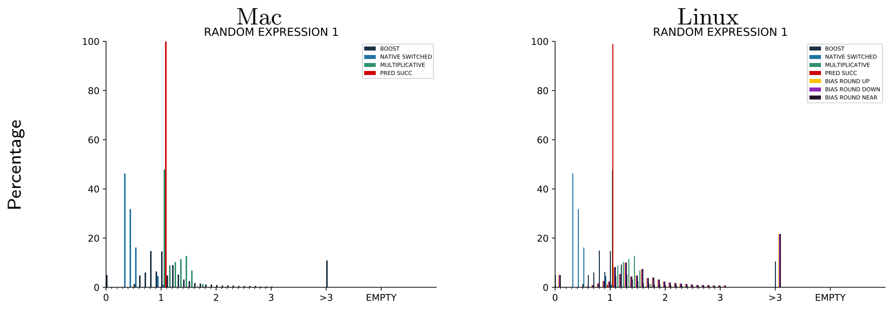

**Fig. 4.** Distribution of each library's interval width on expression 2, normalized.

bility is limited due to the lack of updates since 2011 and the use of autoconf to generate the makefile. [4]

BIAS is not maintained [5] and currently does not compile out of the box on modern windows and macOS versions. We thus only tested it on Linux.

## 4.5 Application on Continuous Collision Detection Queries

As a further benchmark of correctness, we integrate three interval libraries in the continuous collision detection (CCD) benchmark of [26]. The CCD benchmark features two interval based algorithms to detect collisions along a continuous linear trajectory. Both the univariate and multivariate interval-based CCD perform interval-based bisection root finding [22] and use interval arithmetic to compute an estimate of the codomain of a function. The correctness of the interval arithmetic ensures that no false negatives (no collision is reported when there is a collision) occur and smaller interval width helps to reduce the number of false positives (a collision is reported when there is no collision).

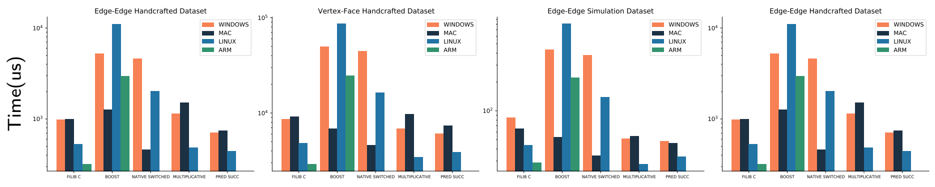

**Fig. 5.** Average time of each query using different interval types on different platform in univariate interval root finder test.

[4] filib++ source: http://www2.math.uni-wuppertal.de/wrswt/software/filib.html, last updated in 2011.

[5] BIAS source: https://www.tuhh.de/ti3/keil/profil/index_e.html, last updated in 2009,

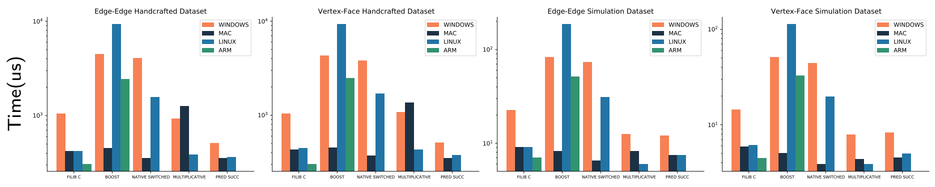

**Fig. 6.** Average time of each query using different interval types on different platform in multivariate interval root finder test.

Timing-wise (Figure 5, 6), all libraries are in a similar ballpark, with the exception of Boost being slightly slower than the others in certain tests.

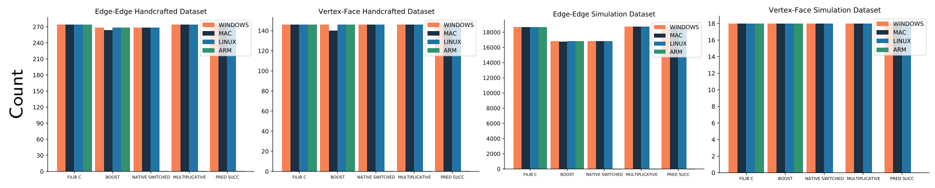

**Fig. 7.** Number of false positives using different interval types on different platform in univariate interval root finder test.

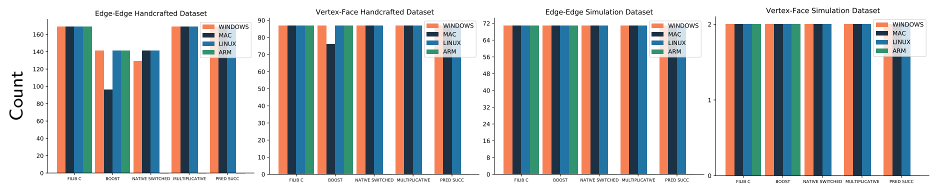

**Fig. 8.** Number of false positives using different interval types on different platform in multivariate interval root finder test.

None of the libraries produces false negatives in this benchmark. The number of false positives varies as expected, as the intervals are different (Figure 7, 8). It is concerning to see that Boost and filib++'s `native_switched` produce different numbers of false positives on different architectures. filib, filib++'s `pred_succ`, and filib++'s `multiplicative` method produce consistent results across all operating systems and architectures.

## 5 Conclusion

In this paper, we designed a benchmark that tests interval libraries for correctness, interval width, speed, and consistency. Using our benchmark we evaluated four interval libraries: filib, filib++, Boost, and BIAS (Table 1). We also provide the complete results along with all the expressions on our github page. [6]

In our study, filib is the only library that is correct, consistent, portable, and efficient. We believe it is the best option between the libraries we tested. To make deployment on multiple platforms easier, we provide a copy of the library with a modern cmake build system on github. [7]

## References

1. Abrams, S., Cho, W., Hu, C.Y., Maekawa, T., Patrikalakis, N., Sherbrooke, E., Ye, X.: Efficient and reliable methods for rounded-interval arithmetic. Computer-Aided Design **30**(8), 657 – 665 (1998). https://doi.org/https://doi.org/10.1016/S0010-4485(97)00086-9, `http://www.sciencedirect.com/science/article/pii/S0010448597000869`
2. Benet, L., Sanders, D.: Juliaintervals.jl package — Rigorous numerics with interval arithmetic & applications (2015), `https://github.com/JuliaIntervals/IntervalArithmetic.jl`
3. Benhamou, F., Older, W.J.: Applying interval arithmetic to real, integer, and boolean constraints. The Journal of Logic Programming **32**(1), 1 – 24 (1997). https://doi.org/https://doi.org/10.1016/S0743-1066(96)00142-2, `http://www.sciencedirect.com/science/article/pii/S0743106696001422`
4. Brönnimann, H., Melquiond, G., Pion, S.: The design of the Boost interval arithmetic library. Theoretical Computer Science **351**(1), 111 – 118 (2006). https://doi.org/https://doi.org/10.1016/j.tcs.2005.09.062, `http://www.sciencedirect.com/science/article/pii/S0304397505006110`, real Numbers and Computers
5. Chabert, G.: IBEX (2007), `http://www.ibex-lib.org/`
6. Damouche, N., Martel, M., Panchekha, P., Qiu, J., Sanchez-Stern, A., Tatlock, Z.: Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. Numerical Software Verification (July 2016)
7. Goldberg, D.: What Every Computer Scientist Should Know about Floating-Point Arithmetic. ACM Comput. Surv. **23**(1), 5–48 (Mar 1991). https://doi.org/10.1145/103162.103163, `https://doi.org/10.1145/103162.103163`
8. Goualard, F.: Gaol: NOT Just Another Interval Library (2005), `https://sourceforge.net/projects/gaol/`
9. Granlund, T., Team, G.D.: GNU MP 6.0 Multiple Precision Arithmetic Library. Samurai Media Limited, London, GBR (2015)
10. Harrison, J., Tak, P., Tang, P.: The Computation of Transcendental Functions on the IA-64 Architecture. In: Intel Technology Journal. vol. 4, pp. 234–251 (1999)
11. Harrison, J.: Formal Verification of Floating Point Trigonometric Functions. In: Hunt, W.A., Johnson, S.D. (eds.) Formal Methods in Computer-Aided Design. pp. 254–270. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
12. Hickey, T., Ju, Q., Van Emden, M.H.: Interval Arithmetic: From Principles to Implementation. J. ACM **48**(5), 1038–1068 (Sep 2001). https://doi.org/10.1145/502102.502106, `https://doi.org/10.1145/502102.502106`

[6] https://geometryprocessing.github.io/intervals/
[7] https://github.com/txstc55/filib.

13. Hofschuster, W., Krämer, W.: C-XSC 2.0 – A C++ Library for Extended Scientific Computing. In: Alt, R., Frommer, A., Kearfott, R.B., Luther, W. (eds.) Numerical Software with Result Verification. pp. 15–35. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
14. IEEE: Ieee standard for binary floating-point arithmetic. ANSI/IEEE Std 754-1985 pp. 1–20 (1985). https://doi.org/10.1109/IEEESTD.1985.82928
15. Kearfott, R.: Interval Computations: Introduction, Uses, and Resources. Euromath Bulletin **2** (01 1996)
16. Knüppel, O.: PROFIL/BIAS—A fast interval library. Computing **53**(3), 277–287 (Sep 1994). https://doi.org/10.1007/BF02307379, https://doi.org/10.1007/BF02307379
17. Lefevre, V., Muller, J..: Worst cases for correct rounding of the elementary functions in double precision. In: Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001. pp. 111–118 (2001). https://doi.org/10.1109/ARITH.2001.930110
18. Lerch, M., Tischler, G., Gudenberg, J.W.V., Hofschuster, W., Krämer, W.: FILIB++, a Fast Interval Library Supporting Containment Computations. ACM Trans. Math. Softw. **32**(2), 299–324 (Jun 2006). https://doi.org/10.1145/1141885.1141893, https://doi.org/10.1145/1141885.1141893
19. Poppe, M.: interval-arithmetic (2015), https://github.com/mauriciopoppe/interval-arithmetic
20. Redon, S., Kheddar, A., Coquillart, S.: Fast Continuous Collision Detection between Rigid Bodies. Computer Graphics Forum **21** (May 2002)
21. Schling, B.: The Boost C++ Libraries. XML Press (2011)
22. Snyder, J.: Interval Analysis For Computer Graphics. In: ACM SIGGRAPH. pp. 121–130. ACM (August 1992), https://www.microsoft.com/en-us/research/publication/interval-analysis-computer-graphics/
23. Snyder, J.M., Woodbury, A.R., Fleischer, K., Currin, B., Barr, A.H.: Interval Methods for multi-point collisions between time-dependent curved surfaces. In: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques. p. 321–334. SIGGRAPH '93, Association for Computing Machinery, New York, NY, USA (1993)
24. The CGAL Project: CGAL User and Reference Manual. CGAL Editorial Board, 5.3 edn. (2021), https://doc.cgal.org/5.3/Manual/packages.html
25. Tibken, B., Hofer, E.P., Seibold, W.: Quality control of valve push rods using interval arithmetic. IFAC Proceedings Volumes **32**(2), 409–412 (1999), 14th IFAC World Congress 1999, Beijing, Chia, 5-9 July
26. Wang, B., Ferguson, Z., Schneider, T., Jiang, X., Attene, M., Panozzo, D.: A Large Scale Benchmark and an Inclusion-Based Algorithm for Continuous Collision Detection. ACM Transactions on Graphics **40**(5) (Oct 2021)
27. Wolfram Research Inc.: Mathematica 12.0 (2020), http://www.wolfram.com
28. Zorin, D.: A Method for Analysis of C1-continuity of Subdivision Surfaces. SIAM Journal on Numerical Analysis **37**(5), 1677–1708 (Jan 2000). https://doi.org/10.1137/s003614299834263x, https://doi.org/10.1137/s003614299834263x