



UNIVERSITÀ DEGLI STUDI DI GENOVA

Department of Informatics, Bioengineering, Robotics
and Systems Engineering (DIBRIS)

Master of Science in Computer Science

Degree course 10852

An IEEE 1788-2015 Compliant Library for Correctly Rounded Interval Computations

Bringing the TIGHT interval library into conformance,
and measuring what conformance costs

Candidate:

Gabriele Francesco Berruti

Supervisor:

Prof. Enrico Puppo
Università degli Studi di Genova

Co-supervisor:

Prof. Daniele D'Agostino
Università degli Studi di Genova

Academic Year 2025/2026

In collaboration with:



Federico Sichetti
New York University



Marco Attene
CNR-IMATI, Genova

Abstract

This thesis asks whether an interval library written for speed can be made conformant to the interval arithmetic standard IEEE 1788-2015 without giving up what makes it useful, and what conformance costs. Interval arithmetic computes with sets of real numbers instead of a single approximation, and returns a result guaranteed to contain the exact answer. That guarantee is what a geometric predicate needs: it decides a sign rather than estimating one. It holds only where the rules are fixed, and they come from two standards: IEEE 1788-2015 for the intervals, IEEE 754-2019 for the numbers their endpoints are made of. TIGHT is that library: the numeric core of a system of geometric predicates, where speed decides whether a computation is feasible at all. Its arithmetic runs through a branch-free kernel of its own, NFG. It was built for that purpose, not for the standard.

The obligations of the two standards meet at a single point: the optional claim of 754-conformance, which carries over accuracy of values and not exception flags. Conformance can be obtained almost entirely in the wrapper above the kernel, but not entirely. A wrapper repairs only what it recognises from the operands before calling, and one defect of the kernel gives a wrong value on two finite intervals.

Seventy-one atomic changes close the thirteen blocks of clauses that make up the roadmap agreed with the supervisor, each carrying one clause, one commit and one test block. Correctness is decided against an external corpus written for other libraries, grown with the work from 4358 to 9433 assertions. Twenty-six lines still fail, twelve of them because the answer given here is *tighter* than the corpus expects and still encloses the exact result: the corpus compares by equality where the standard asks only for containment. Coverage is established by reading the 192 obligations of the standard one at a time. Nine defects of the kernel against IEEE 754-2019 were analysed, and four repairs are in its official repository.

The cost was measured on a benchmark with a calibrated margin. Over the 332 cells in which both builds answer the same on every operand, the median surcharge follows the cost of the operation: fifteen per cent on those below two nanoseconds, two per cent on those above a hundred. The added work is fixed, so it weighs in inverse proportion to that cost.

Besides the library, the work leaves behind the instrument that judged it: an external corpus connected by a plugin, a benchmark that states its own margin, and a registry of the changes with their evidence. Conformance is selected by a compilation flag, so the behaviour the predicates were built against survives beside the conformant one.

Acknowledgements

This thesis was written at DIBRIS, the Department of Informatics, Bioengineering, Robotics and Systems Engineering of the University of Genoa. I thank my supervisor, Prof. Enrico Puppo, for the latitude he gave me in deciding how far to take the work, and for asking from the start what the standard requires rather than what the code already did. I thank Prof. Daniele D’Agostino for agreeing to co-supervise it.

The library studied here is not mine, and the two people who wrote it gave their time to a student who turned up proposing changes to it. I thank Federico Sichetti, Assistant Professor and Faculty Fellow at New York University, who wrote TIGHT, opened its repository and read this work in detail, and Marco Attene, of CNR-IMATI in Genoa, who wrote NFG and took four of the corrections proposed here into it. What is right in the pages that follow owes much to their answers; what is not is mine.

Finally, I thank my family and my friends, who were there for the whole of this path.

Gabriele Francesco Berruti
Genoa, October 2026

Declaration on the Use of AI Tools

Claude, developed by Anthropic, was used through the Claude Code interface as a support tool during the preparation of this thesis. It supported four tasks: cross-checking clauses of the two standards against the source documents, drafting and revising the English text, writing the scripts that produce and aggregate the measurements, and proposing candidate changes to the source code. No other part of this work was produced with its assistance.

The choices that determine the content were made by the author: which clause governs each case, how each modification is designed, the experimental protocol, and the interpretation of the results. Every part produced with the tool was reviewed and verified against evidence. Each modification to the code carries the clause it closes, the commit that introduces it and a test block that exercises it, and every figure reported here comes from a measurement that was executed and recorded.

No personal data was submitted to the tool. The material given to it consists of the two standards, the public repositories of the two libraries and the documents written for this work, all cited in the bibliography.

Contents

Abstract	i
Acknowledgements	iii
Declaration on the Use of AI Tools	iv
List of Figures	viii
List of Tables	ix
List of Algorithms	xi
Notation and Acronyms	xii
Chapter 1 Introduction and Motivation	1
1.1 Robust geometric computation and the cost of exactness	1
1.2 Problem statement	2
1.3 Goals and research questions	3
1.4 Contributions	4
1.5 Structure of the thesis	5
I Context	7
Chapter 2 Background	8
2.1 Floating-point arithmetic and IEEE 754-2019	8
2.1.1 Formats and the binary64 representation	8
2.1.2 Rounding-direction attributes	11
2.1.3 Exceptions and status flags	12
2.1.4 The fused multiply-add	12
2.2 Interval arithmetic	13
2.2.1 Intervals, containment and the Fundamental Theorem	13
2.2.2 Dependency and tightness	15
2.3 IEEE 1788-2015	15
2.3.1 The four levels of specification	16
2.3.2 Flavors, and the set-based flavor	16
2.3.3 The required operations	17
2.3.4 Decorations	17

2.3.5	Accuracy: valid, accurate, tightest	18
2.3.6	Exceptions and the conformance claim	19
2.4	How the two standards relate	19
Chapter 3	Related Work	21
3.1	Interval arithmetic libraries	21
3.2	Testing conformance to IEEE 1788-2015	22
3.3	Correctly rounded elementary functions	24
3.4	The starting point: NFG and TIGHT	24
3.4.1	NFG: the numerical kernel	25
3.4.2	TIGHT: the wrapper	28
3.4.3	The versions this work starts from	29
II	Contribution	31
Chapter 4	Design	32
4.1	Requirements	32
4.1.1	What has to be shown	32
4.1.2	Why conformance sits behind a compilation flag	33
4.1.3	Three classes of cost	35
4.2	External verification of conformance	35
4.2.1	The corpus and the plugin	35
4.2.2	What the corpus covers, and what it does not	38
4.3	The benchmark platform	39
4.3.1	Three axes of comparison	39
4.3.2	The measurement protocol	40
4.3.3	The margin of uncertainty	43
4.4	The change registry	44
4.5	Reproducibility	46
Chapter 5	The Kernel, and the Semantics of Evaluation	48
5.1	Where a defect belongs: kernel or wrapper	48
5.2	The defects of NFG against IEEE 754-2019	50
5.3	What was accepted upstream	53
5.4	TIGHT against IEEE 1788-2015: the thirteen blocks	54
5.5	The empty interval and the constants	56
5.6	The semantics of evaluation	56
5.6.1	Empty, the pairings and the domain filter	58
5.6.2	The powers and the exponentials	65
5.6.3	The constructor, the predicates, and the rest of Table 9.1	70
Chapter 6	Constructors, Operations and Reverse Mode	77
6.1	Constructors and text literals	77
6.2	Boolean, set and cancellative operations	83
6.3	Reverse-mode operations	90
Chapter 7	Exceptions, Decorations and the Conformance Claim	98

7.1	Exceptions and decorations	98
7.2	The four reductions, and why they must be exact	107
7.3	The conformance claim	114
7.4	Tightness of the reverse operations	118
7.5	The exhaustive audit of the obligations	123
7.6	What remains open	125
Chapter 8	Experiments	127
8.1	Conformance results	127
8.1.1	The kernel, commit by commit	127
8.1.2	The corpus, block by block	128
8.1.3	What the residue is	129
8.2	The cost of conformance	131
8.2.1	Results by block	131
8.2.2	Where conformance is not a cost	134
8.2.3	The cost of the decoration	135
8.3	Threats to validity	136
Chapter 9	Conclusions and Future Work	138
9.1	Answers to the research questions	138
9.2	What the work leaves behind	139
9.3	Limitations	140
9.4	Future work	141
Appendix A	Summary of the Change Registry	143
Appendix B	Reproducing the Measurements	147
B.1	Building the two compilations	147
B.2	Running the conformance corpus	148
B.3	Relaunching a benchmark	149
Appendix C	The Probe on the Kernel	150
C.1	What the probe is	150
C.2	Building it on the three paths	150
C.3	The rules the probe obeys	151
C.4	What a complete run reports	152
Appendix D	The Operation Map	153
D.1	What the table does not prove	153
D.2	Three names that still mislead	154
Bibliography		158

List of Figures

2.1	The binary64 word	9
4.1	Where the conformance flag acts	34
4.2	The external verification chain	37
4.3	Calibration of the margin of uncertainty	43
4.4	The anatomy of a registry entry	46
5.1	The representation of NFG, and why it is fast	49
5.2	The external corpus along the work	55
6.1	The bisection used by the reverse operations	92
9.1	The direction that remains open	141

List of Tables

2.1	Parameters of the binary64 format	9
2.2	The classes of a binary64 word	10
2.3	The five rounding-direction attributes	11
3.1	The interval arithmetic libraries	21
3.2	The 261 test cases of the corpus, by provenance	23
3.3	How <code>interval_number</code> stores ten notable values	27
3.4	The three compilation paths of the kernel	27
3.5	The public interface of TIGHT at the starting commit	29
4.1	The four stages of the conformant dispatch	34
4.2	The three classes of cost	36
4.3	The corpus, at the first run and today	38
4.4	The four defects the corpus could not see	39
4.5	The three axes of the benchmark	40
4.6	The operand shapes of the benchmark tables	41
4.7	The fourteen blocks of the roadmap and their prerequisites	45
5.1	The nine defects of the kernel	50
5.2	The thirteen blocks of the roadmap	54
5.3	The empty interval before and after M.2.1	59
5.4	The measurement behind M.2.1	60
5.5	The product with unbounded endpoints, before and after M.2.2	60
5.6	Division with a divisor that touches zero	61
5.7	The double rounding of the first conformant division	62
5.8	<code>Empty</code> outside the four operators, before and after M.2.5	63
5.9	The elementary functions on <code>Empty</code> , before and after M.2.13	67
5.10	The exponentials below their minimum, before and after M.2.14	68
5.11	The midpoint under a global rounding mode	71
5.12	Equality before and after M.2.22	72
5.13	The properties of the thirteen unexercised functions	73
5.14	<code>fma</code> against a multiplication and an addition	74
5.15	The two ways of rounding a tie	75
6.1	The four constructions that must fail	79
6.2	The three faults of the constructor from a string	80
6.3	The three unary predicates	84

6.4	The three set relations	86
6.5	The intersection on the cases of its block	87
6.6	The convex hull, where the empty interval is not absorbing	88
6.7	The two outputs of <code>mulRevToPair</code>	88
6.8	The cancellative operations on the cases of their block	89
6.9	The twelve reverse operations	91
6.10	The three inverses of even functions	93
6.11	The periodic inverses, and where they stop	94
6.12	The two inverses of <code>atan2</code>	96
7.1	The five decorations	99
7.2	The register, step by step	101
7.3	The constructors signal, and only where they fail	102
7.4	The decorations, one rule at a time	104
7.5	What a decoration costs	105
7.6	The decorated reverse operations, and the one that is not one of them . .	106
7.7	<code>atan2</code> decorated: touching the cut is not crossing it	107
7.8	The decorated constructor from numbers	110
7.9	The decorated literal, one row per decision	111
7.10	The sign of a zero, and where the rule does not apply	113
7.11	The 754 flags of the reductions	114
7.12	The names of the standard against the spellings they forward to	115
7.13	The interchange encoding, and what comes back	117
7.14	The twenty-six residual lines	120
7.15	The borders the refinement closes, and the two it must not	121
7.16	The two sets of measure zero	122
7.17	The periodic inverses on the grid, and what the rule leaves alone	124
7.18	The audit of the 192 obligations	124
8.1	The kernel across the five versions	128
8.2	The external corpus, block by block	129
8.3	Four variants of the sweep	130
8.4	The surcharge by block of the roadmap	132
8.5	The surcharge by compilation path	132
8.6	The surcharge against the cost of the operation	133
8.7	The cost of carrying a decoration	135
9.1	What the work leaves behind	140
A.1	The seventy-one changes to TIGHT	144
D.1	The operations of the corpus and the functions of TIGHT	154

List of Algorithms

6.1	Printing a literal that is proved to enclose	78
6.2	The bisection of the reverse operations	91
7.1	The exact reduction	108

Notation and Acronyms

The following conventions are used throughout the thesis. Every symbol is also defined where it first occurs; this section collects the ones that recur across chapters.

Standards

IEEE 1788-2015	the standard for interval arithmetic, which this thesis takes as its normative reference
IEEE 754-2019	the standard for floating-point arithmetic, on which IEEE 1788-2015 builds its Level 2
Cl. 10.5.4	a clause of IEEE 1788-2015; clauses of IEEE 754-2019 are named explicitly
Level 1 to 4	the four levels of IEEE 1788-2015: the sets of reals, the values a format can represent, their representation in memory, and their encoding in bits

Software

TIGHT	the interval library this work brings into conformance
NFG	<i>Numbers For Geometry</i> , the arithmetic kernel that TIGHT computes with
CORE-MATH	the library of correctly rounded elementary functions
ITF1788	the conformance testing framework, and the corpus of test cases it ships
wrapper, kernel	TIGHT is the wrapper, which decides and builds the intervals; NFG is the kernel, which computes on their <code>double</code> endpoints
dispatch	the computational path the wrapper takes, chosen by looking at the operands before the kernel is called
the flag	the compile-time option <code>TIGHT_IEEE1788_STRICT</code> : <code>ON</code> compiles the conformant code and <code>OFF</code> the previous one, so that the two are compared at equal source
the three paths	AVX2, SSE2 and purely scalar, the three implementations NFG chooses between at compile time; they differ in cost and at times in result, so every measurement reports them apart

Intervals

x	an interval, written bold as the standard writes it
$[a, b]$	the interval with lower endpoint a and upper endpoint b
hull, enclosure	the hull of a set of reals is the smallest interval that contains it, which Cl. 10.4 asks of every operation; an enclosure is any interval that contains it
Empty	the empty interval
Entire	the whole real line
Nal	<i>Not an Interval</i> , the result of an ill-formed construction
decoration	the tag carried beside an interval, which says what is known of the function that produced it (Cl. 11.2)
ulp	<i>unit in the last place</i> , the distance between two consecutive floating-point numbers
subnormal	a number smaller in magnitude than the smallest normal one: the step stays fixed, and what is consumed are the significant digits
realmax	the largest finite number of the format

Accuracy

<i>valid</i>	the result contains the exact image
<i>accurate</i>	the result is at most one <i>ulp</i> per endpoint wider than the tightest one
<i>tightest</i>	the result is the smallest representable interval that contains the exact image

Verification

corpus	the test cases ITF1788 ships in its <code>.it1</code> files, one line each: the operation, its operands and the answer expected of a conformant library
plugin	the table that says which function of TIGHT answers each operation of the corpus; the cases are those of ITF1788, the plugin is what this work adds to them
residual operand shapes	the lines of the corpus that still fail once the work is done the identifiers the benchmark gives to the operand configurations it measures, such as <code>finito</code> , <code>empty</code> or <code>semisup</code> : they keep the names the bench uses, and are listed with their examples in Table 4.6

Labels

roadmap	the plan agreed with the supervisor, which divides the work into blocks, one per group of clauses, from <code>G.0</code> to <code>G.13</code>
<code>G.N</code>	an entry of the gap analysis, one block of the roadmap
<code>M.N.K</code>	one atomic change to TIGHT, the K -th of those that close <code>G.N</code>
<code>N.k</code>	one defect of NFG against IEEE 754-2019
RQ1, RQ2, RQ3	the three research questions of Section 1.3

Acronyms

FTIA	the Fundamental Theorem of Interval Arithmetic
NaN	<i>Not a Number</i> , the answer a floating-point operation returns where no real value exists; it is a number of the format, unlike <code>Nal</code> , which is an interval that is not one
FMA	<i>fused multiply-add</i> , the operation $ab + c$ rounded once instead of twice
API	the set of names a library exposes to the code that uses it
AVX2, SSE2	the vector instruction sets against which NFG is compiled, alongside a purely scalar path

Chapter 1

Introduction and Motivation

This chapter states the problem this thesis addresses, the questions that follow from it, and what the work contributes. Section 1.1 gives the setting. Geometric algorithms decide by the sign of an arithmetic expression, exact arithmetic answers correctly and too slowly, and interval arithmetic is the compromise that keeps the guarantee at a price in width. Section 1.2 states the gap the work starts from, which is the distance between a library that rounds correctly and one that conforms to IEEE 1788-2015. Section 1.3 turns that gap into three research questions, and Section 1.4 lists the four contributions, each with the figure that measures it. Section 1.5 is a map of the rest of the document.

1.1 Robust geometric computation and the cost of exactness

Many applications in computer graphics rest on geometric computation: collision detection, minimum distance computation, element inversion and Boolean operations are all built on it. The account of the problem given in this section and the next follows that of TIGHT [SAP25], the library this thesis works on, and the claims made here about exact arithmetic, about interval arithmetic and about the cost of the two are the ones argued there. An algorithm of this kind spends most of its effort deciding, rather than computing. It asks whether a point lies inside or outside a region, whether three points turn left or right, whether two moving primitives touch. Each of those questions is answered by the sign of an arithmetic expression, and the algorithm then builds its data structures on the answer.

A wrong sign is therefore not a small numerical error. It is a decision the algorithm takes and never revisits, and the structure built on it is inconsistent with the structures built on the decisions around it. Floating point arithmetic is fast, but it is inexact, and the results it produces can be inaccurate enough to lead to unpredictable and often catastrophic outcomes. The rounding error of a single operation is tiny; the consequence of one sign that error reverses is not.

The direct remedy is to stop rounding. Exact computation, by rational arithmetic or by arbitrary-precision floating point, always returns the right sign, and it is extremely slow

and often impractical. It also has a limit that no amount of precision removes: those methods are exact only for algebraic operations. As soon as an expression contains a square root taken beyond the algebraic case, a trigonometric function or an exponential, the exact answer is not a number the machine can write down, and exact arithmetic has nothing to say about it.

Interval arithmetic bridges that gap. Every real quantity is represented by an interval whose endpoints are floating-point numbers, and every operation is performed conservatively, so that the interval computed for an expression is guaranteed to contain the exact value of that expression. The guarantee is what makes the method useful for deciding. When the interval computed for a predicate does not contain zero, its sign is the sign of the exact value and the decision is certain; when it does contain zero, the arithmetic reports that it cannot decide, and the program falls back on a slower and more precise method for that case alone. The price of the guarantee is a moderate penalty over plain floating point, rather than the order of magnitude that exact arithmetic costs.

What the method costs in accuracy is width. An interval grows at every operation, and a computation that lets it grow faster than necessary ends up unable to decide anything. Width is contained by computing each elementary operation so that it returns the tightest floating-point interval containing the exact result, which is what a correctly rounded operation does. Correct rounding buys a second property along the way. The result is defined by the operands alone, so an expression evaluated with correctly rounded operations returns the same value on any machine and against any correctly rounded implementation, and a program that goes wrong goes wrong reproducibly.

This thesis works on `TIGHT` [SAP25], a C++ library that follows that principle to its conclusion: it extends `NFG`, the fastest implementation of interval arithmetic on algebraic operations, with the transcendental functions of `CORE-MATH`, and every one of its operations is correctly rounded. The library is fast, it is tight, and it answers the question it was built to answer. The question this thesis asks is a different one, and Section 1.2 states it: whether a library that rounds correctly is, for that reason, a library that behaves as the standard for interval arithmetic requires.

1.2 Problem statement

Correct rounding and conformance are two different properties, and a library can have the first without the second. Correct rounding says how tight the result of an operation is once that operation has a result. IEEE 1788-2015 [IEE15], the standard for interval arithmetic, says more than that. It says which operations an implementation must provide, what each of them must return when an argument leaves the domain of the function or when an operand is empty or unbounded, how an operation reports that it could not do what was asked, and what an implementation must document before it may call itself conformant. A library can return the tightest possible interval on every ordinary pair of operands and still answer, on the cases the standard legislates, something the standard forbids.

`TIGHT` is in that position. The gap has three shapes. Some operations return a value that

is not the one the standard prescribes: the equality test, for one, compared an interval against another by asking whether both were the same single point, so that two intervals with the same endpoints compared unequal, and the inequality operator was not the negation of it. Other operations that the standard requires are absent, so that no answer is wrong because no answer is given. Whole mechanisms are missing: decorations, which carry alongside an interval what is known about the computation that produced it, the exception mechanism that reports a failure the returned value cannot express, the textual input and output of intervals, and the declarations without which a claim of conformance cannot be made at all.

The problem does not stop at the boundary of the library. TIGHT is built on NFG, whose kernel is written without branches in order to be fast, and that kernel is itself an implementation of arithmetic on pairs of floating-point numbers, subject to IEEE 754-2019 [Mic19]. On a small number of operations it departs from that standard. The distinction that matters is where a departure can be repaired. A wrapper can repair only what it recognises in the operands before it calls: an empty interval, an infinite endpoint, a divisor that touches zero. When the result is wrong on ordinary finite operands, the wrapper would have to know the right answer already, and the defect has to be repaired at its source.

One constraint runs through all of this. TIGHT is fast, and it is used; speed is the reason it exists and the reason its kernel has no branches. Conformance is bought with tests, and a test is a branch. The question is therefore not only how a library of this kind is made conformant, but what conformance costs when it is, and whether the cost falls where it can be afforded. That question shapes the whole of this work, and Section 1.3 puts it in the form in which it is answered.

1.3 Goals and research questions

The goal of this work is to establish whether a library of this kind can be made conformant to IEEE 1788-2015 without giving up what makes it useful, and to say what that conformance costs. Four objectives follow from it. The first is to state how the two standards involved relate to one another, since one of them governs the numbers that serve as endpoints and the other governs the intervals built on them. The second is to find, clause by clause, every point at which TIGHT departs from what the standard requires, and to repair each departure where it belongs, which is not always inside TIGHT itself. The third is to verify the result against a criterion this work did not write, because a test suite produced alongside the code it tests inherits the blind spots of that code. The fourth is to measure what the repairs cost, operation by operation, rather than to argue about it. We put those objectives in the form of three research questions, written RQ1, RQ2 and RQ3 and referred to by those names in the chapters that answer them.

RQ1. *Do IEEE 754-2019 and IEEE 1788-2015 specify the same thing in two ways, or do they govern different objects?* The question is not a formality for an implementation. It decides whether conformance to one of them says anything about conformance to the other, and it decides at which of the two levels a given obligation has to be met.

Section 2.4 answers it from the text of the two documents, and the answer runs through the whole of this work.

RQ2. *Can conformance be obtained by acting on the wrapper alone, or must the numerical kernel be changed as well?* The wrapper sees the operands before the kernel does, and it can therefore answer for the cases it recognises there without touching what lies underneath. Whether that is enough depends on the nature of each departure, and Chapter 5 states the criterion that separates the two cases and applies it to the ones this work found.

RQ3. *What does conformance cost, and does the cost fall where it can be afforded?* A test costs time even when the branch it guards is never taken, and the operations of the library differ by several orders of magnitude in what they cost to begin with, so a single figure would say nothing. Chapter 8 answers with a measurement of each block of changes against the same library built without them, on the platform described in Chapter 4.

Section 1.4 states what this work delivers in answering the three questions.

1.4 Contributions

This work delivers four things, and each of them is stated here with the figure that measures it.

1. **A conformant version of the library.** TIGHT is brought into conformance with IEEE 1788-2015 in the set-based flavor, in the compilation that carries the flag `TIGHT_IEEE1788_STRICT`, by seventy-one atomic changes grouped in the thirteen blocks of a roadmap agreed with the advisor. Each change carries the clause that requires it, the commit that applies it, the test that exercises it and the measurement of what it costs. Conformance is not concluded from the fact that the blocks are closed: the sentences of the body of the standard that state an obligation with the word *shall* are one hundred and ninety-two, and each of them was judged on its own. That audit found two obligations that no block of the roadmap named. Chapter 7 describes the result and Appendix A documents it change by change.
2. **Four defects repaired in the numerical kernel, and accepted by the project that owns it.** The analysis of NFG against IEEE 754-2019 produced nine entries. Four of them were repaired in our fork and are now part of the official upstream, N.1 with one merge and N.3, N.4 and N.5 with another. They are the only point at which a correction born of this work entered a project that is not ours, and they concern the layer on which TIGHT builds its arithmetic rather than TIGHT itself. Chapter 5 states the criterion that sends a defect there rather than to the wrapper, and applies it to all nine.
3. **An external verification platform.** Conformance is verified against a corpus of test cases written by other people for the standard itself, driven by a plugin that declares to it the operations TIGHT provides. The number of assertions the

corpus exercises grew from 4358 to 9433 as the operations the standard requires were added, and every change in the registry is measured against it before and after. Where the corpus has nothing to say, because an operation has no reference answer to compare against, the verification is written as a property that must hold rather than as a value that must match. Chapter 4 describes both.

4. **A measurement of what conformance costs.** The benchmark compares the same library built with the flag against the same library built without it, on three compilation paths and along three axes, and it gives all forty-two entries from `G.3` to `G.13` their own figures instead of an estimate. Chapter 8 reports them, and Chapter 4 states the protocol under which they were taken, including how the noise of the measurement was calibrated.

The work was carried out in contact with the authors of the two libraries: Federico Sichetti, who wrote TIGHT and gave access to its repository, and Marco Attene, who wrote NFG and accepted four of the corrections reported in Chapter 5 into its official repository. The changes that left the fork did so through the ordinary review of those projects, and the design decisions taken along the way were discussed with them.

One number belongs here rather than to the chapter that produces it. After the last change, twenty-six rows of the corpus out of 9433 still fail. None of them is a violation of an obligation, and twelve of them fail because TIGHT answers an interval that is narrower than the one the corpus expects and still encloses the exact preimage. Section 8.1.3 accounts for all twenty-six, one case at a time.

1.5 Structure of the thesis

The rest of this thesis is divided into two parts, which separate what the work stands on from what it did.

Part I gives the ground.

Chapter 2 states the two standards. From IEEE 754-2019 it takes the format that carries the endpoints, the rounding directions that make an enclosure possible and the exceptions; from IEEE 1788-2015 the levels, the flavors, the required operations, the decorations and the accuracy modes. It closes by answering RQ1.

Chapter 3 places the work among the interval libraries that exist, the frameworks that test conformance and the libraries of correctly rounded elementary functions, and it describes NFG and TIGHT in the versions this work starts from.

Part II is the work itself.

Chapter 4 states what a verification platform had to provide, and how the corpus, the benchmark and the registry were built to provide it.

Chapter 5 analyses the kernel against IEEE 754-2019 and then opens the account of the changes with the constants and the semantics of evaluation.

Chapter 6 carries the constructors, the boolean, set and cancellative operations and the twelve reverse-mode operations, which the library did not have at all.

Chapter 7 carries what the standard requires beyond arithmetic, the exceptions, the decorations, the reductions and the claim, and it closes with the audit of the whole standard. The three together answer RQ2.

Chapter 8 reports what was measured, both the conformance obtained and the cost of obtaining it, and it answers RQ3.

Chapter 9 returns to the three questions, and states what the work leaves behind and what it does not settle.

Four appendices carry what would not fit in the account without breaking it.

Appendix A is the change registry, one entry per change.

Appendix B is the protocol of the measurements, in the detail needed to repeat them.

Appendix C is the probe that established the defects of the kernel, the rules it checks and how to run it.

Appendix D is the map from the corpus to the code: for each operation, the function of TIGHT that answers it and the changes that made it conformant.

Part I

Context

Chapter 2

Background

This chapter states what the rest of the thesis stands on. Section 2.1 gives the part of IEEE 754-2019 that interval arithmetic uses: the format that carries the endpoints, the rounding directions that make an enclosure possible, the exceptions and the fused multiply-add. Section 2.2 gives interval arithmetic itself, the property it guarantees and the price it pays in width. Section 2.3 gives IEEE 1788-2015, which is the specification this work has to satisfy. Section 2.4 answers RQ1 by saying how the two standards relate.

2.1 Floating-point arithmetic and IEEE 754-2019

Interval arithmetic is built on top of floating-point arithmetic. The endpoints of an interval are floating-point numbers, and the operations that move those endpoints are the operations of IEEE 754-2019 [Mic19]. What follows is the part of that standard the rest of the thesis uses, and no more: an interval library needs to know how a number is stored, in which direction an operation may round it, and what it reports when it cannot answer.

2.1.1 Formats and the binary64 representation

A binary floating-point format is a finite set of numbers determined by three integer parameters (754 Cl. 3.3): the precision p , which is the number of significant bits, and the two exponent bounds $e_{\max}$ and $e_{\min}$, for which the standard fixes $e_{\min} = 1 - e_{\max}$. A finite number of the format has the value

$$x = (-1)^s \times m \times 2^e, \quad s \in \{0, 1\}, \quad e_{\min} \leq e \leq e_{\max},$$

where the significand m is written with p bits, so that $1 \leq m < 2$ for a normal number and $0 \leq m < 1$ for a subnormal one. The largest significand is therefore $2 - 2^{-52}$, one *ulp* below two, which is where the value of `realmax` in Table 2.1 comes from. That table gives the three parameters for binary64, which is the format used throughout this work and the one provided by the C++ type `double` on the platforms considered here.

Parameter	Value
Precision p , in bits	53
Maximum exponent $e_{\max}$	+1023
Minimum exponent $e_{\min}$	-1022
Storage width, in bits	64
Largest finite number realmax	$(2 - 2^{-52}) \times 2^{1023}$
Smallest positive normal number	2^{-1022}
Smallest positive subnormal number	2^{-1074}
Distance between 1 and its successor	2^{-52}

Table 2.1: Parameters of the binary64 format, from Table 3.2 and Table 3.5 of IEEE 754-2019. The four values in the lower half are derived from the three parameters above them, and are the ones this thesis refers to by name. The precision p counts the leading significant bit, which the format does not store: the word of Figure 2.1 carries fifty-two bits of fraction, and the fifty-third is implied by the exponent field.

Those parameters are carried in memory by the sixty-four bits of Figure 2.1, divided into three fields. The leading bit is the sign s ; the next eleven hold the exponent, stored with a bias so that the field E is a non-negative integer; the remaining fifty-two hold T , the fraction of the significand. The leading significant bit is not stored at all, being known from E , which is how fifty-three bits of precision fit in fifty-two:

$$x = \begin{cases} (-1)^s (1 + T \cdot 2^{-52}) 2^{E-1023}, & 1 \leq E \leq 2046, \\ (-1)^s (T \cdot 2^{-52}) 2^{-1022}, & E = 0. \end{cases} \quad (2.1)$$

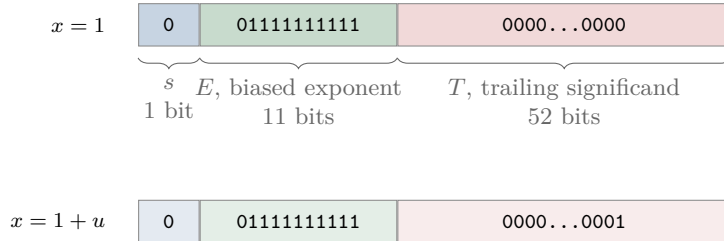


Figure 2.1: The sixty-four bits of a binary64 number, with the three fields of Equation (2.1). The fraction shows its first four bits and its last four, the forty-four in between being elided. The two words are 1 and its successor: they differ in the last bit alone, and that difference is the quantity $u = 2^{-52}$ used throughout this thesis. Reading the word as an unsigned integer orders the non-negative numbers of the format in the same order as their values, which is why the successor of a number is obtained by incrementing its bit pattern, and the predecessor by decrementing it.

A number whose exponent is $e_{\min}$ and whose leading significant bit is zero is called *subnormal*: it is a non-zero number whose magnitude is smaller than that of the smallest

normal number, and it does not use the full precision available to the normal numbers of the same format (754 Cl. 2.1). Subnormal numbers fill the gap between zero and 2^{-1022} , at the cost of significant bits, until the last of them, 2^{-1074} , is reached.

Besides the finite numbers a format carries two zeros, $+0$ and -0 , two infinities, and NaN, which stands for Not a Number (754 Cl. 3.2). NaNs are of two kinds. A quiet NaN propagates through an operation and marks its result as the outcome of something that had no numerical answer. A signalling NaN signals the invalid operation exception when an operation reads it, and is therefore never produced by an arithmetic operation: a program writes one to mark storage that has not been initialised, or a value that is missing (754 Cl. 6.2). The distinction is not academic, and Section 2.1.3 describes the mechanism it drives.

All of those live in the same sixty-four bits, at the two ends of the exponent field that (2.1) leaves out, and Table 2.2 lists them. Three consequences of that table are used later in this thesis and are easier to accept once the pattern is in view. The two zeros differ in the sign bit and in nothing else, so they are the same real number stored as two distinct words, which is how a library can return the right value and the wrong bit pattern at once. The two infinities have a fraction field that is entirely zero, so asking whether an interval is the degenerate pair $[+\infty, +\infty]$ is a comparison of two integers rather than of two floating-point numbers. And the two kinds of NaN differ in the leading bit of the fraction alone, so that writing a signalling NaN rather than a quiet one is a decision a program makes deliberately, one bit at a time. Since no arithmetic operation produces a signalling NaN, a library that answers with one is saying something about how the value came to be, and Section 8.2.2 reports what that decision turned out to cost on one of the three compilation paths.

Table 2.2: The classes of a binary64 word, by the contents of its two fields (754 Cl. 3.4). The sign bit is free in every row, so each class except the NaNs comes in two, one of either sign.

Exponent field E	Fraction field T	Class
all zeros	zero	$+0$ and -0
all zeros	non-zero	subnormal numbers
1 to 2046	any	normal numbers
all ones	zero	$+\infty$ and $-\infty$
all ones	non-zero, leading bit one	quiet NaN
all ones	non-zero, leading bit zero	signalling NaN

The distance between two consecutive numbers of a format is not constant: it doubles at every power of two. We write *ulp* for that distance, after *unit in the last place*, and we write $u = 2^{-52}$ for the distance between 1 and the next larger binary64 number. Since the set is finite, the exact result of an operation on two of its elements is in general not one of its elements, and the standard prescribes how that exact result is replaced by a representable one.

2.1.2 Rounding-direction attributes

Rounding takes a number regarded as infinitely precise and modifies it to fit in the destination format (754 Cl. 4.3). The standard defines five rounding-direction attributes. Two of them round to nearest and differ only in how they resolve a tie: `roundTiesToEven` delivers the candidate whose last significant bit is zero, and `roundTiesToAway` delivers the one with the larger magnitude (754 Cl. 4.3.1). The remaining three are the directed attributes `roundTowardPositive`, `roundTowardNegative` and `roundTowardZero` (754 Cl. 4.3.2). For results in binary formats `roundTiesToEven` is the default (754 Cl. 4.3.3).

The five are told apart by a value that falls exactly halfway between two representable ones, where every attribute has to choose. Table 2.3 takes $1 + u/2$, the midpoint between 1 and its successor, and its negative, which is enough to separate all five: the two to nearest differ on the tie, the two outward ones differ from each other by the direction they take, and `roundTowardZero` changes side with the sign.

Rounding-direction attribute	$1 + u/2$	$-(1 + u/2)$
<code>roundTiesToEven</code>	1	-1
<code>roundTiesToAway</code>	$1 + u$	$-(1 + u)$
<code>roundTowardPositive</code>	$1 + u$	-1
<code>roundTowardNegative</code>	1	$-(1 + u)$
<code>roundTowardZero</code>	1	-1

Table 2.3: The five rounding-direction attributes of IEEE 754-2019, on a value that lies exactly halfway between two representable numbers. The quantity $u = 2^{-52}$ is the distance between 1 and its successor, so $1 + u/2$ is the midpoint of that gap and belongs to no format. `roundTiesToEven` takes 1 because its last significant bit is zero. The pair that an interval uses is `roundTowardNegative` for the lower endpoint and `roundTowardPositive` for the upper one: they are the only two that move away from the exact value on both signs.

An operation is *correctly rounded* when it delivers the result that would be obtained by computing the exact value with unbounded range and precision and then rounding that value once, in the direction in effect (754 Cl. 5.1). Correct rounding makes the result of an operation a function of its operands and of the rounding direction alone. Two implementations that both round correctly agree bit for bit, and the error of a single operation is bounded by construction rather than estimated.

The two outward directed attributes are what makes interval arithmetic possible. When the exact result of an operation falls between two representable numbers, an enclosure of that result is obtained by computing the lower endpoint under `roundTowardNegative` and the upper endpoint under `roundTowardPositive`. Rounding to nearest would move an endpoint inwards half of the time, and the computed interval would then no longer be guaranteed to contain the exact one, which is the property stated in Section 2.2.1.

The rounding direction is a dynamic mode (754 Cl. 4.2): in a C or C++ environment it

is global state, written with `fesetround` and read with `fegetround`. Two consequences follow, and both are met in Chapters 5 to 7. A function of the standard library obeys whatever direction is installed when it is called, whether or not its caller intended it to; and a library that keeps one direction installed for its own arithmetic imposes that direction on every other computation running in the same thread.

2.1.3 Exceptions and status flags

An exception is an event that occurs when an operation on particular operands has no outcome suitable for every reasonable application (754 Cl. 2.1). The standard defines five kinds of exception (754 Cl. 7.1):

- *invalid operation*, signalled when there is no usefully definable result, and whose default result is a quiet NaN (754 Cl. 7.2);
- *division by zero*, signalled when an operation on finite operands has an exact infinite result (754 Cl. 7.3);
- *overflow*, signalled when the exact result exceeds in magnitude the largest finite number of the destination format (754 Cl. 7.4);
- *underflow*, signalled when a non-zero result is smaller in magnitude than the smallest normal number (754 Cl. 7.5);
- *inexact*, signalled when the delivered result differs from the exact one (754 Cl. 7.6).

Under default exception handling the operation delivers a result anyway, and raises a status flag to record what happened. One invocation of an operation signals at most one exception, although the default handling of that exception may signal further ones: overflow signals inexact, and underflow signals inexact when its default result is inexact (754 Cl. 7.1).

The five flags are global to the thread and sticky. Once raised, a flag stays raised until the program clears it, so a flag that is set reports only that some operation since the last clearing signalled that exception. It does not report which operation, nor how many. Reading a flag therefore establishes a property of a block of code, not of a single operation. This is both what makes the flags cheap and what limits what can be concluded from them, and Chapter 7 returns to the point.

2.1.4 The fused multiply-add

The operation `fma( $x, y, z$ )` computes $(x \times y) + z$ as if with unbounded range and precision, rounding only once to the destination format (754 Cl. 5.4.1). It is one operation, not two: an underflow, overflow or inexact exception can arise only from the rounding of the exact value $(x \times y) + z$, and never from an intermediate product that does not exist.

Two properties follow, and both are used in this thesis. The result is at least as tight as the one obtained by a multiplication followed by an addition, because one rounding is applied where two would be. And the operation gives access to the rounding error of a product: for finite x and y , the value $\text{fma}(x, y, -(x \times y))$ is the difference between the exact product and its rounded value, computed exactly whenever that difference is representable. Algorithms that accumulate a sum of products without losing bits are built on that identity; the exact reduction of Chapter 7 reaches the same end by another road, accumulating the integer significands in fixed point.

2.2 Interval arithmetic

Interval arithmetic replaces a number that is not known exactly by a set that is known to contain it, and replaces each operation by one that carries that containment forward. This section states the property that makes the method useful and the price it pays for it. The definitions are those of the set-based flavor of IEEE 1788-2015, which is the one this work implements.

2.2.1 Intervals, containment and the Fundamental Theorem

The intervals of the set-based flavor are the subsets of the real line that are closed and connected, that is the empty set, written **Empty**, together with the sets

$$[\underline{x}, \bar{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \bar{x}\},$$

where the bounds $\underline{x}$ and $\bar{x}$ are extended reals satisfying $\underline{x} \leq \bar{x}$, $\underline{x} < +\infty$ and $\bar{x} > -\infty$ (Cl. 10.2). The set of such intervals is written $\mathbb{IR}$. A bound may be infinite, and $[-\infty, +\infty]$, called **Entire**, is the whole real line; but an infinity is never a member of an interval, so **Entire** contains every real number and neither infinity. The square brackets are the notation of the standard and are kept throughout this thesis, although an infinite bound is excluded from the set and the round bracket of ordinary usage would say so on sight. What the brackets delimit here is the *pair of bounds*, and membership is settled by the definition above, which admits only real x . The empty set is given the bounds $\inf \text{Empty} = +\infty$ and $\sup \text{Empty} = -\infty$, which is the one convention that makes the map from a valid pair of bounds to the interval it denotes one to one.

The *hull* of an arbitrary set s of reals, written $\text{hull}(s)$, is the tightest interval that contains it (Cl. 10.3). It is what turns a set that is not an interval, such as the image of a function that has a gap, into one that is.

Given a real function f of n variables, an *interval extension* of f is a total map $\mathbf{f}$ from boxes to intervals such that $f(x) \in \mathbf{f}(\mathbf{x})$ whenever $x \in \mathbf{x}$ and $f(x)$ is defined; equivalently, $\mathbf{f}(\mathbf{x})$ contains $\text{Rge}(f \mid \mathbf{x})$, the set of values f takes on the part of the box where it is defined, which is the notation of the standard (Cl. 10.4). Among all interval extensions of f there is a smallest one, the *natural interval extension*

$$\mathbf{f}(\mathbf{x}) = \text{hull}(\text{Rge}(f \mid \mathbf{x})),$$

which encloses exactly the values the function actually takes on the part of the box where it is defined, and nothing more. The standard gives three examples that show what the definition does at the edges of a domain: $\sqrt{[-1, 4]} = [0, 2]$, because the square root is undefined on the negative part and the definition simply ignores it; $\sqrt{[-2, -1]} = \text{Empty}$, because there is nothing left; and $\mathbf{x}/[0, 0] = \text{Empty}$ for every $\mathbf{x}$ (Cl. 10.4).

For the four basic operations the natural extension is read off the endpoints, since each of them is monotone in either argument on the part of the plane where it is defined:

$$\begin{aligned} \mathbf{x} + \mathbf{y} &= [\underline{x} + \underline{y}, \bar{x} + \bar{y}], & \mathbf{x} - \mathbf{y} &= [\underline{x} - \bar{y}, \bar{x} - \underline{y}], \\ \mathbf{x} \cdot \mathbf{y} &= [\min P, \max P], & \mathbf{x}/\mathbf{y} &= \mathbf{x} \cdot [1/\bar{y}, 1/\underline{y}] \quad (0 \notin \mathbf{y}), \end{aligned} \quad (2.2)$$

where P is the set of the four products of an endpoint of $\mathbf{x}$ by an endpoint of $\mathbf{y}$. The product takes the extremes over those four because a product changes direction with the sign of its operands, and the division is written through the reciprocal only where the divisor keeps one sign: the case $0 \in \mathbf{y}$, where the exact range is unbounded and in general not an interval, is one of the places where TIGHT was not conformant, and Chapter 5 returns to it.

What makes the method usable is that these properties survive composition. The result that states it goes back to Moore [Moo66], and IEEE 1788-2015 carries it as Theorem 6.1 of Cl. 6.4.

Theorem 2.1 (Fundamental Theorem of Interval Arithmetic). *Let f be an arithmetic expression built from library functions $\varphi_1, \dots, \varphi_k$, and let $\mathbf{y}$ be the result of evaluating f over a box $\mathbf{x}$ using any interval extensions $\varphi_1, \dots, \varphi_k$ of those functions. Then*

$$\mathbf{y} \supseteq \text{Rge}(f \mid \mathbf{x}).$$

Proof. By induction on the structure of the expression. If f is a variable x_i , the evaluation returns $\mathbf{x}_i$ and the claim is the hypothesis $x_i \in \mathbf{x}_i$; if f is a constant, both sides are that constant.

For the inductive step let $f = \varphi(g_1, \dots, g_m)$, and let $\mathbf{u}_j$ be the interval computed for g_j . Take any $x \in \mathbf{x}$ at which f is defined. Each g_j is then defined at x , and by the induction hypothesis $g_j(x) \in \mathbf{u}_j$. So the point $(g_1(x), \dots, g_m(x))$ lies in the box $\mathbf{u}_1 \times \dots \times \mathbf{u}_m$ and φ is defined there, whence by the defining property of an interval extension

$$f(x) = \varphi(g_1(x), \dots, g_m(x)) \in \varphi(\mathbf{u}_1, \dots, \mathbf{u}_m) = \mathbf{y}. \quad \square$$

Two things about that proof are worth noticing, because the rest of this thesis rests on both. The first is that it never uses tightness: any extension will do, and a wider one only makes $\mathbf{y}$ wider. Containment is therefore a property of the *weakest* link and cannot be lost by an implementation that overestimates, which is why Cl. 12.10.2 of IEEE 1788-2015 can require *tightest* of a few operations and merely *valid* of the rest without endangering the theorem. The second is that the induction carries whatever invariant the interval extensions themselves guarantee. Running it with *defined* in place of *contained* gives the second form of the theorem: if every library operation is everywhere defined on its inputs

during the evaluation, then f is everywhere defined on $\mathbf{x}$. Running it with *defined and continuous* gives the third, and with *nowhere defined* the fourth. Those three forms are exactly the statements the decorations of Section 2.3.4 propagate, and the min-rule they use is the inductive step written for an implementation.

The standard is built so that the theorem holds in finite precision and not only at Level 1 (Cl. 6.4), which is the reason the accuracy requirements of Section 2.3.5 are stated as they are: a Level 2 operation must return an interval containing the Level 1 one, so the induction goes through unchanged when φ is read as the implemented operation rather than the mathematical one.

The basic form is the one that a program deciding a sign relies on. If the interval computed for a predicate does not contain zero, then the exact value does not either, and the sign of the interval is the sign of the exact value. If it does contain zero, the theorem says nothing, and the program has learnt that this input needs another method.

2.2.2 Dependency and tightness

An interval extension is allowed to be wider than the range it encloses, and in practice it is. Two different things make it so, and only one of them is a matter of arithmetic.

The first is rounding. Each operation must move its endpoints outwards to representable numbers, so each operation may add up to one *ulp* at each end. This part is bounded, it is what correct rounding minimises, and it is the part a library controls.

The second is dependency, and it is a property of the expression rather than of the arithmetic. An interval extension evaluates each operation as though its operands were independent, so an expression that mentions the same variable more than once loses the fact that the two occurrences carry the same value. Evaluating $x - x$ over $\mathbf{x} = [0, 1]$ gives $[0 - 1, 1 - 0] = [-1, 1]$, by the definition of the natural extension of subtraction applied to the two operands separately, where the range of the function $x \mapsto x - x$ is the single point $[0, 0]$. No amount of precision removes that gap: the two intervals differ at Level 1, before any rounding takes place. This is why an interval computation is written so that each variable occurs as few times as possible, and why an implementation is judged on how tight each single operation is rather than on the width of an arbitrary expression.

The strongest an implementation can be on a single operation is to return the tightest interval of its type that contains the exact result. IEEE 1788-2015 turns that into a graded requirement, with three named modes, and Section 2.3.5 states which mode is required of which operation.

2.3 IEEE 1788-2015

IEEE 1788-2015 [IEE15] specifies interval arithmetic for programming environments. It says which operations an implementation must provide and what each of them must

return, including on the inputs that have no ordinary answer; how an implementation reports that an operation could not do what was asked; and what an implementation must document before it may claim conformance. This section states the parts of it that the rest of the thesis uses.

2.3.1 The four levels of specification

The standard is structured into four levels, which match those of IEEE 754-2008, the edition current when IEEE 1788-2015 was published, and are summarised in its Table 5.1 (Cl. 5).

Level 1 is mathematics: the intervals of Section 2.2.1 and the operations on them, defined without reference to any finite representation. Level 2 is discretization, and the standard calls it its central part. Its entities are *datums*, that is mathematical intervals tagged with the finite set they belong to; such a set is an *interval type*, for instance the intervals whose endpoints are binary64 numbers. Level 3 is representation, where an interval datum becomes an object, usually a pair of floating-point values. Level 4 is the encoding of those objects into bit strings.

Two of the maps between levels matter later. From Level 1 down to Level 2 the map is the *T-hull*: every mathematical interval has a unique smallest enclosing datum of the type *T*, which is how an operation whose exact result is not representable acquires a result that is. From Level 2 up to Level 1 the map merely forgets the type tag. The Level 3 and Level 4 requirements are few and concern mainly the conversion to and from external representations.

The distinction decides questions rather than recording them, and Chapters 5 to 7 turn on it twice. An obligation stated at one level is not the same obligation at another: at Level 1 an operation may simply have no value, whereas at Level 2 an operation always returns something, and the standard prescribes what (Cl. 12.1.3). The midpoint of *Empty* is the plainest case, since Table 10.2 of Cl. 10.5.9 gives it *no value* while Cl. 12.12.8 requires that it return *NaN*.

2.3.2 Flavors, and the set-based flavor

The standard does not fix one interval model. A *flavor* is an interval model that conforms to a core specification, stated in Clause 7 together with Clause 8 and Clause 9, and an implementation shall provide at least one *standard* flavor, that is one specified by the standard itself (Cl. 7.1). The set-based flavor, described by Clause 10 through Clause 14, is currently the only standard flavor, and it is the one whose intervals are the sets of Section 2.2.1. A flavor is a property of the execution context and not of an individual interval, so exactly one flavor is in force at any point of execution.

An implementation of the set-based flavor shall provide the decorations of Cl. 11.2, at least one supported bare interval type, the required operations of Cl. 12.12 with the accuracies of Cl. 12.10, the recommended operations of Cl. 12.13 in the specified form if

it provides them at all, and the conversions between intervals and strings of Cl. 13.2 to Cl. 13.4 (Cl. 4.2). Everything this thesis does is inside this flavor.

2.3.3 The required operations

What an implementation must provide is fixed by four tables, and the distinction between what the standard requires and what it recommends is the distinction between *shall* and *should*.

Table 9.1 (Cl. 9.1) lists the required forward elementary functions, grouped in five categories that reappear in the accuracy requirements: the *basic operations* **neg**, **add**, **sub**, **mul**, **div**, **recip**, **sqr**, **sqrt** and **fma**; the power functions **pown**, **pow**, the exponentials and the logarithms; the trigonometric and hyperbolic functions with their inverses, including **atan2**; the *integer functions* **sign**, **ceil**, **floor**, **trunc**, **roundTiesToEven** and **roundTiesToAway**; and the *absmax functions* **abs**, **min** and **max**. Each row gives the domain and the range of the point function, which is what the natural interval extension of Section 2.2.1 is taken over. Two constant functions, **empty()** and **entire()**, are required as well (Cl. 10.5.2).

Table 10.1 (Cl. 10.5.4) lists the required reverse-mode operations, which recover an argument from a result. For a unary function ϕ , a reverse interval extension satisfies

$$\phi\text{Rev}(\mathbf{c}, \mathbf{x}) \supseteq \{x \in \mathbf{x} \mid \phi(x) \text{ is defined and lies in } \mathbf{c}\},$$

and the binary case has two forms, one for each argument recovered. There are twelve of them, and they are the subject of one of the blocks of Chapter 6.

Table 10.2 (Cl. 10.5.9) lists the numeric functions, whose argument is an interval and whose result is a number: **inf**, **sup**, **mid**, **wid**, **rad**, **mag** and **mig**. Some of them may return an infinity, and the table says which. Table 10.3 (Cl. 10.5.10) lists the eight boolean comparisons between two intervals, with their set-theoretic definitions, and Table 10.4 states what those definitions give when at least one of the two intervals is empty, which does not always follow from the formula on the bounds.

Cl. 10.6 lists the recommended operations in Table 10.5 and Table 10.6. An implementation need not provide them, but if it does, it shall provide them as specified (Cl. 4.2).

2.3.4 Decorations

A decoration is information attached to an interval, and the pair is a decorated interval (Cl. 8.1). What it describes is not the interval but the function that produced it: whether that function was defined everywhere on the input box, whether it was continuous there, whether the box was bounded. It is the mechanism by which the stronger forms of the Fundamental Theorem become checkable while the computation runs, rather than proved by hand afterwards.

The set-based flavor has five decorations, listed in the table of Cl. 11.2 in decreasing order of strength: **com**, for common, which holds when the box is bounded and nonempty, is inside the domain of f , f is continuous at each of its points and the computed interval is bounded; **dac**, defined and continuous, when the box is a nonempty subset of the domain and the restriction of f to it is continuous; **def**, defined, when the box is a nonempty subset of the domain; **trv**, trivial, which is always true and therefore carries no information; and **ill**, ill-formed, which says that the object is not an interval at all. A decorated interval whose decoration is **ill** is **Nal**, Not an Interval.

Decorations are initialised and then propagated. The operation **newDec** attaches to a bare interval the strongest decoration it deserves, which is **com** for a nonempty bounded interval, **dac** for an unbounded one and **trv** for the empty one (Cl. 11.5.1). A decorated operation then computes a *local* decoration d_{v_0} , which describes what the operation itself did on its inputs. The decoration it returns is the smallest among that one and those of its inputs,

$$d_w = \min\{d_{v_0}, d_{v_1}, \dots, d_{v_k}\},$$

which is the *min-rule* of Cl. 11.6. The minimum is taken with respect to the propagation order **com** > **dac** > **def** > **trv** > **ill**. Since **Nal** is treated as the empty interval decorated **ill**, the min-rule gives without a special case the rule that a result is **Nal** exactly when one of its inputs is.

Three further operations belong to the system: **intervalPart** and **decorationPart**, which take a decorated interval apart, and **setDec**, which puts one together and repairs the combinations the standard forbids, so that a decorated interval that claims more than it can carry cannot be constructed (Cl. 11.5.2).

2.3.5 Accuracy: **valid**, **accurate**, **tightest**

The standard grades how tight an operation is with three modes, from weakest to strongest (Cl. 12.10.1). An operation is *valid* when its result contains the exact Level 1 result, which is the enclosure property and nothing more. It is *tightest* when its result equals the T -hull of the exact result, that is when it is the best the type allows. The intermediate mode, *accurate*, is defined for inf-sup types only and asks that the result be no wider than the tightest result on a box widened by one *ulp* at each end, then itself widened by one *ulp* at each end.

The requirements are not uniform, and this is the point that decides how a failing test is to be read. The basic operations, the integer functions and the absmax functions of Table 9.1 shall be *tightest*. For every other operation of Table 9.1, for the reverse-mode operations of Table 10.1 and for the recommended operations of Table 10.5 and Table 10.6, the accuracy shall be *valid*, and for inf-sup types it should be *accurate* (Cl. 12.10.2). For any operation in those four tables, if any input is **Empty** the result shall be **Empty**. A disagreement with a reference implementation is therefore not by itself a violation: it is one only if it breaks the mode that the clause requires of that operation, and Chapter 8 reads the residue of the test corpus with this distinction in hand.

The same subclause adds an obligation that is documentary rather than numerical: an implementation shall document the tightness of each of its interval operations for each supported bare interval type, by dividing the possible inputs into disjoint ranges and stating a tightness for each (Cl. 12.10.3).

2.3.6 Exceptions and the conformance claim

At Level 1 an operation may have no value; at Level 2 it always returns one (Cl. 12.1.3). When the Level 1 value does not exist, the operation returns either a value that marks the event, such as a NaN from a numeric function, or a value that is reasonable in practice, such as `Empty` from a constructor given invalid input. Because the returned value cannot always say that something went wrong, the standard defines exceptions that may be signalled alongside it: `UndefinedOperation`, when a constructor is certain that no Level 1 value exists; `PossiblyUndefinedOperation`, when in finite precision it cannot decide; `IntvlPartOfNaN`, when `intervalPart` is applied to `NaN`; and `InvalidOperand`, when the inputs of an operation are not valid. These exceptions are separate from the exceptional conditions that the decoration system handles, and the mechanism by which a handler deals with them is left to the implementation.

Conformance itself is declared, not inferred. An implementation may claim conformance in the form given in Cl. 4.3, naming the flavor, the types for which it claims IEEE 754 conformance and whether it provides compressed arithmetic; and part of the claim shall be the completion of the questionnaire of Cl. 4.4, whose questions ask, among other things, which decorations are provided and with what mathematical definition, how exceptional cases are handled, and which operations beyond the required ones exist. A library that satisfies every numerical obligation and publishes neither the claim nor the questionnaire has not finished the job, and Chapter 7 returns here to finish it.

2.4 How the two standards relate

RQ1 asks whether IEEE 754-2019 and IEEE 1788-2015 specify the same thing in two ways or govern different objects. They govern different objects, and they are built to be used together.

IEEE 754-2019 governs numbers. It fixes what a floating-point format contains, what an operation on its members returns, and how the exact result is replaced by a representable one. IEEE 1788-2015 governs intervals. It fixes what an interval is, which operations exist on intervals, what each returns including on empty and unbounded inputs, and what an implementation must document. Neither answers the questions of the other: the first says nothing about what the square root of an interval that straddles zero should be, and the second says nothing about how many bits a significand has.

They are nonetheless connected, in three places that the standard states explicitly. The first is the normative reference: IEEE 1788-2015 lists IEEE 754-2008 as the document that must be understood and used to apply it (Cl. 2). The second is the structure: the four levels

of Section 2.3.1 are declared to match those of IEEE 754-2008, so the two specifications describe their objects at the same four degrees of abstraction, from the mathematical value down to the bits. The third, and the one that bears on this work, is the optional claim: an implementation of the set-based flavor *may* claim IEEE 754 conformance for all or part of itself, provided the requirements of Cl. 12.6 are satisfied (Cl. 4.2.1), and a 754-conforming type is defined there as an inf-sup type derived from an IEEE 754 format whose operations meet the accuracy requirements of Cl. 12.10 (Cl. 12.6.1).

Three consequences follow, and they run through the rest of the thesis. Conformance to one standard does not imply conformance to the other: a library may be conformant to IEEE 1788-2015 on a format that IEEE 754 does not describe, and an arithmetic that is impeccable on numbers may still answer, on an empty or unbounded interval, something IEEE 1788-2015 forbids. The claim of 754-conformance is about *values*: what Cl. 12.6.1 imports is the accuracy requirement on the endpoints, not the exception mechanism of Section 2.1.3, whose flags IEEE 1788-2015 deliberately replaces with the decorations of Section 2.3.4. And the two standards divide the labour of an implementation built in layers: a numerical kernel that computes on endpoints answers to IEEE 754-2019, while the wrapper that turns those endpoints into intervals answers to IEEE 1788-2015. That division is exactly the one RQ2 asks about, and Chapter 5 takes it up.

One point of vocabulary, since the two editions differ. IEEE 1788-2015 cites the 2008 edition of the floating-point standard, and this thesis reads that standard in its 2019 edition, which is the one the platforms used here implement. The parts on which this work relies, that is the formats of Section 2.1.1, the rounding-direction attributes, the five exceptions and the fused multiply-add, are specified in the same terms in both editions.

Chapter 3

Related Work

This chapter places the present work among what already exists. Section 3.1 reviews the interval arithmetic libraries that precede TIGHT and states where each of them stands with respect to IEEE 1788-2015. Section 3.2 describes the platform on which conformance can be tested from the outside, which is what makes the results of this thesis verifiable by someone who did not write the library. Section 3.3 introduces the library of correctly rounded elementary functions that TIGHT depends on. Section 3.4 describes NFG and TIGHT as we found them, in the exact versions the work starts from.

3.1 Interval arithmetic libraries

Interval arithmetic has been implemented many times, and the libraries listed in Table 3.1 are the ones this work meets, either because the standard was written with them in view or because the corpus of Section 3.2 exercises them.

Table 3.1: The interval arithmetic libraries named in this chapter, with the year of the publication that describes them. The four that precede IEEE 1788-2015 could not have been written to it.

Library	Year	Endpoints	Language
C-XSC [HK04]	2004	several formats	C++
MPFI [RR05]	2005	arbitrary precision	C
filib++ [LTWvG ⁺ 06]	2006	binary64	C++
Boost.Interval [BMP06]	2006	by policy	C++
libieeeep1788 [Neh14]	2014	by type	C++
IntervalArithmetic.jl [SBF ⁺ 22]	2022	by type	Julia

MPFI [RR05] builds intervals on arbitrary precision floating-point numbers. An endpoint carries as many bits as the caller asks for, so a result that is too wide can be narrowed by asking for more precision, and the width of a computed interval is not bounded below

by a format. The price is that an operation costs what arbitrary precision costs. TIGHT makes the opposite choice: its endpoints are binary64, the tightest result is fixed by the format, and the aim is to reach that bound at the speed of the hardware.

filib++ [LTWvG⁺06] is a C++ library whose stated aim is fast containment computations, and C-XSC [HK04] is a C++ library for scientific computing with verified result types, of which intervals are one. Boost.Interval [BMP06] is a template whose behaviour is chosen through policies: the rounding policy and the checking policy are type parameters, so one library serves several arithmetics, and what an operation does on an empty operand or outside a domain is decided by the type rather than by the operation.

These four precede IEEE 1788-2015, which was published in 2015, and cannot be conformant by construction: the standard was written after them, and in part from the experience of using them. libieeeep1788 [Neh14] stands differently, having been written on the draft that became the standard and following its structure, and IntervalArithmetic.jl [SBF⁺22] states conformance among its aims.

The distance between these libraries is not only a matter of speed or of precision, and it is the reason the standard exists: on the same operands they may legitimately answer differently, because each one decides for itself what to return when an argument leaves the domain of a function, when an operand is empty, or when an endpoint is infinite. The corpus of Section 3.2 records those decisions, and carries their names: of its 19 files, one is `c-xsc.itl`, one `mpfi.itl`, one `fi_lib.itl`, and eleven have names beginning with `libieeeep1788`.

None of these libraries was written for the purpose TIGHT has. TIGHT [SAP25] exists to give geometric computation the tightest binary64 interval it can, as fast as the hardware allows, which is a narrower aim than any of them pursue and, as Chapter 8 shows, one that makes the cost of conformance worth measuring rather than assuming.

3.2 Testing conformance to IEEE 1788-2015

A library can be read against IEEE 1788-2015 clause by clause, and this work does that as well, but a reading is an argument and not a measurement. ITF1788 [BFR24] turns it into one. Its authors write the test cases once, in a small domain specific language in which a case is an operation, its operands and the interval the standard requires, and a plugin translates the whole corpus into the test suite of a given library, in the names and the syntax that library uses.

Listing 3.1 is one case of that language, taken from the corpus as it ships. It is worth reading line by line, because the whole of Chapters 5 to 7 provides an answer to lines of this kind. The first line states that the empty interval is absorbing. The second through the fifth state what Cl. 10.4 requires of an argument that leaves the domain, which is not a failure but the enclosure of the image of the part of the argument that lies inside it, so that the square root of $[-5, 25]$ is $[0, 5]$ and not an error. The last line states tightness, and does so in hexadecimal because that is the only notation in which the two endpoints of a one *ulp* wide result can be written without ambiguity.

```

testcase minimal_sqrt_test {
  sqrt [empty] = [empty];
  sqrt [entire] = [0.0,infinity];
  sqrt [-infinity,-0x0.0000000000001p-1022] = [empty];
  sqrt [-5.0,25.0] = [0.0,5.0];
  sqrt [-5.0,infinity] = [0.0,infinity];
  sqrt [0X1.999999999999AP-4,0X1.999999999999AP-4]
    = [0X1.43D136248490FP-2,0X1.43D136248491P-2];
}

```

Listing 3.1: One test case of the ITF1788 corpus, abridged from `libieeep1788_elem.it1`. Every line is an operation, its operands and the interval the standard requires. The hexadecimal literals of the last line are two consecutive binary64 numbers: the exact square root lies between them, so no narrower enclosure exists.

Two properties of this arrangement matter here. The corpus is *external*: it was written by people who do not know TIGHT and cannot have written it around what TIGHT happens to do, so a case it fails is a fact about the library rather than about the test. And the translation is mechanical, so the cost of testing a library is the cost of writing its plugin, not of writing thousands of cases.

The corpus that ships with the framework holds 261 test cases in 19 files, and Table 3.2 groups them by where they come from. Two thirds of the cases are transcriptions of the suites of two existing libraries, and the single largest file, with eighty cases, states what the standard requires of the elementary functions. The composition matters for reading Chapter 8: a library is not measured against an abstract requirement alone, but also against what other implementations were found to do, which is why a case can fail while the library remains conformant.

Table 3.2: The 261 test cases of the ITF1788 corpus by provenance. The counts were obtained by reading the files, not from the documentation. The eleven files of the first row carry the suite of `libieeep1788`, which was written on the draft of the standard, and are the ones that state the requirement itself.

Provenance	Files	Cases
libieeep1788 [Neh14]	11	179
MPFI [RR05]	1	57
C-XSC [HK04]	1	8
flib++ [LTWvG ⁺ 06]	1	5
written against the standard	2	8
reverse mode and <code>atan2</code>	3	4
Total	19	261

How this corpus is used here, what fraction of it TIGHT can express and how the plugin was written are the subject of Section 4.2.

3.3 Correctly rounded elementary functions

An interval extension of an elementary function can be no tighter than the function it calls. Write ∇ and Δ for rounding towards $-\infty$ and towards $+\infty$, as Section 2.1.2 does. If a library computes f to within k units in the last place, then the only enclosure its caller may claim is

$$f(x) = [\nabla^k(\hat{y}), \Delta^k(\hat{y}')].$$

Here $[\underline{y}, \bar{y}]$ is the exact result, written with the bars this thesis uses for the endpoints of an interval, while $\hat{y}$ and $\hat{y}'$ are the two values the scalar library actually returns for those endpoints. Each of the two may sit up to k *ulp* away from the endpoint it approximates, on either side of it, which is why the layer has to widen them further: ∇^k and Δ^k move k steps outwards, and only then is containment certain again. Every one of those $2k$ steps is width that the interval layer adds and can never take back, whatever care it takes elsewhere.

The CORE-MATH project [SZG22] removes that limit: it provides implementations of the elementary functions in the binary32 and binary64 formats that are *correctly rounded*, that is, for which k is zero because the returned value satisfies

$$\text{fl}_\circ(f(x)), \quad \circ \in \{\nabla, \Delta\},$$

the exact value of f at x rounded once in the direction in effect. The enclosure then costs nothing beyond the rounding that binary64 imposes on any result at all.

TIGHT depends on it, and not marginally. At the commit this work starts from, the 810 lines of `src/Interval.cpp` carry 123 calls into CORE-MATH, spread over 121 lines and over 37 distinct functions. Twenty-nine of those functions are called exactly twice, once per endpoint, which is what a monotone function on an interval requires. The remaining eight are called more often, and the reason is that they are not monotone over the argument they receive: `pow` twenty times, `sin` ten, `atan2` nine, `sinpi` eight, `cos` seven, `cospi` five, `hypot` and `log2p1` three each. Where a function has an interior extremum or a branch, the layer must decide which endpoints to combine before it can call, and that decision is exactly where Chapters 5 to 7 find defects.

The consequence is worth stating plainly, because it shapes Chapters 5 to 7: the transcendental functions of TIGHT do not go through the arithmetic kernel at all. They take the two endpoints as plain binary64 numbers, hand them to CORE-MATH, and rebuild an interval from the two results. The library therefore contains two arithmetics, the kernel of Section 3.4.1 for the algebraic operations and CORE-MATH for the elementary ones, which agree on the format but not necessarily on how they answer at the edges of a domain.

3.4 The starting point: NFG and TIGHT

The two pieces of software this work modifies are described here as they were found, before any change of ours. Section 3.4.1 gives NFG, the numerical kernel, with the representation it stores and the three compilation paths it offers. Section 3.4.2 gives

TIGHT, the wrapper built above it, and the interface it exposed at the outset. Section 3.4.3 names the two commits the work starts from, which is what allows every figure in Part II to be reproduced.

3.4.1 NFG: the numerical kernel

NFG [Att26], *Numbers For Geometry*, is a header-only C++ library that provides the number types on which the indirect geometric predicates of Attene [Att20] are built. Intervals are one of those types, and they are there for a purpose: a predicate first evaluates its sign in interval arithmetic, and only if the interval straddles zero, so that the sign is undecided, does it fall back on an exact and far more expensive computation. The interval layer is therefore on the fast path of every predicate, and its speed is the reason the whole construction pays off.

Three design decisions of that type govern everything that follows.

The representation is not the pair of endpoints. An interval is stored as the pair $(\text{min_low}, \text{high})$, where min_low is the *negated* lower endpoint (lines 262 and 264):

$$R(x) = (-\underline{x}, \bar{x}).$$

The reason is the rounding direction, and it rests on one identity of IEEE 754-2019,

$$\nabla(x) = -\Delta(-x), \tag{3.1}$$

which holds exactly, for every x and in every format, because negation is exact and the two directed attributes are mirror images of each other. An enclosure needs the lower endpoint rounded towards $-\infty$ and the upper one towards $+\infty$; by (3.1) the first of the two is obtained by rounding the stored, negated value towards $+\infty$. One direction therefore serves both, and the mode can be installed once for the process and never changed: `numerics.h` sets it at line 110, and TIGHT calls that function from `Interval::init` (`src/Interval.cpp`, line 67).

That is what the trick buys, and it is worth saying why it is worth buying. The obvious implementation rounds the lower endpoint towards $-\infty$ and the upper one towards $+\infty$, which means changing the rounding mode twice per operation. Writing the control register is not an ordinary instruction. It serialises the floating-point pipeline, so the work in flight has to drain before the new mode takes effect, and the two endpoints can no longer be computed by one vector instruction, since a single instruction cannot round its two lanes in two directions. A representation that needs one direction only keeps both properties. Entry M.2.20 puts a figure on what the alternative costs, in the one place where this work could not avoid switching: about thirty nanoseconds, on an operation that otherwise costs two thirds of one.

The gain is visible on the addition. The result required of $\mathbf{a} + \mathbf{b}$ is $[\nabla(\underline{a} + \underline{b}), \Delta(\bar{a} + \bar{b})]$, and in the stored form that is

$$R(\mathbf{a} + \mathbf{b}) = (\Delta((- \underline{a}) + (- \underline{b})), \Delta(\bar{a} + \bar{b})),$$

in which both components are a sum of the components already in memory, rounded in the one direction installed. A single vector instruction performs both, and the subtraction reuses it after swapping the two components of the second operand, since $R(\mathbf{a} - \mathbf{b})$ pairs $-\underline{a}$ with $\bar{b}$ and $\bar{a}$ with $-\underline{b}$. What the instruction computes is not one mathematical operation but *two roundings at once*, which is the whole point of the arrangement.

The arrangement covers addition and subtraction, and nothing else. This is the fact that governs Chapters 5 to 7, and it is worth stating before any defect is named. For the product the four candidate products must be formed and the extremes selected, so the sign of the operands decides which endpoints pair with which; for the quotient the pairing is not the one of the product, because division reverses the order in its second argument; for the square root the lower endpoint comes out rounded the wrong way and has to be corrected by hand, which the kernel does by decrementing the bit pattern of the result, that is by taking the predecessor in the ordering of Figure 2.1. Empty, the infinities and NaN are outside the scheme altogether, since they are not obtained by rounding anything. Every defect reported in Chapter 5 lies in one of these four places, and none lies in the addition.

Table 3.3 makes the convention visible, and with it two facts that no prose states as well. The empty interval and the whole line are not sentinel values of any kind: they are the pair in which both stored components are $-\infty$ and the pair in which both are $+\infty$, which under R read back as $[+\infty, -\infty]$ and $[-\infty, +\infty]$. And an interval whose endpoints arrive in the wrong order is stored without complaint, since nothing in the representation forbids it.

That last pair is worth a word, because it denotes the empty set and is not treated as Empty. In the terms of Section 2.3.1 it is a Level 3 object that represents no Level 2 datum, and the standard does not ask for it to be repaired. It leaves the result of operating on such an input implementation-defined, and requires instead that the user be given a way to have the `InvalidOperand` exception signalled when one occurs (Cl. 14.3). The kernel does neither, so the malformed pair simply propagates into whatever the arithmetic makes of it; Chapter 7 adds the switch the clause asks for, and Chapter 5 shows how the same latitude at Level 3 is what allows the empty interval to be the pair $(+\infty, -\infty)$ in the first place.

The kernel is written without branches. The operations avoid conditionals, which on the fast path of a predicate is worth more than the work a conditional would save. This is the decision that RQ3 asks about. Conformance is bought with tests, and a test is a branch: the question is therefore not only whether the operations of TIGHT can be made to answer as IEEE 1788-2015 requires, but what that costs in a kernel designed on the opposite principle.

There are three compilation paths. The header selects, through preprocessor macros, an AVX2 implementation, an SSE2 implementation, or a purely scalar one. Addition, subtraction, negation and the square root have one semantics across the three, which was

Table 3.3: How `interval_number` stores ten notable values. The two right-hand columns are the machine words; the two in the middle are what the accessors return. They were read with the probe of Appendix C, compiled along the AVX2 path of the three that Table 3.4 lists. On the level of IEEE 754-2019 the endpoints are native binary64 numbers and are conformant by construction; the pair, `Empty` and `Entire` are matter for IEEE 1788-2015 (Cl. 10.2).

Value	$[x, \bar{x}]$	accessors		stored	
		<code>inf()</code>	<code>sup()</code>	<code>min_low</code>	<code>high</code>
a number	$[5, 5]$	5	5	-5	5
a finite interval	$[-5, 77]$	-5	77	5	77
inverted input	$[7, -20]$	7	-20	-7	-20
finite to $+\infty$	$[1, +\infty]$	1	$+\infty$	-1	$+\infty$
$-\infty$ to finite	$[-\infty, 2]$	$-\infty$	2	$+\infty$	2
<code>Entire</code>	$[-\infty, +\infty]$	$-\infty$	$+\infty$	$+\infty$	$+\infty$
<code>Empty</code>	$[+\infty, -\infty]$	$+\infty$	$-\infty$	$-\infty$	$-\infty$
NaN	none	NaN	NaN	-NaN	NaN
NaN below	$[\text{NaN}, 1]$	NaN	1	-NaN	1
NaN above	$[-\infty, \text{NaN}]$	$-\infty$	NaN	$+\infty$	NaN

verified with the probe rather than assumed; the product and the quotient do not, and Table 3.4 says how they differ. The AVX2 path forms all the candidate products in wide registers and takes the extremes, so it has no case analysis and therefore no pairing to get wrong; the other two select one of nine sign configurations and compute only the two products that configuration calls for.

Table 3.4: The three compilation paths of `interval_number` and their verdict on the product and the quotient. The line numbers are those of `numerics.h` at 52a1d3d. Every measurement of Chapter 8 is reported for all three, since both a defect and a cost may appear on one path and not on the others.

Path	Flags	Lines	How $\times$ and $/$ are computed
AVX2	<code>-mavx2 -mfma</code>	1289–1345	eight products, then two maxima; no branch, and correct on finite operands
SSE2	<code>-msse2</code>	1378–1460	nine sign cases, two products each; the quotient is wrong on finite operands
scalar	<code>-mno-avx -mno-avx2 -mno-sse2</code>	1611–1691	the same nine cases, selected by shifting bits; the same wrong quotient

The nine cases are selected from the sign bits. The two paths that branch read the sign bit of the two stored components directly, without any arithmetic. Nine configurations are foreseen, and whatever falls outside them reaches the default branch, which *constructs* a NaN interval rather than computing one. The distinction is not pedantic and returns in Chapter 5: a constructed NaN performs no floating-point operation, so it raises no exception flag, whereas the value a computation would have produced raises the invalid operation that IEEE 754-2019 prescribes. **Empty**, whose two stored components are both negative, is one of the configurations that fall through in this way.

NFG is arithmetic on pairs of binary64 numbers, so it falls under IEEE 754-2019 as described in Section 2.1, and IEEE 1788-2015 makes that dependency explicit through Cl. 12.6.1: an implementation may only call itself 754-conforming if the arithmetic it rests on is. Whether NFG satisfies that is RQ2, and it is answered in Chapter 5.

3.4.2 TIGHT: the wrapper

TIGHT [SAP25] is a thin layer above that kernel. At the commit this work starts from the whole project is 11 files, of which `include/tight/Interval.hpp` is 164 lines and `src/Interval.cpp` is 810.

What the layer adds is of two kinds. It gives the elementary functions, which the kernel does not have, by calling CORE-MATH on the endpoints as described in Section 3.3; and it presents the whole as a C++ class with the arithmetic operators, so that a program written in terms of intervals reads like a program written in terms of numbers. Table 3.5 is the public surface that results, and it is the point of departure against which Chapters 5 to 7 should be read: what is listed there exists and may be wrong, and what is not listed has to be added before it can be right.

One property of that interface is not a function and has to be stated separately, because it constrains every program that uses the library. The rounding mode is global process state, installed once as Section 3.4.1 describes, so the library has to be opened and closed: **init** before the first interval is built and **deinit** after the last, with the surrounding program not free to change the mode in between. This is the price of the representation, and Section 4.3.2 has to honour it for a measurement to be measuring the library at all.

What the layer does not do, at that commit, is decide what happens when an argument leaves the domain of a function. `src/Interval.cpp` contains 23 occurrences of **return constants::nan**, all of the shape of Listing 3.2: on an argument that is partly or wholly outside the domain, the library returns a pair of NaNs.

That answer is not an interval, so it cannot be composed with another operation, and it is not what Cl. 10.4 of IEEE 1788-2015 requires, which is the enclosure of the image of the part of the argument that does lie in the domain: the very line `sqrt [-5.0,25.0] = [0.0,5.0]` that Listing 3.1 carries. This is one of the gaps that Chapters 5 to 7 close, and it is representative of the rest: the library is not careless, it simply answers a question that the standard poses differently.

Table 3.5: The public interface of TIGHT at 116cae5, the commit this work starts from. Read against Tables 9.1 and 10.1 of IEEE 1788-2015, the absences are as informative as the entries: there is no empty interval, no reverse operation, no decorated type, no textual constructor other than the one that throws, and no boolean relation of Table 10.3.

Category	Members
Constructors	<code>Interval()</code> , <code>Interval(double, double)</code> , <code>Interval(double)</code> , <code>Interval(string)</code> , <code>fromRational</code> in two forms
Endpoint accessors	<code>lower</code> , <code>upper</code>
Numeric functions	<code>width</code> , <code>middle</code> , <code>contains</code> on a number and on an interval
Arithmetic	unary and binary <code>+</code> <code>-</code> <code>*</code> <code>/</code> , and the four compound forms, with an interval and with a <code>double</code>
Comparison	<code><</code> <code><=</code> <code>></code> <code>>=</code> <code>==</code> <code>!=</code> , with an interval and with a <code>double</code>
Algebraic	<code>sqrt</code> , <code>sqr</code> , <code>inv</code> , <code>abs</code> , <code>min</code> , <code>max</code> , <code>rsqrt</code> , <code>cbrrt</code> , <code>ipow</code> , <code>pow</code> , <code>hypot</code>
Trigonometric	<code>sin</code> , <code>cos</code> , <code>tan</code> , <code>asin</code> , <code>acos</code> , <code>atan</code> and the <code>*pi</code> forms; <code>atan2</code> , which returns a pair of an interval and a boolean
Hyperbolic	<code>sinh</code> , <code>cosh</code> , <code>tanh</code> , <code>asinh</code> , <code>acosh</code> , <code>atanh</code>
Exponential and logarithmic	<code>exp</code> , <code>exp2</code> , <code>exp10</code> , <code>expm1</code> , <code>exp2m1</code> , <code>exp10m1</code> , <code>log</code> , <code>log2</code> , <code>log10</code> , <code>log1p</code> , <code>log2p1</code> , <code>log10p1</code>
Special	<code>erf</code> , <code>erfc</code>
Constants	<code>halfpi</code> , <code>pi</code> , <code>halfpi3</code> , <code>pi2</code> , <code>reals</code> , <code>nan</code>
Output	<code>operator<<</code> , and explicit casts to <code>std::string</code> and to <code>double</code>

```

tight::Interval tight::sqrt(const Interval &x) {
    if (!(x >= 0.)) return constants::nan;
    return sqrt(x.data);
}

```

Listing 3.2: The behaviour outside a domain, at the starting commit. The guard is not a check on the argument being empty or malformed: it rejects an argument that merely *overlaps* the complement of the domain, so that the square root of $[-5, 25]$ is refused although a quarter of it is perfectly well defined. Twenty-three functions of `src/Interval.cpp` are written this way.

3.4.3 The versions this work starts from

Both pieces of software are under active development, so the versions are named by commit rather than by date, and the work was carried out on forks rather than on the original repositories.

For TIGHT, the last commit of the upstream repository before the fork is 116cae5, by Federico Sichertti, of 27 April 2026; the first commit of the present work is 685a568, of 16

May 2026. The fork is `Berrutz/tight` on GitLab, and the upstream is `fsichetti/tight`. For NFG, the version analysed is `52a1d3d`, of 16 June 2026, which at that date was the head of the upstream repository `MarcoAttene/NFG` on GitHub; the fork is `Berrutz/NFG`.

One property of the starting point is worth recording, because it had to be repaired before any measurement could be trusted. At `116cae5` the build of TIGHT took its kernel from a moving target. Line 4 of `cmake/recipes/nfg.cmake` reads

```
CPMAddPackage("gh:MarcoAttene/NFG#main")
```

Listing 3.3: The dependency on the kernel as it stood at the starting commit: a branch name, not a commit.

which fetches whatever the `main` branch of NFG happens to point at at the moment of the build. Two builds a few days apart could therefore compile against different kernels, and a change in behaviour could come from either side without leaving a trace. The present work pins that dependency to a commit and makes it a build parameter, so that a measurement can be repeated against the kernel it was taken with; Section 4.5 describes the arrangement.

Part II

Contribution

Chapter 4

Design

A change to an interval library is a claim about three separate things: that it repairs what it says it repairs, that it leaves the rest of the library alone, and that it costs a stated amount of time. None of the three follows from the other two, and none of them can be settled by reading the source. This chapter describes the platform built to settle them, which is the second of the four contributions listed in Section 1.4 and the instrument every figure in Chapters 5 to 8 comes from.

The platform has two halves, answering to different masters. The first is an external corpus of conformance tests, written by other people for other libraries, which decides correctness against the standard rather than against our own expectations. The second is a benchmark that compares the conformant build of the library with the non-conformant one, operand shape by operand shape, and reports what the added checks cost. A third component, the change registry, binds a clause of the standard, a commit, a test block and a measurement into a single unit of work that a reader who was not present can check.

4.1 Requirements

The platform answers to three requirements, and they are stated before the components built to satisfy them. Section 4.1.1 sets out the three questions every entry of the registry has to answer, and why none of them can be settled by reading the source. Section 4.1.2 says why conformance is selected by a compilation flag instead of replacing the previous behaviour outright. Section 4.1.3 defines the ratio that every benchmark cell in this thesis reports, together with the three classes a change can fall into.

4.1.1 What has to be shown

Every entry of the registry answers three questions, and the platform exists to make each of them answerable without an act of faith.

Does the change repair what it claims to repair? The answer must come from a case the change did not choose for itself. A test written together with a fix tends to test the fix rather than the requirement, so the primary evidence is the external corpus of Section 4.2, whose expected values were transcribed from the standard by people who had never seen this library.

Does it break anything else? This cannot be settled by comparing two totals. Write F_{OFF} and F_{ON} for the *sets* of assertions that fail in the two builds, or before and after a change. What the platform must report is not their sizes but the two *set differences*, where $X \setminus Y$ is the set of the elements that belong to X and not to Y ,

$$R = F_{\text{OFF}} \setminus F_{\text{ON}}, \quad B = F_{\text{ON}} \setminus F_{\text{OFF}}, \quad (4.1)$$

the assertions the change repaired (R) and those it broke (B). The two totals cannot substitute for them, since $|F_{\text{ON}}| = |F_{\text{OFF}}|$ is compatible with $|R| = |B| = 5$, that is with a change that repairs five assertions and breaks five others, which a platform reporting only the count would call harmless. The two lists are therefore compared line by line with `comm`, the Unix utility that given two sorted files prints in three columns what belongs to the first alone, to the second alone and to both: exactly R , B and the assertions the change left as they were. Every entry of the registry reports both numbers, and B is empty throughout.

What does it cost? The library is the arithmetic core of a geometric predicate system, where an operation is executed billions of times and a factor of two is not an academic quantity. The benchmark of Section 4.3 answers with a measurement rather than an estimate.

4.1.2 Why conformance sits behind a compilation flag

Conformance is selected by a single macro, `TIGHT_IEEE1788_STRICT`, which is off by default. The reason is not caution about the changes: it is that both behaviours are wanted, by different callers. The library was written as the numeric core of NFG's indirect predicates, where the previous behaviour is what the surrounding geometry was built against, and the standard prescribes a different one. A flag keeps both, and keeps them in one source tree rather than in two diverging forks.

The flag governs what the wrapper does *before* it calls the kernel, and never which kernel is called: the dependency on NFG is the same commit in both builds. Figure 4.1 shows where it acts.

What that dispatch does, and in which order, is the architectural decision of the whole work, and Table 4.1 states it. The design rests on one observation about where the defects are: every non-conformant answer that the wrapper is in a position to repair arises from operands that can be *recognised before the call*, an empty operand, an infinite endpoint, a NaN, a divisor touching zero. They fall into three families, which are violations of the rule that the empty interval is absorbing, missing domain semantics, and the treatment of the whole line in the product and the quotient. The dispatch is a pipeline of four stages,

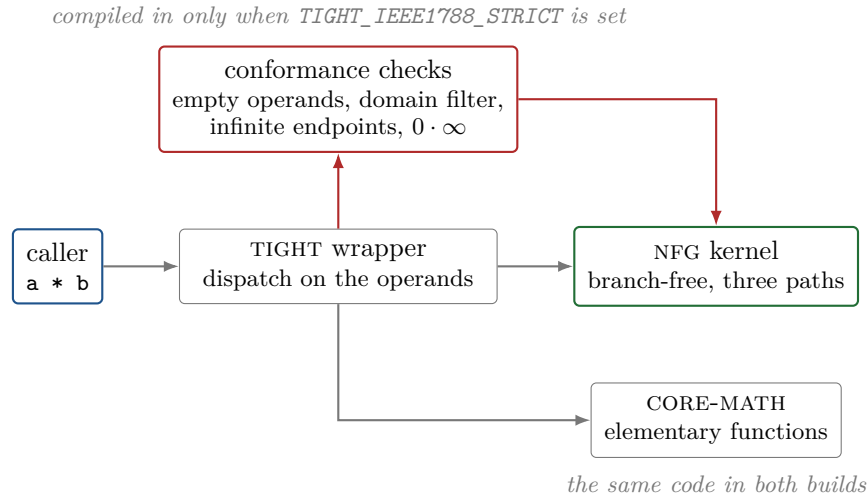


Figure 4.1: Where the conformance flag acts. The checks the standard requires are added to the wrapper, which is the layer that can still read the operands. The kernel and the elementary functions are the same code in both builds, so a difference in time between them is a difference in what the wrapper did, not in what was computed.

and it takes them in order of how cheaply they can be recognised, so that what reaches the kernel is what the kernel computes correctly, tightly and fast.

Stage	What it recognises	Clause	What leaves here
1	an operand is empty	Cl. 12.10.2	Empty
2	an endpoint is infinite, for the product and the quotient; the sum and the difference are already right	Cl. 10.4	Entire, the half-lines, $[0, 0] \cdot \text{Entire} = [0, 0]$
3	the argument leaves the domain: intersect with $[0, +\infty)$ for the square root, remove zero from the divisor	Cl. 10.4	Empty, the half-lines
4	ordinary finite operands	none	the kernel result, correct, tight to one <i>ulp</i> and fast

Table 4.1: The four stages of the conformant dispatch, all of them inside the flag. The order is not arbitrary: each stage removes the operands the next one would not know how to treat, and the fourth is the kernel untouched. The one prerequisite outside TIGHT is the repair of N.1, the defect of the kernel quotient reported in Chapter 5, without which stage four would be wrong on the two non-vector paths.

An operation that did not exist before the work started is a different case, and stays *outside* the flag. There is no previous behaviour to preserve, and putting a new name behind the macro would give the library two different public interfaces depending on how it was compiled, which is a worse outcome than a name that is always present. This rule changes what the ratio reported for such operations means, and the registry says so entry by entry: it measures not the flag but the cost the new operation *inherits* from what it calls, typically the public constructor.

4.1.3 Three classes of cost

Every benchmark cell in this thesis reports one quantity, defined once here and used without further comment from now on:

$$r = \frac{t_{\text{ON}}}{t_{\text{OFF}}}, \quad (4.2)$$

the time of the conformant build divided by the time of the other one, so that $r > 1$ is a surcharge and $r < 1$ means the conformant build is the faster of the two. Equation (8.1) gives it the shape it turns out to have.

That ratio does not always mean the same thing, and an entry failing to say which case it is in would invite the reader to compare quantities that are not comparable. Each entry therefore declares one of the three classes of Table 4.2. The declaration is not a matter of judgement: the divergence column of the entry's own table decides between the first two, and the criterion is in the table below.

4.2 External verification of conformance

Correctness is not judged against expectations of ours. It is judged against a corpus of conformance tests written by other people, for other libraries, from the text of the standard. Section 4.2.1 describes that corpus and the plugin that binds it to the interface of TIGHT, which is the one part of the arrangement this work wrote. Section 4.2.2 states what the corpus does not reach, and it is the reason the registry carries evidence of two further kinds.

4.2.1 The corpus and the plugin

Verification of correctness is delegated to ITF1788 [BFR24], the interval test framework described in Section 3.2. Its corpus consists of nineteen `.itl` files holding 261 test cases, written in a language-independent notation in which each line names an operation, its operands and the result the standard requires. The corpus is not ours and was not written for this library: eleven of its files carry the assertions of `libieeeep1788`, and the others come from `C-XSC`, `MPFI` and `fi_lib`. That outside provenance is what makes it evidence.

A generator turns those files into compilable test code once a *plugin* states how the operations of the standard are spelled in the target library. The plugin written for TIGHT

Class	Criterion	What the ratio between the two builds measures
A	divergence zero	The flag adds a test that, on ordinary operands, does not change the result. Both builds compute the same thing and return the same value, so the ratio is the cost of the added test in pure form.
B	divergence equal to the whole test set	The flag selects a different procedure, and the build without it returns a non-conformant answer. The comparison is not made at equal work, so the ratio is the difference between two procedures rather than the surcharge of a check. An entry in this class must state what the non-conformant build answers.
C	the cost is independent of the operands	It is paid on every invocation: reading a state variable, or validating a pair of endpoints in a constructor. It is not subject to branch prediction, so it appears uniformly on any data set and cannot be avoided by choosing favourable operands.

Table 4.2: The three classes of cost. The class is declared by the entry before its table is read, because the class decides in what sense the ratio is interpretable. The criterion for the first two is the divergence column described in Section 4.3.2, which counts on how many operands the two builds answer differently: a count of zero is class A and a count equal to the whole test set is class B, so those two are read off the measurement rather than asserted. Class C is recognised instead from where the cost lies, a read or a check performed on every invocation, and it shows the same divergence as class A.

is a single YAML file of 705 lines declaring 99 operations, plus a preamble of C++ that adapts the library to what the generated code expects.

Two things in that preamble are deliberately not taken from TIGHT, and Listing 4.1 shows the more instructive of them. The library inherits from the kernel an `operator==` that holds only when both operands are the same single point, so that `[1,2] == [1,2]` is false, and an `operator!=` that is disjointness rather than the negation of equality. Cl. 10.5.10 defines `equal` as equality of sets. Had the harness used the library's operator, every test on a non-degenerate interval would have failed for a reason unrelated to the property under test, and the corpus would have reported a wall of failures that said nothing. The harness therefore compares endpoints itself, and the defect of `operator==` is recorded and repaired separately, as M.2.22. The second is the rounding mode, which the harness installs itself, since the kernel assumes a global `FE_UPWARD` that no generated test would otherwise set.

An operation the library does not provide is simply left undeclared. The generator then emits a warning and *skips* that test rather than failing it, which keeps the numbering of the remaining cases stable while a plugin grows. The absent entries are therefore a measurement of what is not covered, not a defect of the plugin. The corpus grew with the library: every entry that added an operation to TIGHT also added its key to the plugin, and the denominator of the measurement moved with it. The two figures an entry reports are therefore read against the total of its own run, never against an earlier one.

```

imports: |
    #include "tight/Interval.hpp"
    #include "tight/DecoratedInterval.hpp"
    #include "tight/Reduction.hpp"

preamble: |
    // Equality is NOT taken from TIGHT. Its operator== forwards to the
    // kernel, where equality holds only when both operands are the same
    // single point, so [1,2] == [1,2] is false. Cl. 10.5.10 defines
    // equal(x,y) as equality of sets, hence the harness compares the
    // endpoints itself.
    static bool itf_equal(const I &a, const I &b) { ... }

    // The rounding mode. NFG assumes a global FE_UPWARD, installed by
    // tight::Interval::init(). BOOST provides main(), so a static object
    // runs it before any test body.

op_add*:  itf_add($ARG1, $ARG2)
op_mul*:  itf_mul($ARG1, $ARG2)
op_div*:  itf_div($ARG1, $ARG2)
op_sqrt*: itf_sqrt($ARG1)
op_pow*:  itf_pow($ARG1, $ARG2)

```

Listing 4.1: The plugin that connects the corpus to the library, abridged. An operation is a template with the arguments substituted in; the preamble holds the C++ the generated file needs. Source: `plugins/tight/arith.yaml`.

Figure 4.2 shows the whole path, from the corpus to the two lists of failures and the comparison that separates them.

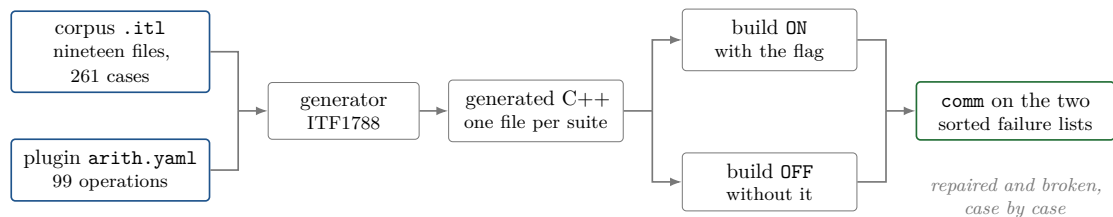


Figure 4.2: The external verification chain. The same generated sources are compiled twice, once with `TIGHT_IEEE1788_STRICT` defined and once without it, and the two lists of failing assertions are compared line by line rather than by their totals: a change that repaired five assertions and broke five others would leave the totals unmoved.

Table 4.3 states what the platform measured when it was first run and what it measures today. The first two rows are the corpus itself, which is not ours and did not change. The third is how many of its 9545 assertions were generated, that is those whose operation the plugin declares, the generator skipping the others with a warning. The last two are how many of the generated ones fail in each of the two builds.

Getting it to run at all took two changes to somebody else’s tool, and they are recorded here because they are the only ones. The framework dates from 2018 and does not start under Python 3.12: one line calls `time.clock`, removed from the language in version 3.8, and one calls `yaml.load` without a loader, which the library now refuses. Substituting `time.perf_counter` for the first and `yaml.safe_load` for the second is the whole of it. A third accommodation was needed on our side rather than theirs: the generated suites are written against the macros of Boost.Test, whose development package requires administrator rights to install, so a stand-in of about ninety lines provides the four macros the generated code actually uses, and the corpus runs without it.

What is counted	First run	Present state
.itl files	19	19
Test cases	261	261
Assertions generated	4358	9433
Failures, build ON	1084	26
Failures, build OFF	1392	n/a

Table 4.3: The corpus, at the first run and today. The number of generated assertions more than doubles because the plugin declares only the operations the library provides, and the library gained operations throughout the work. The twenty-six residual failures are classified one by one in Table 7.14.

4.2.2 What the corpus covers, and what it does not

The corpus is the strongest evidence available, and it is not sufficient. It measures only what it knows how to exercise, and four real defects were found outside it during the work. They are listed in Table 4.4 because together they are the argument for the second half of the platform, and because each of them shows a different way in which an external corpus can be blind.

Two consequences follow, and both are design decisions rather than observations. The first is that every change also carries a test block inside `main.cpp`, tagged with the label of its entry, whose expected output is transcribed in the registry: those blocks cover what the corpus does not reach, and they are the reason the sign of a zero and the subnormal exponent have a case at all. The second is that where an operation has no reference answer to compare against, verification is written as a *property* that must hold rather than as a value that must match. Thirteen elementary functions of TIGHT have no line in the corpus, the standard not listing them, and for each of them four properties are asserted on a sweep of arguments:

$$\begin{aligned}
 f(\text{Empty}) &= \text{Empty}, & f(x) &\text{ carries no NaN endpoint,} \\
 \underline{y} \leq \bar{y} &\text{ for } \mathbf{y} = f(x), & f(x) &= f(x \cap \text{Dom } f).
 \end{aligned}$$

The second and the third say that the answer is an interval at all, the first is the rule that `Empty` is absorbing, and the fourth says that the domain filter is idempotent, which is the

Defect	Why the corpus could not see it
M.2.22	The harness reimplements equality on the endpoints, precisely so that the library's own broken <code>operator==</code> does not swamp every other test. The defect is therefore invisible to the very corpus that makes the rest of the work measurable.
M.10.5	The corpus compares by value, and -0 equals $+0$. The sign of a zero endpoint, which Cl. 12.12.8 legislates, becomes observable only through the interchange encoding of Cl. 14.4.
M.2.27	The corpus has no line whose exponent is subnormal, and that is where <code>pow</code> returned an upper bound below the true value.
M.13.4	The defect was introduced by a change of ours, one that tightens: the corpus has no line for the case, and only a completeness check written as a property caught it.

Table 4.4: The four defects found outside the corpus. Three were pre-existing and one was introduced by a change of ours, which is the case a test suite is least likely to cover.

only one of the four that needs no oracle: it compares the library with itself on the already restricted argument, so it holds whatever the right value may be. The sweep found a real defect on its first run, in an operation nobody had ever exercised: the reciprocal square root took both endpoints from the lower one, so every argument wider than a point came back with its endpoints inverted. The defect was in the upstream of `TIGHT` itself and not something this work introduced, which is how the third property earned its place.

4.3 The benchmark platform

The second half of the platform measures what conformance costs, and it compares the two builds of the same source rather than two libraries. Section 4.3.1 states the three axes along which that comparison is made, since the pair of things compared is not always the same one. Section 4.3.2 gives the protocol, its five parameters and the reason the minimum is the statistic reported. Section 4.3.3 fixes the margin of uncertainty below which a difference between the two builds says nothing at all.

4.3.1 Three axes of comparison

What the benchmark compares is not always the same pair of things, and the axis is stated by the entry before its table. The three axes are those of Table 4.5. The first is the one RQ3 asks about: it is the price of conformance. The other two exist because two families of changes cannot be measured along it, decorations because they are a new type rather than a new branch, and the tightening of the reverse operations because there the flag has nothing to do and the term of comparison is the previous commit.

Axis	What is held fixed	What the ratio says
conformant against non-conformant	the source, the kernel, the operands	the price of the checks the standard requires
decorated against bare	the build, the operation	the price of carrying a decoration alongside the interval
one commit against the preceding one	the build, the operands	the price, or the saving, of a change that tightens the result

Table 4.5: The three axes of the benchmark. On the second axis the two columns are two rows of the *same* build; on the third the term of comparison is a commit, and the divergence column counts the operands on which the answer became tighter rather than the operands on which something went wrong.

4.3.2 The measurement protocol

The benchmark is a program of 3193 lines with twenty-four measurement modes, one per family of operations, kept outside the library so that no measurement changes the code it measures.

One cell of one of its tables is a minimum over five nested repetitions. The innermost is a vector of **4096 distinct operands of one shape**, for instance four thousand and ninety-six different pairs of finite intervals, never a finite one and an empty one in the same row. The timed loop runs over that vector **32 times** in a row, so that one clock reading covers 131 072 operations rather than one. That timed loop is repeated **9 times** within a single execution of the program. The program is run **5 times** for each build, the two builds alternating, so that neither of them occupies a favourable or an unfavourable moment of the machine on its own. And the whole is compiled **3 times**, with different instruction alignment options. The cell is the smallest time per operation observed across all of them.

The figures throughout this thesis were taken on one machine and one toolchain: WSL2 Ubuntu 24.04 on x86-64, g++ 13.3.0 with C++20, CMake 3.28.3. Six decisions make those figures comparable, and each was forced by an observation rather than chosen in advance.

The three paths are measured separately. NFG selects its kernel from the compiler macros, so AVX2, SSE2 and the pure scalar path are three different programs with different base costs. A figure averaging them would describe no machine.

One operand shape per row. Each measurement considers a single kind of pair, two finite intervals, or a finite one and an empty one, and never a mixture. Within one shape the branch is always predicted the same way, so the row measures one path through the dispatch; a mixed stream would measure the branch predictor instead. The shapes are named, and the names are the first column of every benchmark table in this thesis: Table 4.6 defines the nine that recur, each with an example in the representation the

library actually stores. A shape is not a set of interesting values but a class of operands that take the same branch, which is what makes one row one path.

Name	What it is	Example
finito	both endpoints finite	[1.5, 3.25]
zero	the interval of the value zero, both endpoints null	[0, 0]
zerosup	zero as lower endpoint and a positive finite as upper: it carries a null endpoint like zero without being degenerate	[0, 3.25]
azero	bounded, with zero in its interior but neither endpoint null, so that in a product it behaves as a finite shape	[-2, 2]
denom	the divisor shape: finite endpoints of the same sign, so that it does not contain zero	[3, 3.4]
semisup	finite lower endpoint, upper endpoint $+\infty$	[2, $+\infty$]
semiinf	lower endpoint $-\infty$, finite upper endpoint	$[-\infty, 7]$
entire	the whole real line	$[-\infty, +\infty]$
empty	the empty interval, whose endpoints are exchanged by Cl. 10.2 and which isEmpty recognises	$[+\infty, -\infty]$

Table 4.6: The operand shapes that recur in the benchmark tables. Shapes introduced by a single entry for its own path are defined by that entry. When the operation takes two operands the column names a pair, and the entry says which member is which.

The operands are produced once, from a fixed seed, so that the two builds see the same sequence of bits, and `Interval::init()` installs the `FE_UPWARD` rounding mode the library uses in ordinary service. Without it the timings would not be those of the code that actually runs.

The cited figure is the minimum, not the median. Noise on a shared machine is not symmetric. Another process, a change of processor frequency or a suspension imposed by the operating system can only lengthen a measurement, never shorten it, so an observation has the form

$$t_{\text{obs}} = t + \varepsilon, \quad \varepsilon \geq 0,$$

and the minimum over a set of observations is the estimator of t that the disturbance cannot inflate, while the median moves with it. The median is still computed, as an indicator: when it drifts away from the minimum the cell was disturbed, and is declared as such instead of being quietly averaged.

Every variant is compiled three times, with three instruction alignment options, and the figure is the minimum over all three. The same source compiled twice with equivalent options can produce the same instructions at different addresses, and between two such compilations differences of up to twenty per cent have been observed with nothing substantial changed. A minimum over a set of placements is a defined and reproducible quantity; the minimum of a single compilation is not.

The divergence count is taken outside the timed region. Listing 4.2 shows both halves. The timed loop brackets the whole double loop rather than one operation, since a clock reading around a one-nanosecond operation would measure the clock; the per-operand digest that counts on how many operands the two builds answer differently is written in a separate pass, so the comparison does not enter the timings.

The digest itself deserves one sentence, because a weak one would make the divergence column report zero where the builds disagree. It is a hash of the bit patterns of the two endpoints, not of their values: two answers that are the same real number stored differently, such as an endpoint that is $+0$ in one build and -0 in the other, must count as a divergence, since Table 2.2 shows they are different words and Cl. 12.12.8 legislates which of the two is required. An early version multiplied one endpoint by a constant and exclusive-ored the other, which collapses for numbers whose fraction field is zero: the infinities, the powers of two and the empty interval all shared one digest, and a whole column of divergences read zero. The digest now mixes each endpoint separately before combining them, and the affected measurements were recounted.

```
// warm-up, not timed
for (int r = 0; r < 3; ++r)
    for (size_t i = 0; i < n; ++i) sink ^= digest(fun(A[i], B[i]));

for (int round = 0; round < rounds; ++round) {
    const auto t0 = std::chrono::steady_clock::now();
    for (int k = 0; k < inner; ++k)
        for (size_t i = 0; i < n; ++i)
            sink ^= digest(fun(A[i], B[i]));
    const auto t1 = std::chrono::steady_clock::now();
    const double ns =
        std::chrono::duration<double, std::nano>(t1 - t0).count();
    printf("%s\t%s\t%s\t%s\t%s\t%d\t%.4f\t%llu\n",
        viaAttiva(), compilazione(), op, etichetta, round,
        ns / (double(n) * double(inner)), sink);
}

// The digest per operand, OUTSIDE the measurement: comparing the two files
// counts exactly on how many answers the two builds diverge.
```

Listing 4.2: The timed loop and the digest. Reading the vector and folding the digest are inside the timed region and identical in both builds: they are the common floor, which is why the faster build never drops below about six tenths of a nanosecond. Source: `bench.cpp` of the benchmark.

The divergence column deserves a sentence of its own, because it is what makes a timing readable. It does not say which of the two builds is right; it says where they differ, and correctness is established elsewhere. A zero in that column is the guarantee that both builds did the same work, and therefore that the difference in time is not a difference in what was computed. A count equal to the whole test set, conversely, is the signal that the entry belongs to class B and that its ratio compares two procedures.

4.3.3 The margin of uncertainty

A ratio close to one is not by itself evidence of anything, and the platform has to state below which difference there is nothing to read. That threshold was not assumed: it was calibrated on operations whose true ratio is known in advance.

The ten boolean predicates of G.4 lie outside the flag and construct no interval; they read the two endpoints and answer. Their assembly was compared line by line between the two builds with the addresses normalised, and it is identical in all thirty-six comparisons, which are the ten predicates plus the two loops of the benchmark itself, over the three paths. The true ratio of each of their cells is therefore exactly one, and whatever the benchmark reports instead of one is the noise of the measurement itself. Figure 4.3 shows the distribution of the fifty-four cells that result from eighteen operand shapes on those same three paths.

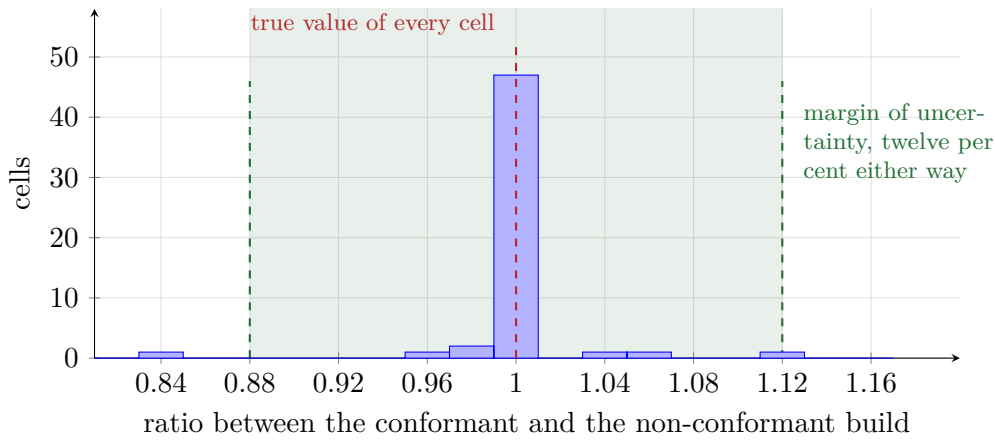


Figure 4.3: Calibration of the margin of uncertainty. The fifty-four cells of the boolean predicates of G.4, whose two builds compile to identical assembly and whose true ratio is therefore one. Forty-five of them give 1.00; the nine that deviate lie between 0.84 and 1.12. Data: `Figures/calibrazione.dat`, from the run recorded in M.4.1.

The rule that follows is used throughout Chapter 8:

$$|r - 1| < 0.12 \implies \text{the cell measures nothing,} \quad (4.3)$$

and an entry whose ratio falls inside that band reports the figure and says so.

The threshold is twelve and not sixteen per cent, although the lowest of the nine deviating cells reads 0.84, and the reason is that the nine were measured again one at a time under controlled conditions. Eight of them did not reproduce: they are the noise of a single run and carry no information about the platform. The one that did reproduce is `disjoint` on nested operands in the scalar path, which reads 1.12 in the campaign and 1.12 again in the probe, and whose assembly is identical in the two builds, so what varies is the placement of the timed loop, which the three alignments do not absorb. The margin is therefore set by the largest deviation that reproduces, not by the largest observed, and

it is symmetric because it applies to a ratio and not to a time: a disturbance can only lengthen a measurement, but it may fall on either of the two builds, so it moves the ratio in either direction.

Two further habits come from the same discipline. A table containing a row whose ratio is one *by construction*, such as a guard that answers on the empty interval without constructing anything, uses that row as an internal control: if the control lies further from one than the rows carrying the signal, the entry states that the cost it wanted to measure is below the noise of that table, instead of reading those cells as a measurement. And when a cell is disturbed, the ratio between its median and its minimum says so; such cells are measured again in a short probe under controlled conditions, and the table carries a note rather than a silently averaged number.

4.4 The change registry

The third component is documentary, and it is what turns a sequence of commits into something a reader can check. Traceability uses two stable labels. **G.N** is a *gap analysis* entry, one block of the roadmap, stated in terms of the clauses of the standard it fails to satisfy. **M.N.K** is a single atomic change to the library, where *N* is the gap it closes in whole or in part and *K* counts within it. A third label, **N.k**, is used for the defects of the kernel against IEEE 754-2019, which are not changes to TIGHT at all.

The blocks themselves are not a partition chosen for tidiness, and their order is a decision of design that Table 4.7 records together with the clauses each block answers to. Three criteria fixed that order, in this priority. First, the *logical dependencies between clauses*: the decoration system of Cl. 11 presupposes that the empty interval already exists as a first-class entity, which is Cl. 10.2, so block nine cannot precede block one. Second, *risk*: local interventions verifiable by an isolated test come before those that touch the architecture of the library. Third, *alignment with the existing interface*: the operations already present are made conformant before the missing ones are added, so that at every step the library is a library and not a building site.

The rule attached to those labels is that one **M.N.K** is exactly three things at once: one commit in the fork, one test block in `main.cpp` whose marker is the label itself, and one expected output transcribed in the registry. A change that cannot be written as those three things is not one change, and is split. A reader who wants to check any statement of Chapters 5 to 7 can therefore find the commit by its label, run the block by its marker, and compare the output with the transcription without asking anybody anything.

One block escapes that rule, and saying why is the best way to show what the rule is for. **G.0** produces no **M.0.K** at all, because it changes no algorithmic code: it installs a framework, writes a plugin, and records a baseline. There is no commit in the library to point at, no marker to grep for in `main.cpp`, and no expected output of the library to transcribe, so the triple cannot be formed and forcing one would be a fiction. What **G.0** leaves behind instead is this chapter and the numbers of Table 4.3, which every later block is measured against.

Block	What it closes	Clauses	Requires
G.0	the external verification platform	Cl. 4.3, Cl. 4.4	none
G.1	the empty interval, the constants and the basic predicates	Cl. 10.1, Cl. 10.2, Cl. 10.5.2, Cl. 10.5.10	none
G.2	the evaluation semantics of Table 9.1, the domain filter and the numeric functions	Cl. 10.4, Cl. 10.5.3, Cl. 10.5.9	G.1
G.3	conformant constructors from numbers and from text	Cl. 10.5.8, Cl. 12.12.7	G.1
G.4	the boolean functions of Tables 10.3 and 10.4	Cl. 10.5.10	G.1
G.5	intersection and convex hull	Cl. 10.5.7, Cl. 12.12.6	G.1
G.6	two-output division and the cancellative operations	Cl. 10.5.5, Cl. 10.5.6, Cl. 12.12.3, Cl. 12.12.5	G.1–G.2
G.7	the twelve reverse-mode operations	Cl. 10.5.4	G.1–G.2
G.8	the flavour-independent exceptions and their signalling	Cl. 7.5.3, Cl. 12.1.3, Cl. 12.12.7	G.1–G.3
G.9	the decorated type and the propagation of decorations	Cl. 8, Cl. 11, Cl. 12.5	G.1–G.8
G.10	the remaining operations of Cl. 12.12 and the reductions	Cl. 12.11.4, Cl. 12.12, Cl. 12.12.12	G.9
G.11	literals and conformant textual output	Cl. 9.7, Cl. 12.11, Cl. 13	G.1–G.9
G.12	the standard names, the claim and what it drags with it	Cl. 4.3, Cl. 4.4, Cl. 12.2, Cl. 12.10.3, Cl. 14.3, Cl. 14.4	G.1–G.11
G.13	the tightness of the reverse operations	Cl. 12.10.2, as a recommendation and not an obligation	G.7

Table 4.7: The fourteen blocks of the roadmap, the clauses each one answers to, and the blocks it presupposes. The correspondence between a block and a gap analysis entry is one to one, and blocks are not merged into broader categories. Table 5.2 gives the same blocks with the number of atomic changes each one produced.

An entry is made of ten paragraphs, appearing in a fixed order and each carrying its own colour, so that the role of a paragraph is recognised before it is read. Figure 4.4 lists them. An entry may lack some of them, and there are two recurring reasons: a change that adds a new name has no previous behaviour to show, and an entry that is not measured has nothing to observe. The paragraphs that remain keep their colour and their position.

Two documents are kept, and the division between them is deliberate. The gap analysis is forward-looking: it states, clause by clause, what the library fails to satisfy and what would have to change. The registry is backward-looking: it states what was done, with the commit and the evidence. Keeping them apart is what makes it possible to say at any moment which part of the roadmap is closed, and it is the reason the audit of Section 7.5 could be carried out against the 192 sentences of the standard containing *shall*, one at a time, instead of against a list of blocks somebody believed to be finished.

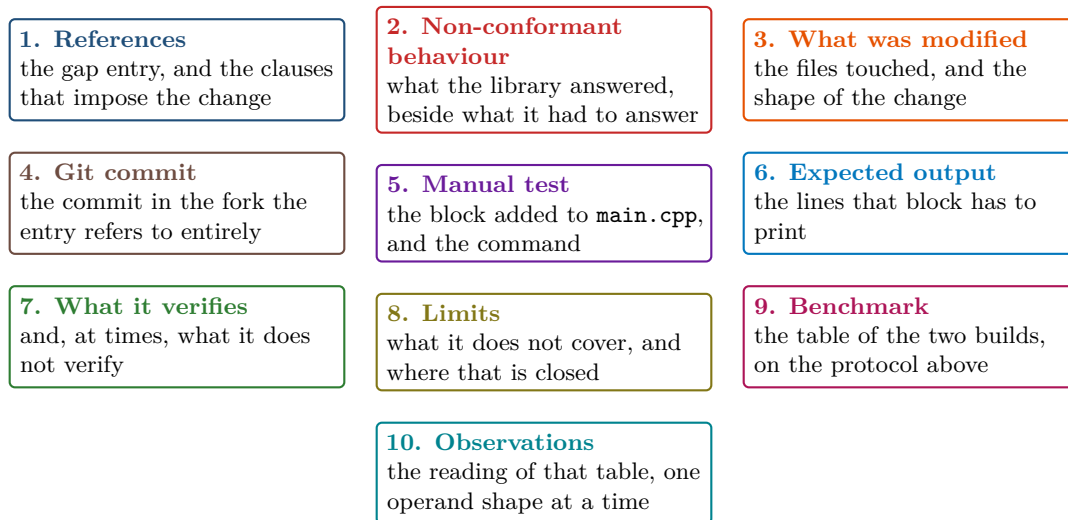


Figure 4.4: The anatomy of a registry entry. The ten paragraphs always appear in this order, and the colour of a heading names its role; an entry that lacks one of them keeps the others where they are. The registry is the backward-looking document; the gap analysis, which states what has to change and why, is the forward-looking one.

4.5 Reproducibility

A measurement that cannot be repeated against the same dependencies is an anecdote. Three arrangements and one rule make the runs of this work repeatable, and the first of the three was itself one of the changes.

The kernel is pinned. The moving dependency of Section 3.4.3 is now pinned to a commit and exposed as a build parameter, `-DNFG_GIT_TAG`, which is also how the defects N.1 and N.9 of the kernel, both reported in Chapter 5, are reproduced through the library.

The build is described, not assumed. Both halves of the platform are driven by scripts kept in the repository, which locate the root from their own position and stop with a message naming what is missing rather than failing half way. Listing 4.3 is the one that runs the corpus: it generates the sources, builds both compilations, and closes with the comparison that matters.

A figure is never carried forward. Absolute times depend on the instantaneous frequency of the machine, so a figure taken in one session is never set beside a figure taken in another; ratios are far more stable, because the two builds are interleaved within the same run. When a change made a previous measurement obsolete, that measurement was taken again rather than carried forward, and the registry says which entries were measured again and why.

Two adaptations to the build system were needed before any of this could run, and both lie outside the library code. In `cmake/recipes/nfg.cmake` the dependency is fetched with `DOWNLOAD_ONLY YES`, because NFG defines an executable target whose name collides with the interface library of the same name; and `CMakeLists.txt` gained the

```
controlla_prerequisiti
genera_corpus

misura "$REPO/tight" on "-DTIGHT_IEEE1788_STRICT"
misura "$REPO/tight" off ""

printf "repaired (fail only without): "
comm -13 <(sort "$HOME/itf_out_on/fails.txt") \
    <(sort "$HOME/itf_out_off/fails.txt") | wc -l
printf "broken (fail only with) : "
comm -23 <(sort "$HOME/itf_out_on/fails.txt") \
    <(sort "$HOME/itf_out_off/fails.txt") | wc -l
```

Listing 4.3: The script that runs the external corpus. The last two commands are the point of the whole arrangement: the effect of the flag is reported case by case, and not as a difference of totals. Source: `scripts/run_corpus.sh`.

option `TIGHT_BUILD_TESTS`, which allows the library and its executable to be built in the absence of `libmpfr-dev`, alongside `TIGHT_IEEE1788_STRICT` itself. Both appear in the build commands quoted throughout Chapters 5 to 7.

Chapter 5

The Kernel, and the Semantics of Evaluation

This chapter and the two that follow report what was analysed and what was changed, and together they answer **RQ2**: conformance could not be obtained by acting on the wrapper alone.

This one begins underneath. NFG is the kernel TIGHT is built on, and the question is whether the arithmetic below the interval layer is faithful to IEEE 754-2019: it is not, in nine identified respects, two of which are beyond the reach of any layer above them, and four of the repairs have been accepted into the official repository. The chapter then turns to the wrapper, presents the thirteen blocks of the roadmap and the form in which each change is reported, and carries the first two of those blocks: the constants, which had to come first because nothing compiled without them, and the semantics of evaluation, which holds twenty-seven of the seventy-one changes and is the largest block of the work. Chapter 6 carries the constructors and the operations that had to be added, and Chapter 7 what the standard requires beyond arithmetic.

The order is not chronological but causal. A defect of the kernel cannot be repaired by the layer above it whenever the layer above cannot tell, from the operands alone, that something is about to go wrong; the criterion that decides this is stated first, because it is what assigns every one of the defects found to one side or the other.

5.1 Where a defect belongs: kernel or wrapper

TIGHT is a wrapper: it inspects the operands, decides which path to take and calls the kernel. A wrapper can therefore repair exactly what it *recognises from the operands before calling*, which is to say the empty interval, infinite endpoints, a NaN, a divisor that touches zero, an argument that leaves the domain of an elementary function. All of these are recognisable in advance, and all of them became changes to TIGHT.

If instead the result is wrong on *ordinary finite operands*, the wrapper would have to

know the right answer already in order to notice, and a layer that knows the right answer does not need the layer beneath it. A defect of that kind is repaired at the source. This is the criterion, and it made two of the changes unavoidable rather than optional: the division, `N.1`, is wrong on two finite positive intervals, which is the most ordinary input there is, and the conversion out of exact arithmetic, `N.9`, delivers a wrong value from an ordinary rational.

Understanding what the kernel does with those operands requires the design decision of Section 3.4.1, which Figure 5.1 restates in the form the analysis uses. An `interval_number` is a pair of doubles, and the first of the two is the lower endpoint *negated*. The rounding mode of the processor is set once, to `FE_UPWARD`, and never changed afterwards.

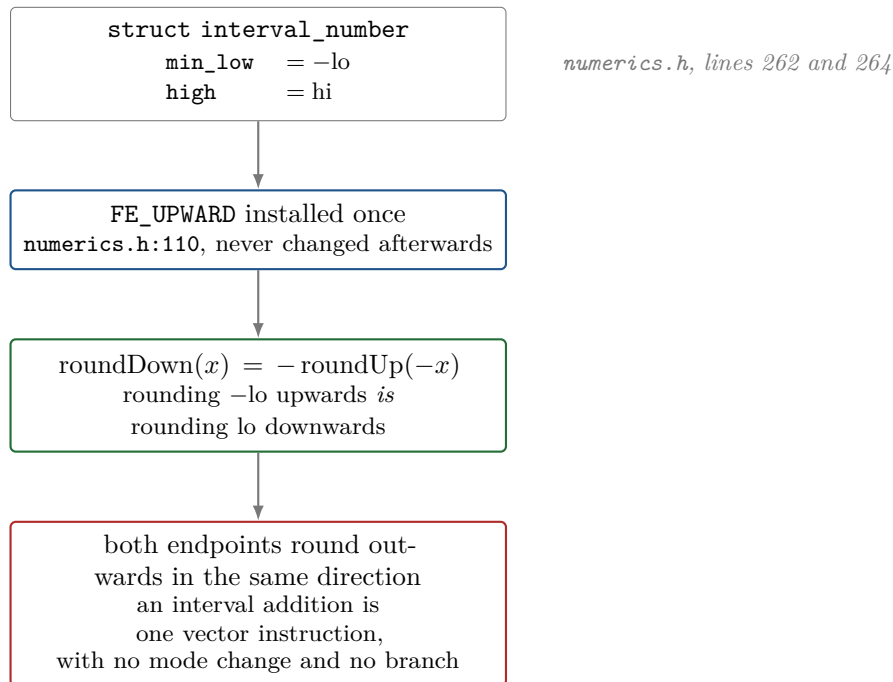


Figure 5.1: The representation of NFG, and why it is fast. Storing the lower endpoint negated makes outward rounding a consequence of a single rounding mode, so the two endpoints of a sum are computed by one instruction on a vector of two doubles. The design is deliberate, and it is also what makes the correctness of each operation a question about *which endpoints are paired with which*, rather than about rounding.

That last observation is the one that matters for the analysis. Since rounding is settled once and for all by the representation, an error in one of these operations is not an error of rounding: it is an error in the choice of which endpoints to combine. This is interval logic, governed by IEEE 1788-2015, rather than scalar arithmetic, governed by IEEE 754-2019. It is also why the two standards turned out to be complementary rather than alternative, which is the first half of the answer to **RQ1**.

5.2 The defects of NFG against IEEE 754-2019

The analysis was carried out on the version of NFG described in Section 3.4.1, reading the source and confirming every statement with a self-checking probe compiled along all three code paths and at two optimisation levels. Nine entries came out of it, listed in Table 5.1. The identifier is also the marker in the probe, so from an entry one command reaches the experimental evidence that produced it.

Entry	Defect	Repair site	Clause
N.1	division wrong on finite operands in the two non-AVX2 paths	NFG only	Cl. 10.4
N.2	product with $0 \cdot \infty$: an invalid enclosure delivered in silence	NFG, or the TIGHT dispatch	Cl. 10.4
N.3	<code>pow3</code> violates containment on finite operands, by one to two <i>ulp</i>	NFG, unused API	Cl. 10.4
N.4	<code>fmsub</code> computes $ab + 1/c$ instead of $ab - c$	NFG, unused API	754 Cl. 5.4.1
N.5	<code>isNAN()</code> inspects only the upper endpoint	NFG, one line	754 Cl. 6.2
N.6	branches that depend on the stored sign of a zero	NFG; kept away from TIGHT by the dispatch	754 Cl. 5.11
N.7	spurious <code>FE_INVALID</code> on quiet NaN in the comparisons	NFG	754 Cl. 6.2
N.8	spurious <code>FE_INVALID</code> from <code>MAXPD</code> and <code>MINPD</code>	NFG, by design	754 Cl. 6.2
N.9	<code>bigrational::get_d()</code> loses the sign on overflow and returns $-\infty$ on underflow	NFG, two lines	754 Cl. 7.4, 754 Cl. 7.5

Table 5.1: The nine defects of the kernel against IEEE 754-2019. The numbering is the order in which the entries were opened and, up to N.8, also the order of severity. Five of them are objective defects with small local repairs; four are design decisions of a branch-free kernel, legitimate for the geometric use it was written for. Two of those four are produced only by operands the dispatch recognises before calling, so through TIGHT they do not occur at all; the other two raise exception flags that an interval layer does not expose.

The division, N.1. It is the most serious of the nine, and the only one that reaches ordinary code through TIGHT. The operator was written by cloning the product and replacing the multiplications with divisions, *without changing which endpoints are paired*; even the comments of the product survived. The mathematics does not survive the substitution.

Which endpoints are paired is decided by the direction in which the operation is monotone in each argument. For two positive intervals the product is increasing in both, so

$$\mathbf{a} \cdot \mathbf{b} = [a_{\text{lo}}b_{\text{lo}}, a_{\text{hi}}b_{\text{hi}}], \quad (5.1)$$

while the quotient is increasing in the numerator and *decreasing* in the denominator, so the pairings cross:

$$\mathbf{a}/\mathbf{b} = [a_{\text{lo}}/b_{\text{hi}}, a_{\text{hi}}/b_{\text{lo}}]. \quad (5.2)$$

Substituting the operator without exchanging b_{lo} and b_{hi} leaves (5.1) in place of (5.2), and the answer is either too narrow or inverted. Listing 5.1 shows what that costs.

case	SSE2		correct
[8,15] / [4,5]	[2, 3]	(!)	[1.6, 3.75]
[6,6] / [2,4]	[3, 1.5]	(!)	[1.5, 3]
[8,15] / [-5,-4]	[-0.33, -0.5]	(!)	[-3.75, -1.6]

Listing 5.1: The division on three finite cases, two with a positive divisor and one with a negative one. The middle column is what the SSE2 path answered; the scalar one reuses the same pairings and violates containment on the same cases. In the first case the result does not contain the true quotient, since $8.5/5 = 1.7$ lies outside it, and in the other two the endpoints come out inverted, which is not an interval at all. The AVX2 path forms all eight quotients and takes the extremes, and is immune by construction.

The repair pairs the endpoints as division requires, and leaves the AVX2 path untouched. It was verified on a matrix of three code paths, two optimisation levels and six sign configurations: forty-two containment checks, all passing after the change and five failing before it.

The exit door from exact arithmetic, N.9. The entry was opened much later than the others, while making the text constructors of TIGHT conformant, and it is the second of the two defects that deliver a wrong value from ordinary data. Listing 5.2 is the whole of it: two lines that decide what happens when an exact rational is converted to a double it does not fit into.

```
if      (E > 1023) return INFINITY;    // sign lost
else if (E < -1022) return -INFINITY;  // and this is not a zero
```

Listing 5.2: Two lines that break three rules of IEEE 754-2019. The sign is lost on overflow, so a large negative rational returns $+\infty$; underflow returns $-\infty$ for both signs, where the standard asks for a zero of the right sign; and the threshold sits at the beginning of the subnormal range instead of its end, so subnormal results are never produced at all. The sibling function on the neighbouring type already did the right thing, which is why the repair is an alignment rather than a new convention.

The repair rewrites those two decisions as the standard states them, and takes its form from the sibling function on the neighbouring type, which already did the right thing. On overflow the sign of the rational is carried into the result, so a large negative value returns $-\infty$ and not $+\infty$, which is what 754 Cl. 7.4 requires of a directed conversion. On underflow the result is a zero of the sign of the value, not an infinity of the wrong one. And the threshold moves to the *end* of the subnormal range instead of its beginning, so a value that a subnormal can represent is delivered as that subnormal, which is the second half of 754 Cl. 7.5: an implementation shall always deliver a rounded result, that is a zero, a subnormal or $\pm b^{e_{\min}}$, and never skip the subnormals altogether.

The probe measures the three rules on fifteen cases, with expected values that are powers of two, so that no rounding decision is in question: six passed before the change and fifteen after, identically along the three code paths and at both optimisation levels. One line of the corpus moved with it, the last literal whose interval `textToInterval` did not enclose, which is how the entry came to be opened so late.

Two defects in an exposed but unused API, N.3 and N.4. Neither is reached through TIGHT, and both are one expression long.

The cube of an interval was computed as $x \cdot x \cdot x$ on the stored lanes, signs included. With FE_UPWARD installed, $x \cdot x$ is an *overestimate* of the square; multiplying an overestimate by a factor of the opposite sign gives an *underestimate* of the product, so the lower endpoint of the cube came out one to two *ulp above* the true cube and containment failed. The repair is the identity

$$v^3 = (v \cdot |v|) \cdot |v|, \quad (5.3)$$

in that association and no other: written $v \cdot (|v| \cdot |v|)$ it reproduces the defect, because the inner product is again rounded away from the sign it will be multiplied by. The form is branch-free, which is what the kernel requires of it.

The fused operation $ab - c$ was written as $\text{fma}(a, b, c^{-1})$, where the inverse meant is the *additive* one, so the function computed

$$ab + \frac{1}{c} \quad \text{instead of} \quad ab - c = \text{fma}(a, b, -c), \quad (5.4)$$

which 754 Cl. 5.4.1 states in the second form. Beyond the wrong value it raised FE_DIVBYZERO for a c touching zero and FE_INVALID for a c straddling it, in an operation that contains no division at all. The repair is the unary minus, which is the exact swap of the two lanes and is the quiet negation of 754 Cl. 5.5.1.

The test for a corrupt interval, N.5. The predicate that reports whether an interval carries a NaN inspected the upper endpoint alone, so a NaN in the stored lower component passed unnoticed and the object was declared sound while it was not. The case is not contrived. With the convention of Table 3.3, `Empty` is the pair whose two stored components are both $-\infty$, so in the sum `Empty` + $[-\infty, 7]$ the two lower components added are $-\infty$ and $+\infty$, and their sum is a NaN that the upper component never sees; the

subtraction `Empty` – $[7, +\infty]$ produces it in the same way. Ordinary operations therefore deliver an interval that the predicate calls sound.

The repair is one disjunction, which tests the second component as well. 754 Cl. 6.2 propagates a `NaN` through every operation so that corrupt data stays recognisable downstream, and a predicate that reads half of the datum defeats that intent. The line sits outside the `#ifdef` that selects the code path, so one change covers all three; and the comparison used is `!=`, which 754 Cl. 5.11 defines as a quiet predicate, so the repair removes a silent failure without adding the spurious flags of N.7. The probe checks nine cases in six configurations: three of the nine failed before the change in every one of them, and none after.

The four that are not defects. N.2, N.6, N.7 and N.8 are design decisions of a branch-free kernel. The vector product forms eight products and takes their extremes, which is what makes it fast and also what makes it deliver $0 \cdot \infty$ as a silent `NaN`; the comparisons in the pre-checks raise `FE_INVALID` on a quiet `NaN`, which IEEE 754-2019 reserves for signalling ones. Repairing them inside the kernel would cost the performance the kernel exists for, so they are reported rather than repaired. What becomes of them through `TIGHT` is not the same for all four.

N.2 and N.6 are *prevented*. Both arise only from operands the wrapper can recognise before it calls: a zero endpoint in one operand together with an unbounded endpoint in the other for the first, a divisor touching zero for the second, which are exactly the conditions under which M.2.2 and M.2.4 take the conformant path instead of the kernel. Through `TIGHT` those cases therefore do not occur, and the entries remain open only for a caller who uses `NFG` on its own.

N.7 and N.8 are neither prevented nor made less harmful: the flags are still raised in the processor. What makes them of no consequence here is that IEEE 1788-2015 rejected global flags in favour of decorations, so no interval operation of `TIGHT` reports an exception flag of IEEE 754-2019 in the first place, and nothing in the layer above can observe them.

5.3 What was accepted upstream

Four of the nine entries are in the official repository of `NFG`. N.1 was proposed as a single-file pull request and merged by the author in under seventeen hours without a request for changes; N.3, N.4 and N.5 followed in a second one. The branch carrying N.9 is published on the fork and its request has not been opened, by explicit decision: the entry was written after the package that had been sent to the maintainer, and opening it would have put in front of him a defect he had not been told about.

The consequence for this work is a matter of provenance rather than of pride. The repairs of N.1, N.3, N.4 and N.5 are no longer something a reader must take on trust from a fork: they are in the code everybody clones, and the statement that the kernel had those defects is checkable against a history that is not ours.

5.4 TIGHT against IEEE 1788-2015: the thirteen blocks

The work on the library follows the roadmap agreed with the advisor, which the supervisor asked not to be merged into broader categories: thirteen blocks, each one a group of clauses, plus a block zero that builds the verification platform of Chapter 4 and produces no change to the code. Table 5.2 lists them with the number of atomic changes each one produced.

Block	What it closes	Entries
G.0	the external verification platform	none
G.1	the empty interval and the constants	1
G.2	the semantics of evaluation: special values, domain filter, infinities, and the rest of Table 9.1	27
G.3	conformant constructors from numbers and from text	5
G.4	the boolean functions of Tables 10.3 and 10.4	3
G.5	intersection and convex hull	2
G.6	two-output division and the cancellative operations	2
G.7	the twelve reverse-mode operations of Table 10.1	7
G.8	the exception mechanism	2
G.9	the decorations, and their propagation	5
G.10	the remaining operations of Cl. 12.12	6
G.11	textual output	2
G.12	the standard names, the interchange encoding and the claim	5
G.13	the tightness of the reverse operations	4
total		71

Table 5.2: The thirteen blocks of the roadmap, and the seventy-one atomic changes that closed them. G.2 is by far the largest because it holds the arithmetic itself; G.13 is the only block that closes no obligation of the standard, and is discussed separately in Section 7.4.

Before the blocks are described one at a time, Figure 5.2 shows what they did to the external corpus. Two curves are needed rather than one, because the denominator moves: the plugin declares only the operations the library provides, so a block that adds operations also enlarges the corpus it is measured against. Reading the failures alone would suggest that the work went backwards twice, when in fact both rises are blocks that brought in hundreds of new assertions.

How an entry is presented. Each block is described first as a whole, and then entry by entry. An entry is the atomic unit of this work, one clause answered by one commit. It is given a card whose head names the label, the title, the clause and the commit, so that a reader can go from the claim to the code without an intermediary, and whose body has four fields, always in this order.

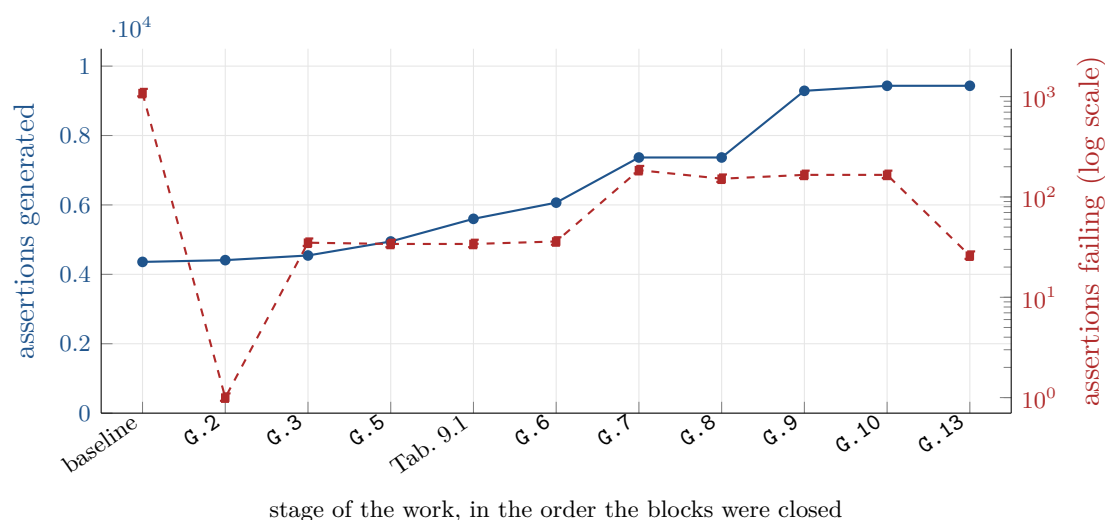


Figure 5.2: The external corpus along the work: assertions generated on the left axis, assertions still failing in the conformant build on the right, which is logarithmic. The two rises in the failures are **G.7** and **G.9**, the blocks that added the reverse operations and the decorated type: each brought in several hundred assertions that had never been exercised before, and neither broke a single line that passed before, which was checked case by case. Data: `Figures/corpus.dat`.

Defect. What the library did before, and why that was not conformant.

Remedy. What it does now.

Evidence. What was run to believe it.

Cost. The measurement, or the reason there is none.

Behind every entry there are two tables in the registry, the answers of its test block before and after the change and the measurement of what the change costs, and a card that reproduced both would run to a page. What the cards carry is therefore the reading of those tables and not the tables themselves. The first of them, **M.2.1**, is given with both, so that the figures the other cards report in prose can be read against one worked example; and the few later entries whose defect is a *wrong value* keep the table of the two answers, since seeing them side by side is the shortest way to say what was wrong.

What the cards leave out is the reasoning that produced them, the alternatives tried and rejected and the assembly that explained a ratio. That is in the registry, one or two pages per entry, where the same entry carries the ten paragraphs of Figure 4.4; each card is an entrance to its own page there rather than a replacement for it.

5.5 The empty interval and the constants

The first block is the smallest and had to come first, because nothing else compiles without it. Cl. 10.5.2 requires the two constants **Empty** and **Entire**, and Cl. 10.5.10 the predicates that recognise them; **TIGHT** had neither. The empty interval also has a conventional representation, the pair with its endpoints exchanged, which the assertion in the constructor rejected as malformed: the assertion had to be widened to admit exactly that pair.

The measurement that justifies the block is not a count of repaired assertions, since the corpus could not run at all before it: it is that the nineteen generated files went from zero compiled to nineteen compiled. That is also the argument with which the change was proposed upstream, and it is a better one than quoting the clause, because it says what the maintainer gains.

M.1.1.1 The constants **empty** and **entire**

Cl. 10.5.2, Cl. 10.2

7c6b978

Defect. NFG has no notion of an empty interval, and **TIGHT** exposed no constant for one. Cl. 10.5.2 requires an implementation to provide both **Empty** and **Entire**, and Cl. 10.2 fixes the convention that makes the first representable, $\inf \emptyset = +\infty$ and $\sup \emptyset = -\infty$. The constructor of the upstream asserted $\bar{x} \geq \underline{x}$, so that pair could not be built at all in a debug build.

Remedy. The assertion is relaxed to admit the reversed pair, and **tight::constants** gains **entire**, which is $[-\infty, +\infty]$ and a synonym of the existing **reals**, and **empty**, which is $[+\infty, -\infty]$.

Evidence. A test block in **main.cpp** prints the two constants and their four endpoints. The corpus cannot see the entry on its own, but it is the entry that made the corpus compile at all.

Cost. Not measured, and there is nothing to measure: constants are objects and not operations, and an assertion does not exist under **NDEBUG**, which is how every build of the benchmark is compiled. The cost of the two constants appears where they are used, in M.2.1 and M.2.19.

5.6 The semantics of evaluation

G.2 is the arithmetic itself, and it holds twenty-seven of the seventy-one entries. What it repairs falls into four families, and the progression of Figure 5.2 between the first and the second point is almost entirely its work: failures went from over a thousand to one.

All four descend from one obligation. Cl. 10.4 fixes what an operation of the library is allowed to return: given a point function f of n arguments, the result on $\mathbf{x}_1, \dots, \mathbf{x}_n$ must contain the image of f over the part of the box that lies inside the domain of f ,

$$f(\mathbf{x}_1, \dots, \mathbf{x}_n) \supseteq \{ f(x_1, \dots, x_n) \mid x_i \in \mathbf{x}_i, (x_1, \dots, x_n) \in \text{Dom } f \}, \quad (5.5)$$

and Cl. 12.10.2 adds, for the basic operations, that the containment be an equality up to representability, that is *tightest*. Read together, the two say that an answer may be too wide but never too narrow, and that for the operations of Table 9.1 it may not even be too wide. Every one of the four families below is a way in which the library failed one of the two.

The empty interval must be absorbing. Cl. 12.10.2 states that if any input is `Empty`, the result shall be `Empty`. The library propagated the conventional pair by accident in the addition, because the arithmetic happens to carry it, and produced a `NaN` in the product wherever an infinite endpoint met it. The repair is a guard in twenty-one places, and it is the clearest example of a class A cost: both builds compute the same thing on ordinary operands and differ by one test.

The pairings, the special values and the division. For a product of intervals the image is attained at the corners of the box, so the answer is the smallest and the largest of four products,

$$\mathbf{a} \cdot \mathbf{b} = [\min P, \max P], \quad P = \{ a_{lo}b_{lo}, a_{lo}b_{hi}, a_{hi}b_{lo}, a_{hi}b_{hi} \}, \quad (5.6)$$

which is exactly what the kernel computes. The rule that is missing from it is the one for $0 \cdot \infty$: a member of P is that product precisely when one factor is zero and the other infinite, so the wrapper can recognise the case before calling, with

$$(0 \in \{a_{lo}, a_{hi}\} \wedge \pm\infty \in \{b_{lo}, b_{hi}\}) \vee (0 \in \{b_{lo}, b_{hi}\} \wedge \pm\infty \in \{a_{lo}, a_{hi}\}), \quad (5.7)$$

and take the conformant path only there. This is N.2 prevented in the wrapper rather than repaired in the kernel, and the two comparisons per endpoint that (5.7) costs are the surcharge Chapter 8 measures for that entry.

The division was rewritten for a different reason. Its first version, M.2.3, computed $\mathbf{a} \times (1/\mathbf{b})$, which contains the true quotient and is therefore *valid*, but rounds twice: once in the reciprocal and once in the product. Dividing the endpoints directly,

$$\mathbf{a}/\mathbf{b} = [\min Q, \max Q], \quad Q = \{ a_i/b_j \}, \quad 0 \notin \mathbf{b}, \quad (5.8)$$

rounds once and is *tightest*, which Cl. 12.10.2 requires of a basic operation. With M.2.4 an entry that had closed one clause was thus reopened by another, and this is recorded in the registry rather than quietly corrected.

The domain filter. An elementary function evaluated outside its domain returned a `NaN`, where (5.5) asks for the enclosure of the image over the part of the argument that lies *inside* the domain. For a unary f the repair is one intersection performed before the call,

$$f(\mathbf{x}) = f(\mathbf{x} \cap \text{Dom } f), \quad f(\mathbf{x}) = \text{Empty} \text{ when } \mathbf{x} \cap \text{Dom } f = \emptyset, \quad (5.9)$$

so that the square root of $[-1, 4]$ is $[0, 2]$ and not a `NaN`. Fourteen functions needed it, and it is also exposed as a public operation, since the research group had asked for it as a reusable facility. This one change repaired seventy-nine assertions of the corpus.

The same repair does not transfer to `pow` unchanged, and the reason is worth a line because it recurs: the domain of the power is not a box, but

$$\text{Dom}(\cdot) = \{ (x, y) \mid x > 0 \} \cup \{ (0, y) \mid y > 0 \}, \quad (5.10)$$

so intersecting each operand with its own projection would admit the point $(0, 0)$, which the domain excludes. The base alone is filtered and the corner $x = 0$ is treated separately, which is why M.2.10 is one entry and not an instance of the previous one. Its share of the corpus is four hundred and ninety-nine assertions, the largest of any single change in this work.

The elementary functions at their bounds. The remaining entries are a sequence of one-defect-one-entry repairs on the exponentials, the powers and the tangent, and they share a cause worth stating once: the library keeps `FE_UPWARD` installed globally, and a library of correctly rounded functions is not for that reason a library that respects the rounding mode. Four separate entries came from that single misunderstanding, in `exp2` and `exp10` at underflow, in `atan2`, in `nearbyint`, and in `pow` with a subnormal exponent. The last of them, M.2.27, is one of the four defects the corpus could not see, listed in Table 4.4.

The block also closes Table 9.1 itself, and the way that came about is instructive enough to be reported. The table lists thirty-nine required operations; the roadmap described block (2) in terms of the operations that *existed and were wrong*, so the ones that were simply *missing* had no home in any block. An audit of the API against the table found seven of them absent, `fma` and the six integer functions, and they were added here, as M.2.25 and M.2.26, because that is where the clause sends them. The rule that came out of it is used for the rest of the work: the label follows the clause, not the description that had been given of it.

5.6.1 Empty, the pairings and the domain filter

The first nine entries are the arithmetic proper: the empty interval made absorbing, first in the four operators and then outside them; the two pairings the kernel got wrong; the domain filter in its two forms, inside the elementary functions and as an operation of its own; the three numeric functions of Table 10.2 that were missing; and the tangent at its poles.

M.2.1 Empty absorbing in the arithmetic operations

Cl. 12.10.2

dd7cd72, a2d3b9a

Defect. Cl. 12.10.2 requires that if any input of a basic operation is `Empty` the result be `Empty`. NFG has no notion of the empty interval, its operators being pure vector arithmetic, so TIGHT returned whatever that arithmetic produced. Addition and subtraction propagated it by accident, the representation of `Empty` surviving a sum with a finite interval, and not even always: with an unbounded operand the sum of opposite infinities gives a NaN.

Remedy. An `isEmpty` guard in front of twenty-one points of the wrapper: the unary minus, the four binary operators `+`, `-`, `·` and `/` in three overloads each, between two intervals and in the two mixed forms with a number, and the eight compound assignments `+=`, `-=`, `*=` and `/=` in their interval and their number form. The kernel is not touched, and `TIGHT_IEEE1788_STRICT` encloses the whole addition.

Evidence. Table 5.3 gives the six operations of the test block in both builds. Without the flag the product and the quotient answer `Entire` and `[2,3]/Empty` answers `[0,0]`, which are well formed intervals and are not `Empty`; with the flag every one of the six answers `Empty`. The entry is also the first demonstration of the criterion of Section 5.1: a defect of NFG on special values is repairable in the wrapper alone, which is what separates it from N.1 on finite operands.

Operation	Flag off	Flag on
<code>empty + [2,3]</code>	<code>[inf,-inf]</code> , <code>Empty</code> by accident	<code>Empty</code>
<code>empty - [2,3]</code>	<code>[inf,-inf]</code> , <code>Empty</code> by accident	<code>Empty</code>
<code>empty * [2,3]</code>	<code>[-inf,inf]</code> , <code>Entire</code>	<code>Empty</code>
<code>empty / [2,3]</code>	<code>[-inf,inf]</code> , <code>Entire</code>	<code>Empty</code>
<code>[2,3] / empty</code>	<code>[0,0]</code>	<code>Empty</code>
<code>empty * empty</code>	<code>[-inf,inf]</code> , <code>Entire</code>	<code>Empty</code>

Table 5.3: The empty interval in the arithmetic operations, from the test block of M.2.1, in the two builds of the same source. The two rows that answer `Empty` without the flag do so by accident, the representation of the empty interval surviving a sum; the other four return a well formed interval that is not `Empty`, which Cl. 12.10.2 forbids.

Cost. Class A, and the cleanest measurement of the campaign, since addition and subtraction are the two operations in which the guard is the only thing the flag adds. Table 5.4 gives the ten shapes on the AVX2 path. The addition goes from 0.931 to 1.190 nanoseconds, a ratio of 1.28; the same ratio is 1.23 on SSE2 and 1.57 on the scalar path. It does not depend on the operands: the six shapes with no empty operand agree to the third digit, which is what one expects of a test that is always executed and of a branch that is always predicted. Of the four that carry one, the three in which the empty interval comes first fall below unity, 0.94 and 0.95, the guard recognising it and the kernel never being called; `(finito, empty)` stays at 1.26, because the disjunction short-circuits and there two tests are evaluated instead of one. The last two rows are the only ones in which the two builds answer differently, 4096 out of 4096, since without the flag the sum of opposite infinities gives a NaN.

M.2.2 Multiplication with unbounded endpoints

Cl. 10.4, Cl. 12.10.2

e0afc8c, e79c0af

Defect. The kernel forms the product by combining the endpoint configurations with `max` and `min`, whose treatment of a NaN is asymmetric in the order of the operands. The NaN that `0 · ∞` produces is therefore absorbed or propagated according to where it sits and to the compilation path. On AVX2 four combinations give three distinct outcomes, and `[0, 1] · Entire` returns `[-∞, -∞]`, a well formed interval that does not contain the product.

Shape	addition			subtraction			div.
	off	on	ratio	off	on	ratio	
(finito, finito)	0.931	1.190	1.28	1.000	1.539	1.54	0
(finito, zero)	0.930	1.190	1.28	0.996	1.539	1.55	0
(finito, entire)	0.931	1.190	1.28	0.998	1.540	1.54	0
(finito, semisup)	0.931	1.194	1.28	0.999	1.540	1.54	0
(entire, entire)	0.929	1.190	1.28	0.993	1.540	1.55	0
(semisup, semiinf)	0.926	1.187	1.28	0.997	1.540	1.54	0
(finito, empty)	0.931	1.170	1.26	0.996	1.233	1.24	0
(empty, empty)	0.930	0.878	0.94	0.998	0.925	0.93	0
(empty, semisup)	0.926	0.878	0.95	0.998	0.925	0.93	4096
(empty, entire)	0.926	0.878	0.95	0.995	0.925	0.93	4096

Table 5.4: The measurement behind M.2.1, on the AVX2 path: nanoseconds per operation, the minimum over all samples, with the shapes of Table 4.6. The last column counts the operand pairs out of 4096 on which the two builds answer differently, and is one column because the two counts coincide row by row. The other two paths are in the registry: the shape of the measurement is the same and only the base times change.

Remedy. A dispatch in front of the whole `operator*` family, five call sites in all. The predicate `needsMulExtended` asks whether one operand has a zero endpoint and the other an infinite one, which is the only way $0 \cdot \infty$ can arise; when it holds the conformant `mulExtended` is used, and otherwise the kernel is, where it is correct.

Evidence. Table 5.5 gives the ten cases of the test block. Containment is restored on all three paths, and N.2 proves to be repairable in the wrapper alone, the kernel remaining in use on finite values.

Operation	Flag off	Flag on
[0,1] * entire	[-inf,-inf]	[-inf,+inf]
[0,0] * entire	NaN	[0,0]
[0,+inf]*[0,5]	NaN	[0,+inf]
[0,1]*[0,+inf]	[0,0]	[0,+inf]
[2,3] * entire	[-inf,+inf]	[-inf,+inf]
[1,+inf]*[2,3]	[2,+inf]	[2,+inf]
[1,2] * (-inf)	[-inf,-inf]	[-inf,-realmax]
(+inf) * [1,2]	[+inf,+inf]	[realmax,+inf]
[-2,-1]*(+inf)	[-inf,-inf]	[-inf,-realmax]
[1,2] *= (+inf)	[+inf,+inf]	[realmax,+inf]

Table 5.5: The product with unbounded endpoints, from the test block of M.2.2. The first four rows are the cases the entry repairs, where the answer without the flag is either a NaN or an interval that does not contain the product; the two that follow are the ones the narrowed dispatch returns to the kernel, no zero endpoint meeting an infinite one. The last four exercise the overloads that take a number, where the pair $[b, b]$ with b infinite is not a datum and the answer may not be the degenerate infinite interval that M.2.19 forbids.

Cost. Two classes at once, A and B. With both operands bounded the answer does not change and the ratio measures the added checks, which is class A: 1.399 to 3.256 nanoseconds on AVX2, a ratio of 2.33, which is the highest of the library and is so because the operation underneath costs a nanosecond and a half. Where $0 \cdot \infty$ can arise all 4096 operands answer differently, which is class B, and the ratio of 5.32 compares a wrong answer with a right one.

M.2.3 Division with a divisor that touches zero

Cl. 10.4

cae156a

Defect. Every division whose divisor touched or contained zero returned a NaN: $[1, 2]/[0, 2]$, $[2, 3]/[-1, 1]$, $[1, 2]/[0, 0]$ and $[1, 2]/[0, +\infty]$ all did, where the clause requires the half-lines, **Entire** and **Empty**.

Remedy. The quotient is computed as a product by the extended reciprocal, so that the half-lines follow from the same convention $0 \cdot \infty = 0$ adopted in M.2.2, and the dispatch is placed in front of the whole **operator/** family.

Evidence. Table 5.6 gives the thirteen cases of the test block: twelve of them fail without the flag, and the half-lines, **Entire** and **Empty** appear with it where Cl. 10.4 requires them. The kernel stays in use on divisors of definite sign.

Division	Flag off	Flag on
$[1, 2]/[0, 2]$	NaN	$[0.5, +\infty]$
$[2, 3]/[0, 2]$	NaN	$[1, +\infty]$
$[-3, -2]/[0, 2]$	NaN	$[-\infty, -1]$
$[2, 3]/[-2, 0]$	NaN	$[-\infty, -1]$
$[-3, -2]/[-2, 0]$	NaN	$[1, +\infty]$
$[2, 3]/[-1, 1]$	NaN	$[-\infty, +\infty]$
$[-2, 3]/[-1, 1]$	NaN	$[-\infty, +\infty]$
$[1, 2]/[0, 0]$	NaN	Empty
$[0, 0]/[0, 2]$	NaN	$[0, 0]$
$[2, 3]/\text{entire}$	NaN	$[-\infty, +\infty]$
$\text{entire}/[2, 3]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$
$[1, 2]/[0, +\infty]$	NaN	$[0, +\infty]$
$[1, 2]/[-\infty, 0]$	NaN	$[-\infty, 0]$

Table 5.6: Division with a divisor that touches or contains zero, from the test block of M.2.3. The only case already right without the flag is **entire**/ $[2, 3]$, a divisor of definite sign, which never leaves the kernel; the other twelve returned a NaN where Cl. 10.4 requires a half-line, **Entire** or **Empty**.

Cost. Not measured, and the reason is that the code this entry documents no longer exists: M.2.4 replaced it, for the reason its own card gives, so there is nothing left to time. The cost of the conformant division is measured there.

M.2.4 *tightest* division, dividing the endpoints directly

Cl. 12.10.2, Cl. 10.4

6c174a5, c167135

Defect. M.2.3, which this entry replaces, computed the quotient as

$$a/b = \text{mulExtended}(a, \text{recipExtended}(b)).$$

The reciprocal rounds once and the product rounds a second time, so the result contained the quotient without being the narrowest representable interval, as Table 5.7 shows on two cases, and division is a basic operation of Table 9.1, to which Cl. 12.10.2 requires *tightest*. The damage was bounded, in that a divisor of definite sign never reached that path and stayed on the kernel, where it was already *tightest*.

Division	M.2.3, flag on	exact
$[-30, -15]/[-\text{inf}, -3]$	$[0, 10.000000000000002]$	$[0, 10]$
$[-30, -15]/[3, +\text{inf}]$	$[-10.000000000000001, -0]$	$[-10, -0]$

Table 5.7: The double rounding that M.2.4 removes. Both answers of M.2.3 contain the exact quotient, so they are *valid*; neither is the narrowest representable interval that does, which is what Cl. 12.10.2 requires of a basic operation. The one extra rounding is worth one *ulp* on the endpoint that the reciprocal touches.

Remedy. `recipExtended` is removed and `divExtended` rewritten to divide the endpoints directly, one rounding per endpoint. With a divisor of definite sign the endpoints of the quotient over the whole rectangle fall in two of its four corners, chosen by the signs, and that choice is also what keeps ∞/∞ and $0/0$ from ever being formed.

Evidence. On the corpus the failures fall from 1144 to 1091 with the flag on, fifty-three assertions repaired and none broken, compared case by case rather than by subtracting two totals.

Cost. Two classes, A and B, according to the divisor. Where it has definite sign the two builds answer identically and the ratio is a surcharge, which is class A: 2.662 to 4.324 nanoseconds on AVX2, a ratio of 1.62, and 1.35 and 1.32 on the other two paths. Where the divisor contains zero every one of the 4096 operands answers differently, which is class B, and the ratios between 4.0 and 7.5 compare two procedures rather than measuring a cost.

M.2.5 Empty outside the four arithmetic operators

Cl. 12.10.2, Cl. 10.5.9

2a44570

Defect. M.2.1 had made `Empty` absorbing in `+` `-` `*` `/` alone, and every other operation went straight to the kernel. The two serious cases were `min` and `max`, which returned an object whose lower endpoint exceeded its upper one: not an interval, not even the empty one, and any computation receiving it carries on over a meaningless datum.

Remedy. The guard is placed in front of `sqr`, `inv`, `abs`, `sqrt`, `min` and `max`; `operator&` and `operator|` now call the wrapper rather than the kernel and inherit it. `width` and

middle return a NaN on **Empty**, since a function with a real value cannot signal the empty set with an interval, which is what Cl. 12.12.8 prescribes.

Evidence. Table 5.8 gives the five cases in which the flag changes the answer. On the corpus the failures fall from 1091 to 1057 with the flag on, thirty-four assertions repaired and none broken.

Operation	Flag off	Flag on
<code>min(empty, [1,2])</code>	<code>[1,-inf]</code>	<code>Empty</code>
<code>max(empty, [1,2])</code>	<code>[inf,2]</code>	<code>Empty</code>
<code>empty.width()</code>	<code>-inf</code>	<code>NaN</code>
<code>entire.middle()</code>	<code>NaN</code>	<code>0</code>
<code>[-inf,1.2].middle()</code>	<code>-inf</code>	<code>-realmax</code>

Table 5.8: The five cases of the test block of M.2.5 in which the flag changes the answer. The first two are the serious ones: the answer without the flag has its lower endpoint above its upper one, so it is not an interval and not the empty one either. The last three are Cl. 12.12.8, where a function with a real value cannot signal the empty set with an interval, and the midpoint of an unbounded interval is a finite number.

Cost. Class A on an ordinary operand and class B on the empty one. On AVX2 the surcharge runs from 1.05 on **sqr** to 1.38 on **min**, and it is the guard alone: the ratio is largest where the operation underneath is cheapest. On an empty operand the ratio falls below unity, to 0.73 on **sqr**, because the guard recognises the empty interval and never calls the kernel at all.

M.2.6 The domain filter in the elementary functions

Cl. 10.4

0975740

Defect. Every function with a restricted domain had the same shape: either the interval lay wholly inside the domain, or the result was a NaN, written `if (! (x >= 0.)) return constants::nan;`. An interval overlapping the boundary by however little therefore lost all information, where Cl. 10.4 asks for the image over the part of the box that lies inside the domain.

Remedy. One service function, **clampToDomain**, intersects the argument with the domain and reports an empty intersection; the fourteen functions concerned call it instead of the guard. The old guard stays in the **#else** branch, so nothing changes with the flag off.

Evidence. On the corpus the failures fall from 1057 to 978 with the flag on, seventy-nine assertions repaired and none broken: ten each in **log**, **log2**, **log10**, **atanh**, **asin** and **acos**, six in **sqr** and six in **acosh**.

Cost. With the argument inside the domain the surcharge is the filter alone and stays between 1.04 and 1.12 on the two vector paths. On the boundary the comparison is not like for like, since without the flag the function is not evaluated at all; and there the scalar path reads 0.26, conformance being four times faster, because the NaN the other

build returns a signalling one and the x87 registers raise a microcoded assist of about a hundred and fifteen nanoseconds on it.

M.2.7 The domain filter as a public facility

Cl. 10.4

1201e35

Defect. M.2.6 had written the rule once, but in a **static** function hidden inside **src/Interval.cpp**, which the compiler lets only that one file call. A user building functions of their own over a restricted domain, which is the use the research group makes of the library, had to write the intersection again, and that is precisely the part that is easy to get wrong: the open boundary against the closed one, the empty intersection, the propagation of **Empty** and of a NaN.

Remedy. The rule moves up into the header as a free function, **tight::intersectDomain**, which takes the argument, the two bounds of the domain and two flags saying whether each bound is open, and returns the intersected interval, using **Empty** for the empty intersection so that a caller can propagate the result without distinguishing the cases. Being a name that did not exist before, it sits outside the flag.

Evidence. The failures are 978 with the flag on and 1452 with it off, that is exactly those of M.2.6, with zero differences checked assertion by assertion and not by subtracting totals. A change that moves code without changing behaviour must produce no movement at all, and that is the result.

Cost. The ratio does not measure the filter, which is the same code in both builds, but the constructor the filter reaches when it really restricts the argument: it moves in that shape alone, reading 1.10 on AVX2, 0.99 on SSE2 and 1.33 on the scalar path. On the shapes that return **Empty** or the argument unchanged nothing is constructed, and the six cells lie between 0.89 and 1.16, which is one within the margin of Section 4.3.3.

M.2.8 The missing numeric functions: **rad**, **mag**, **mig**

Cl. 10.5.9

2f2071a

Defect. The three functions did not exist at all: zero occurrences in the header. It is a gap in the surface rather than a defect of behaviour, and the external corpus saw it, since the plugin recorded that they were absent and the corresponding cases were skipped.

Remedy. Three member functions, **radius**, **magnitude** and **mignitude**, named after the convention of the library rather than after the abbreviations of the table. They sit outside **TIGHT_IEEE1788_STRICT**: they are new names, there is no earlier behaviour to preserve, and putting them behind a conformance flag would give two different interfaces.

Evidence. In two parts. With the corpus unchanged the failures stay at 978 and 1452, so the three functions disturb nothing. Declaring them in the plugin, the corpus grows from 4358 to 4409 assertions, fifty-one more, and the failures stay the same: all fifty-one new assertions pass in both builds.

Cost. **mag** and **mig** give exactly 1.00, which is the control of the entry: they return a number and construct no interval, so they inherit nothing. **rad** gives 7.12 on AVX2 with

sixty-seven diverging operands, and neither figure belongs to this entry: it calls `middle`, which the flag changes, and the cost is that of M.2.20.

M.2.9 `tan` at its poles

Cl. 10.4, Cl. 10.5.3

a354578

Defect. Around a pole `tan` returned a NaN, in two places in `src/Interval.cpp`: one for an interval at least a period wide, one for an interval that crosses a pole.

Remedy. The domain of `tan` is the line without the odd multiples of $\pi/2$, and that is not an interval, so the filter of M.2.6 does not apply to it. Cl. 10.4 asks for the image over the part of the argument inside the domain all the same, and next to a pole the two one-sided limits are $-\infty$ and $+\infty$: the enclosure is **Entire**.

Evidence. On the corpus the failures fall from 978 to 863 with the flag on, one hundred and fifteen assertions repaired and none broken, all of them and only those of `tan`, whose two suites go to zero failures. The entry is also the one that exposed a defect of the benchmark rather than of the library, and every earlier figure was measured again because of it.

Cost. Where `tan` is really evaluated the flag costs nothing measurable: 1.02 on AVX2 and unity on the other two paths, over an operation of about a hundred and fifty nanoseconds. On the shape that crosses a pole the conformant build answers in 154 nanoseconds where the other takes 272 on the scalar path, the difference being again the assist on a signalling NaN.

5.6.2 The powers and the exponentials

The next nine are the powers and the exponentials, where the same operation is repaired several times over because its defects are independent of one another. The domain of `pow` is among them, and its share of the corpus, four hundred and ninety-nine assertions, is the largest of any single change in this work.

M.2.10 The domain of `pow`

Cl. 10.4, Cl. 10.5.3

7f88aaf

Defect. The guard was `if (lower(x) < 0) return constants::nan;`, which is the all or nothing shape that M.2.6 had already removed from the elementary functions. Empty and the case of a zero base were missing as well.

Remedy. The domain of `pow` is not a rectangle: the base must be non-negative, and a zero base is admitted only against a positive exponent. The two arguments cannot therefore be filtered together, so the filter of M.2.7 is applied to the base alone and the corner $x = 0$ is treated apart, by asking whether the exponent is positive.

Evidence. On the corpus the failures fall from 863 to 364 with the flag on, four hundred and ninety-nine assertions repaired and none broken, all of them in the `pow` suite, which is the largest single repair of the work.

Cost. On an ordinary argument the ratio is 1.02, that is nothing measurable. The 45.47 on the shape whose base overlaps the boundary is the highest figure of the campaign and is *not* the cost of conformance: the build without the flag returns its NaN in two and a half nanoseconds without evaluating anything, while the conformant one restricts the base and computes a power. Measured separately, `pow` itself costs 115.535 nanoseconds and the domain filter 1.747, that is one and a half per cent of the work; Section 8.2.2 takes the case apart.

M.2.11 The lower endpoint of `pow`

Cl. 10.4

392536f

Defect. Where the exact value is zero, the typical case being $x^{+\infty}$ with $x < 1$, `powDown` rounded downwards all the same and produced -2^{-1074} , the smallest negative subnormal. That is not a matter of tightness: it is an endpoint the function never attains.

Remedy. One line: if the computed value is zero, `powDown` returns zero instead of rounding it down. The rule holds in both ways a zero can arise, and for one reason: zero is already a correct lower endpoint, and if a tiny positive value has collapsed to zero by underflow, zero is below it.

Evidence. On the corpus the failures fall from 364 to 140 with the flag on, two hundred and twenty-four assertions repaired and none broken.

Cost. Measured with M.2.10, whose table it shares: the two entries live in the same function and the flag switches them off together.

M.2.12 The base of `pow` and the sign of zero

Cl. 10.2, Cl. 10.4

ca28443

Defect. IEEE 754-2019 keeps the sign of zero, and its rule for `pow` uses it: `pow(-0, y)` with y an odd negative integer is $-\infty$, while the limit of x^y as $x \rightarrow 0^+$ is $+\infty$. The result came out malformed, its lower endpoint above its upper one: `pow([-0, 0.5], [-1, 1])` answered `[0, -inf]`, where the conformant build answers `[0, +inf]`. It is the gravest case of the series, since the object returned is not an interval at all, and in a debug build the assertion in the constructor aborts on it.

Remedy. One line: if the lower endpoint of the base is zero, the interval is rebuilt with the positive zero. The domain filter could not do it, because $-0 < 0$ is false and there was nothing to cut.

Evidence. On the corpus the failures fall from 140 to 113 with the flag on, twenty-seven assertions repaired and none broken, and with them the `pow` suite reaches zero failures.

Cost. Measured with M.2.10. The shape named after the negative zero carries it, and its ratio of 1.07 says that reading the sign of an endpoint costs nothing against a power of a hundred nanoseconds.

M.2.13 Empty absorbing in the elementary functions

Cl. 12.10.2

5d0f187

Defect. The functions defined over the whole real line had no domain to filter and therefore carried no guard at all, so on **Empty** they evaluated the point function at the two endpoints $+\infty$ and $-\infty$ and returned what came out, which as Table 5.9 shows is often not an interval.

Expression	Flag off	Flag on
<code>sin(empty)</code>	[NaN, -NaN]	Empty
<code>cosh(empty)</code>	[+inf, +inf]	Empty
<code>exp(empty)</code>	[+inf, 0]	Empty
<code>atan(empty)</code>	[1.5707..., -1.5707...]	Empty
<code>erfc(empty)</code>	[2, 0]	Empty

Table 5.9: Five of the nineteen cases of the test block of M.2.13. The answer without the flag is the point function evaluated at the reversed pair: three of these five have their lower endpoint above their upper one, and `cosh` returns the degenerate infinite interval that M.2.19 forbids.

Remedy. Three lines at the head of nineteen functions of `src/Interval.cpp`, from `sin` to `erfc`. The change is purely additive: it removes no line.

Evidence. With the flag off sixteen of the nineteen cases fail; the three that pass, `sinh`, `asinh` and `cbrrt`, are odd functions written as $-f(-\ell)$, so on **Empty** that symmetry returns the reversed pair by accident rather than by construction. On the corpus the failures fall from 113 to 105 with the flag on, and the eight assertions repaired are exactly the empty cases of the eight suites concerned.

Cost. On an ordinary argument the guard is invisible, between 1.03 and 1.12, the functions underneath costing tens of nanoseconds. On an empty argument the ratio collapses to 0.14, because the guard answers without calling the elementary function: this is the clearest case in the library of conformance being faster than its absence.

M.2.14 The exponentials rounded below their minimum

Cl. 10.4, Cl. 12.10.2

6f2b476

Defect. The lower endpoint of a monotone function is obtained by rounding the point evaluation downwards. Where the computed value is already the minimum of the image, the predecessor falls outside the image: the predecessor of 0 is -2^{-1074} for `exp`, `exp2` and `exp10`, and the predecessor of -1 is just below -1 for the three `m1` variants, as Table 5.10 shows. This is the one entry of the series that repairs no violation: the previous answer contains the image, so it is *valid*, and it is even *accurate*, since Cl. 12.10.2 requires *tightest* of the basic operations alone. What the entry buys is tightness.

Remedy. A service function of two lines, `atLeast`, lifts a value up to the lower bound of the image, and a call to it is placed in the six exponentials. That bound is a constant known at compile time: zero for the first three, minus one for the others.

Expression	Flag off	Flag on
<code>exp([-inf,0])</code>	$[-2^{-1074}, 1]$	$[0, 1]$
<code>exp(entire)</code>	$[-2^{-1074}, +\infty]$	$[0, +\infty]$
<code>exp2([-inf,-1])</code>	$[-2^{-1074}, 0.5]$	$[0, 0.5]$
<code>expm1(entire)</code>	$[-1.0000000000000002, +\infty]$	$[-1, +\infty]$

Table 5.10: The lower endpoint of the exponentials, from the test block of M.2.14. The value on the left contains the image and is therefore *valid*, and no line of the corpus fails because of it; it is an endpoint the function never attains, which is a question of tightness and not of conformance.

Evidence. On the corpus the failures fall from 105 to 49 with the flag on, fifty-six assertions repaired and none broken.

Cost. Measured with M.2.13, whose table covers the family.

M.2.15 The exponentials that overflow past `realmax`

Cl. 10.4, Cl. 10.2

1a0642c

Defect. The predecessor of $+\infty$ is $+\infty$, so when the point evaluation overflowed the lower endpoint stayed infinite and the answer was $[+\infty, +\infty]$. Read as a set of reals that interval is empty, so it does not contain the true value, which for `exp(710.47)` is about $2.2 \cdot 10^{308}$. This is a violation and not a matter of tightness.

Remedy. A service function `lastFinite` brings a $+\infty$ back to the largest finite `double`, so that `exp(710.47)` answers $[\text{realmax}, +\infty]$, and a call to it follows that of M.2.14 in the six exponentials. The upper endpoint is not touched, since there $+\infty$ is the right answer.

Evidence. On the corpus the failures fall from 49 to 46 with the flag on, three assertions repaired and none broken.

Cost. Not measured, and the code is superseded: M.2.19 applies the same rule in the constructor, which every function passes through on its way out, and that is the general remedy this entry had asked for. Verified by making `lastFinite` the identity: over 120,840 answers per path not one changes, and the 26,300 that overflow still return $[\text{realmax}, +\infty]$.

M.2.16 The exponentials that underflow to zero

Cl. 10.4

3cf7f86

Defect. The mirror image of M.2.14, with a graver consequence. The minimum of the image is a limit the function never attains on a finite argument, so it cannot be an *upper* endpoint either, and it was: $[0, 0]$ asserts that the result is exactly zero where it is merely very small, so the true value lies outside the interval returned.

Remedy. A third service function, `aboveFloor`, replaces an upper endpoint that does not exceed the lower bound of the image with the successor of that bound. On the argument $[-1.1\text{e}197, -2089.9]$ the answer of `exp10` becomes $[0, 2^{-1074}]$ in place of $[0, 0]$.

Evidence. On the corpus the failures fall from 46 to 44 with the flag on, and with those two assertions the unary suite of the `fi_lib` corpus reaches zero failures: the `exp` family is closed.

Cost. Measured with M.2.13. The entry carries the highest ratio of that table, 2.68 on the scalar path of the shape that underflows, because `aboveFloor` calls `nextafter` on every operand concerned.

M.2.17 Empty absorbing in `pown` and `hypot`

Cl. 12.10.2

af2cc96

Defect. The last two operations of Table 9.1 without a guard, M.2.13 having left them out because neither takes a single interval. `pown` was the worse of the two: with exponent zero it answered $[1, 1]$ whatever the input, the case being decided before the argument was looked at. But $0^0 = 1$ is a statement about a real number, not about the empty set.

Remedy. A guard at the head of `pown`, before the switch on the exponent, which is where the answer was escaping, and one at the head of `hypot` on both operands.

Evidence. On the corpus the failures fall from 44 to 38 with the flag on, six assertions repaired and none broken.

Cost. The clean rows are those of `hypot`, where the guard is the only addition: 1.03 on a finite argument, and 0.04 on the empty one, the guard answering in two nanoseconds where the other build takes forty-six. Inside `pown` the rows carry M.2.18 as well, the flag switching the two off together.

M.2.18 `pown` with a negative exponent

Cl. 10.4

d3b0cfd

Defect. Table 10.1 gives the domain of x^e for $e < 0$ as the reals without zero. The code treated an interval containing zero as undefined and answered `Entire`: a valid enclosure that throws away all the information. It is the same mistake of method that M.2.6 had corrected for the restricted domains, and the reason the domain filter had been made public in M.2.7.

Remedy. An auxiliary function `pownNegative` with six cases: the point zero alone, which is `Empty`; an even exponent without zero, which is $[\text{mag}^e, \text{mig}^e]$ and is the second real reuse of M.2.8; an even exponent with zero; an odd exponent without zero; an odd exponent with a zero endpoint, which gives a half-line; and zero strictly inside, which is `Entire`.

Evidence. On the corpus the failures fall from 38 to 10 with the flag on, twenty-eight assertions repaired and none broken.

Cost. Measured with M.2.17. Its two rows diverge on all 4096 operands, and the ratio of 32.32 on the first is not a cost of this entry: the build without the flag receives `Entire` in under three nanoseconds without computing the power at all. Where both builds compute, the conformant path is in fact the cheaper one, eighty-seven nanoseconds against a hundred and sixty, since it calls the power once instead of twice.

5.6.3 The constructor, the predicates, and the rest of Table 9.1

The last nine are the constructor and what came after it: the midpoint, the two predicates that were wrong rather than missing, `atan2` and `rsqrt`, and the two entries that closed Table 9.1 once an audit of the API against it had found seven of its thirty-nine rows absent.

M.2.19 The degenerate infinite interval

Cl. 10.2, Cl. 10.4

328e5f5, ef66d94

Defect. $[\infty, \infty]$ and $[-\infty, -\infty]$ are not intervals: as sets of reals they are empty, but they are not **Empty**, so they pass every malformation check and propagate as ordinary objects. They arise from an evaluation that overflows, and at that moment they stop containing the value they came from. M.2.15 had closed the case one function at a time and had declared the limit of doing so.

Remedy. Four lines in the constructor, where every function passes on its way out. The pair $(\infty, -\infty)$ is left alone, since that one is **Empty**. A second commit changed how the question is asked without changing the answer: the pair is now recognised from the bits, an equality between two integers, because an equality between two **double** values costs four instructions, the compiler having to rule out a NaN through the parity flag.

Evidence. On the corpus the failures fall from 10 to 8 with the flag on, two assertions repaired and none broken, and the **pown** suite reaches zero failures.

Cost. The entry sits in the constructor, so it is class C: it is paid once per interval built, by every operation that builds one. On the isolated constructor the ratio was 2.78 on AVX2 before the second commit and is 1.38 after it; inside **exp** it was already inside the margin and stays there, about one part in ten. It is the only entry of the work whose cost is better read in the entries that inherit it than in its own table.

M.2.20 The midpoint rounded to nearest

Cl. 12.12.8, Cl. 10.5.9

f04e68a, 6eb1b7b

Defect. A numeric function answers with a number and not with an enclosure, so rounding it outwards makes no sense, and Cl. 12.12.8 settles the direction outright: for the case with both endpoints finite it asks that the result be the Level 1 value rounded to nearest, a recommendation in general and an obligation for a 754-conforming type, which ours is by the claim of M.12.5. But the whole library runs with the floating-point unit pinned to **FE_UPWARD**, and the midpoint suffered it, in the three ways of Table 5.11. The graver of them is the last, where the sum of the endpoints overflows while the midpoint itself is perfectly finite.

Remedy. Inside **middle** the mode is switched to nearest, the midpoint is computed, and only if the sum is not finite does the code fall back on halving each endpoint first, which is exact; then the mode is restored. A second commit replaced the three calls into the C library, which go through the procedure table, with two instructions that write the rounding field of the control register directly.

Expression	Flag off	Flag on
mid of two adjacent doubles	the upper one	the one with even significand
$\text{mid}([-2^{-1074}, 2^{-1073}])$	2^{-1074}	0
$\text{mid}([\text{realmax}/2, \text{realmax}])$	$+\text{inf}$	1.348e308

Table 5.11: The three consequences of computing the midpoint under `FE_UPWARD`, from the test block of M.2.20. The first two answer a number that is not the nearest to the Level 1 midpoint, which Cl. 12.12.8 requires of a 754-conforming type; the third answers a number that is not finite, where the midpoint is, because the sum of the two endpoints overflows before the halving.

Evidence. On the corpus the failures fall from 8 to 3 with the flag on, five assertions repaired and none broken, and in particular none on `rad`, which is computed from the midpoint and is where a change of rounding would show first.

Cost. The highest ratio of the campaign among those that really measure the flag: 17.63 on AVX2 and up to 27 on the scalar path, where the mode is actually switched. It is the price of a change of rounding mode, about thirty nanoseconds, against a midpoint that otherwise costs two thirds of one; the second commit is what brought it down by nearly two thirds. On the shapes that answer before reaching the switch, the unbounded ones, the ratio falls to about 1.9.

M.2.21 `isMember` asks about a real number

Cl. 10.5.10, Cl. 10.2

b173a8f

Defect. `isMember` asks whether a real number belongs to an interval. Neither an infinity nor a NaN is real, so neither belongs to any interval, **Entire** included: **Entire** is written with infinities but is the real line and nothing else. The comparison on the endpoints answered yes for both infinities.

Remedy. One line: an argument that is not finite belongs to no interval.

Evidence. On the corpus the failures fall from 3 to 1 with the flag on. That one is `neg [nai]`, which no change to the library can close, since both sides are already identical bit for bit and an equality between two NaN values is false by construction.

Cost. Class A on the two shapes where the answer does not change, and there the added test costs 1.28 on AVX2, about a quarter of a nanosecond. On the infinite argument the two builds answer differently and the conformant one is the faster, since it rejects the argument before comparing anything.

M.2.22 Equality between sets

Cl. 10.5.10

d225098

Defect. `operator==` was true only when both operands were the same single point, so $[1, 2] = [1, 2]$ was false; and `operator!=` was not its negation but a disjunction, so with the flag off both were false at once, as Table 5.12 shows. It was the most insidious defect of the series because the corpus could not see it: the plugin reimplemented equality on

the endpoints, and without that choice every test on a non-degenerate interval would have failed for an unrelated reason.

Expression	Flag off	Flag on
<code>[1,2] == [1,2]</code>	false	true
<code>[1,2] != [1,2]</code>	false	false
<code>[1,2] != [1,3]</code>	false	true
<code>empty == empty</code>	false	true
<code>entire == entire</code>	false	true
<code>[5,5] == [5,5]</code>	true	unchanged

Table 5.12: Equality and inequality, from the test block of M.2.22. The first two rows are the pair that gives the entry away: the same interval is neither equal to itself nor different from itself. The last row is the only case the old comparison answered correctly, and it is the one it was written for, two operands that are the same single point.

Remedy. Equality becomes equality of the two endpoints, with `Empty` treated first, being equal to itself and to nothing else, and inequality becomes the negation of it.

Evidence. The corpus is unchanged at one failure, as expected. The value of the change lay outside it, and the corpus sees it only from G.4, where declaring the operation makes eight failures appear with the flag off, which are exactly the lines the old equality had been answering wrongly.

Cost. Between 1.26 and 2.08 on AVX2, on an operation that costs under a nanosecond. Two of the four shapes diverge on all 4096 operands, which is the defect being measured rather than a surcharge.

M.2.23 The enclosure of `atan2`

Cl. 10.4, Cl. 10.5.3

6119d8f

Defect. The function is the one of the C library, whose first argument is the *ordinate*: `atan2(y, x)` is the angle of the point (x, y) in the plane,

$$\text{atan2}(y, x) = \begin{cases} \arctan(y/x) & x > 0, \\ \arctan(y/x) + \pi & x < 0, y \geq 0, \\ \arctan(y/x) - \pi & x < 0, y < 0, \\ +\pi/2 & x = 0, y > 0, \\ -\pi/2 & x = 0, y < 0, \end{cases}$$

with the origin excluded and image $(-\pi, \pi]$. The two lines with $x < 0$ are the cut along the negative axis, and that is where the defect is born. There are three of them, of which the second sets conformance against usefulness. The origin is the one point outside the domain, so a rectangle reduced to it has an empty image and one containing it still has an image; TIGHT returned a NaN for both. On a rectangle that lies left of the y axis and crosses $y = 0$ the values crowd against both ends of $[-\pi, \pi]$, and the library returned the complement, which is narrower and is not an interval.

Remedy. A conformant function replacing the whole dispatch, organised by the position of the rectangle: the origin alone gives `Empty`, the crossing of the cut gives $[-\pi, \pi]$, the lower half plane is handled by mirroring, since `atan2` is odd in y , and otherwise the endpoints fall on the corners. The constants are the already rounded π and $\pi/2$, because on the exact angles the correctly rounded library returns the nearest `double`, which for π lies below it whatever the rounding mode.

Evidence. The corpus was unchanged at the time, the plugin not declaring `atan2`; declared afterwards it contributes 225 assertions, of which 178 failed before this entry and none after.

Cost. This is the one place in the work where conformance makes the answer *worse*, and the benchmark shows it with the sign reversed: on the cut the ratio is 0.12, because $[-\pi, \pi]$ is a constant and needs no evaluation, while the non-conformant build computes the complement with two calls. The conformant answer is 6.28 wide where the complement was 4.71, and that is why the entry sits behind the flag.

M.2.24 `rsqrt`, and the thirteen functions the corpus never reaches

Cl. 10.2

d4fc879

Defect. `rsqrt` is decreasing, so the lower endpoint of the result comes from the upper endpoint of the argument. The code took both from the lower one, so every argument wider than a point came back with its endpoints reversed: `rsqrt([0.25, 0.75])` answered `[2, 1.1547]`. The same defect is in the upstream, verified there, and a forgotten diagnostic print ran on every call.

Remedy. The endpoints now come from the opposite ones, the exactness test is applied to the right endpoint, and the print is removed. It is the first change of the series that *removes* lines of the maintainer rather than adding to them.

Evidence. The corpus is unchanged at one failure, none of these functions appearing in it: the third case, after M.2.22 and M.2.23, in which the value of a change lies outside the test bench. The entry is therefore verified by properties instead, thirteen functions checked for `Empty` absorbing, for the absence of a `NaN`, for the absence of a reversed pair, and for the domain filter, the last by comparing two calls to the library rather than against a constant written by hand. Table 5.13 gives the counters in the two builds.

Property checked over the thirteen functions	Flag off	Flag on
Functions in which <code>Empty</code> is not absorbing	14	0
Endpoints returned as <code>NaN</code>	41	0
Cases of <code>rsqrt</code> failing, out of four	2	0

Table 5.13: The properties checked on the thirteen functions the corpus never reaches, in the two builds of the same source. None of these functions has a line in the corpus, so the counters are the only evidence there is; the one case of `rsqrt` that fails without the flag for a reason of its own is `rsqrt([1, +inf])`, which without the domain filter answers $[-2^{-1074}, 1]$.

Cost. Not measured. The entry repairs a defect of the upstream rather than a non-conformance, so it sits outside the flag and both builds receive it, and there is no pair of builds to compare.

M.2.25 fma

Cl. 10.5.3, Cl. 12.10.2

f8eb980

Defect. The operation did not exist: zero occurrences in the sources. The kernel exposes a fused multiply-add, but only under the AVX2 macro, so it could not serve as a portable base.

Remedy. `fma` is added as a free function, outside the flag, being a new name. The point of the entry is that `fma` is not a multiplication followed by an addition, which is exactly why the standard lists it separately: the fused operation rounds once, and computing xy and then adding z rounds twice and is not *tightest*. It is the same mistake that had M.2.3 replaced by M.2.4 on the division.

Evidence. Declaring the operation the corpus grows from 4947 to 5511 assertions, the largest single step of the work, and all 564 new ones pass with the flag on and with it off. The tightness check has no expected value written by hand: it compares the fused operation with the multiplication followed by the addition on $a = 1 + 2^{-52}$ and $z = -1$, and requires the first to be contained in the second and strictly narrower, so that any implementation which multiplies and then adds fails it by construction. The measurement is in Table 5.14.

Computed as	Lower endpoint	Upper endpoint
<code>fma([a,a],[a,a],z)</code>	4.44089209850062617e-16	4.44089209850062715e-16
<code>[a,a]*[a,a] + z</code>	4.44089209850062617e-16	6.66133814775093925e-16

Table 5.14: The fused operation against a multiplication followed by an addition, on $a = 1 + 2^{-52}$ and $z = -1$. The lower endpoints coincide, the exact value $2^{-51} + 2^{-104}$ rounding down to the same `double` either way; the two paths are told apart by the upper endpoint alone, where the one that rounds twice overshoots by half. This is why Cl. 10.5.3 lists the operation separately instead of leaving it to the two others.

Cost. 1.05 on a finite argument and exactly 1.00 on the empty one, where the guard answers without constructing anything. The entry is one of those that made the law of Section 8.2.1 visible: it costs seventeen nanoseconds, so the added checks disappear into it.

M.2.26 The six integer functions, and Table 9.1 closes

Cl. 10.5.3, Cl. 12.10.2

cb6e12b

Defect. None of the six existed. The word `sign` occurred ten times in the sources, every one of them inside a comment.

Remedy. Six free functions, outside the flag. All six are non-decreasing over the whole line, so the interval extension is simply the image of the two endpoints and is exact: the

results are integers, or infinities, and nothing is rounded anywhere. The two rounding functions cannot use the library routines that honour the current mode, since the library keeps `FE_UPWARD` installed, so ties to even is written out by hand.

Evidence. Declaring them the corpus grows from 5511 to 5599 assertions and all 88 new ones pass in both builds. With this entry Table 9.1 is covered, thirty-nine operations out of thirty-nine. Table 5.15 gives the rows on which the two ways of rounding a tie part company, which is where the entry could have gone wrong.

Argument	roundTiesToEven	roundTiesToAway
<code>[-1.9, 2.5]</code>	<code>[-2, 2]</code>	<code>[-2, 3]</code>
<code>[-1.5, 2.5]</code>	<code>[-2, 2]</code>	<code>[-2, 3]</code>
<code>[-1.1, -0.5]</code>	<code>[-1, 0]</code>	<code>[-1, -1]</code>

Table 5.15: The two rounding functions on the arguments whose endpoints fall exactly halfway. Ties to even is the one that cannot be delegated to the C library, since `nearbyint` and `rint` honour the rounding mode in force and the library keeps `FE_UPWARD` installed, which would round 2.5 to 3 in both columns; a check written without installing that mode would not see the defect at all.

Cost. Between 1.20 and 1.28 on AVX2, and it is the purest measurement of the inherited cost in the whole work: the flag changes nothing inside these functions, which lie outside it, and what the ratio measures is the constructor of `M.2.19` that each of them calls on the way out. The ratio is highest on `sign`, which is the cheapest of the six.

M.2.27 The containment of `pow` with a subnormal exponent

Cl. 10.4, Cl. 10.5.3

0f3b040

Defect. For $x > 1$ and $y > 0$ the value x^y is strictly greater than one, so an upper endpoint equal to one does not enclose it. Measured, `pow([2, 2], [2-1074, 2-1074])` returned an upper endpoint of exactly one where the true value is $1 + 3.4 \cdot 10^{-324}$. The upper endpoint lies below the value, which violates Cl. 10.4, that is an obligation, and bears directly on the conformance claim. The cause is the one that recurs: the dispatch took its upper endpoints straight from the correctly rounded library, relying on the global rounding mode, and for an exponent that small the internal product underflows and the library returns one whatever the mode.

Remedy. A function `powUp`, the mirror of `powDown`, used by the ten upper endpoints of the dispatch. The *ulp* is added only where it is due: the exactness test already used by `powDown` tells an exact one from a rounded one, and the sign of $(x - 1)y$ says on which side of unity the true value falls.

Evidence. The corpus is unchanged, 79 failures out of 9433, and that is the point: it has no line with a subnormal exponent and could not see the defect. It is the fourth case in which the bench was blind, and the defect was found by the completeness check written for `G.13`.

Cost. Not measured separately: the entry lives inside the dispatch of `pow`, whose table is that of M.2.10, and it adds a comparison to an operation of about a hundred nanoseconds.

Chapter 6

Constructors, Operations and Reverse Mode

The blocks of this chapter differ from those of Chapter 5 in one respect that governs how they are argued. There the library had a behaviour and the behaviour was wrong, so the question was what it returned before and what it returns now. Here, with one exception, the library had nothing at all: the clause requires an operation and TIGHT did not provide it, so there is no previous answer to put beside the new one, and the evidence is the corpus, which could not exercise those lines until the operation existed.

Three groups follow. The constructors and the textual literals are the two directions of the same boundary, from text into the library and back out, and they are the exception just mentioned: a constructor did exist, and it rounded to nearest, which is the substantive defect of the group. The boolean, set and cancellative operations are three blocks in one section: the relations of Tables 10.3 and 10.4, the intersection and the convex hull, and the two-output division of Cl. 10.5.5 with the cancellative subtraction of Cl. 12.12.5. The reverse-mode operations are the twelve of Table 10.1, the largest addition of the work, and the one whose method, a bisection whose predicate is decided in interval arithmetic, is reused in Chapter 7 for the tightness of the same operations.

6.1 Constructors and text literals

Blocks G.3 and G.11 are the two directions of the same boundary, from text into the library and back out.

Inwards, Cl. 12.12.7 requires a constructor from two numbers that can *fail*, and Cl. 12.11.5 requires that the constructor from a string accept any portable interval literal. The library had a constructor that rounded to nearest and threw an exception, and rounding to nearest is the substantive defect. A literal denotes an interval of *reals*, and the constructor has to return a representable interval that contains it: writing $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ for rounding to the nearest double below and above, the requirement is

$$\text{text}(\ell) = [v, \bar{v}] \implies [\lfloor v \rfloor, \lceil \bar{v} \rceil], \quad (6.1)$$

whereas rounding both endpoints to nearest moves each of them inwards half of the time, and the result no longer contains what the literal denotes. The conversion is now made outwards, which costs two calls to the C library with the rounding mode set explicitly around each.

The rational and the uncertain forms cannot be handled that way, because they are not written as two endpoints. The uncertain form $m?rvE$ gives a midpoint, a radius counted in units in the last place of the midpoint, an optional direction and an exponent; with M the significand of m written as an integer, d its number of decimals and R the radius in the same units, the two endpoints are the exact rationals

$$\underline{v} = \frac{(M - R) 10^E}{10^d}, \quad \bar{v} = \frac{(M + R) 10^E}{10^d}, \quad (6.2)$$

with $R = 1$ and both M and the denominator doubled when the radius is absent, which is the way to say half a unit without leaving the integers. Both forms are therefore routed through arbitrary-precision integers and handed to the rational constructor, so that *no rounding happens before the last one*. The five entries took the failures from ninety-three to thirty-five out of four thousand five hundred and forty-three.

Outwards, Cl. 13.3 constrains one thing only, and it is not the appearance of the string: the interval that the printed literal *denotes* must contain the interval that was printed. Six significant digits rounded to nearest violate that, since 1.0000000000000002 becomes 1. The output therefore rounds outwards and *proves* it, by the procedure of Algorithm 6.1.

Algorithm 6.1: Printing a decimal literal that is proved to enclose, which is the procedure of M.11.2. The read-back runs in the direction *opposite* to the one being established, which is what makes the test a proof; reading it back in the same direction would prove nothing. Seventeen significant digits always terminate the loop for binary64.

Input: an endpoint x , a number of significant digits n

Output: a decimal d with $d \leq x$, printed with at most 17 digits

```

1: for  $n \leftarrow n, 17$  do
2:    $d \leftarrow$  print  $x$  with  $n$  digits, rounding downwards
3:    $b \leftarrow$  read  $d$  back, rounding upwards                                     % so  $b \geq d$ 
4:   if  $b \leq x$  then
5:     return  $d$                                                                 %  $d \leq b \leq x$ , hence  $d \leq x$  for certain
6:   end if
7: end for
```

The upper endpoint is printed by the same procedure with both directions reversed. Measured over ten thousand one hundred and ninety-eight intervals and seven values of the format parameter, the loop took on average one iteration per endpoint after the printing itself was made to round in the right direction, and the decimal form was produced in every case.

M.3.1 `numsToInterval`, a constructor that can fail

Cl. 10.5.8, Cl. 12.12.7

99feab5

Defect. TIGHT exposed no conformant constructor at all: in its place there was only the constructor from two numbers, which never fails. On the four cases the corpus prescribes as failures it answered with an object rather than with `Empty`, as Table 6.1 gives them.

Arguments	<code>Interval(l,u)</code>	Required
$(1, -1)$	<code>[1, -1]</code> , the endpoints reversed	<code>Empty</code>
<code>(NaN, NaN)</code>	<code>[nan, nan]</code>	<code>Empty</code>
$(-\infty, -\infty)$	<code>[-inf, -realmax]</code>	<code>Empty</code>
$(+\infty, +\infty)$	<code>[realmax, +inf]</code>	<code>Empty</code>

Table 6.1: The four cases the corpus prescribes as failures, against the answer of the constructor that was there. The last two answers are already those of M.2.19, which normalises the degenerate infinite pair; with the flag off they read `[-inf, -inf]` and `[inf, inf]`, which are not intervals at all. What none of the four can do is fail, and Cl. 12.12.7 requires a constructor that can.

Remedy. A free function `tight::numsToInterval`, three lines, which rejects a reversed pair with the test `!(1 <= u)`, covering a NaN argument as well since every ordered comparison with a NaN is false, and rejects the two degenerate infinite pairs. Being a new name it sits outside the flag.

Evidence. On the corpus the failures fall from 93 to 89 with the flag on and from 1544 to 1540 with it off, four assertions repaired and none broken, compared case by case.

Cost. Measured with M.3.2, which shares its table: the text constructor delegates the final validity check to this one, so the two are timed together. In isolation the function costs about one and a half nanoseconds, which is the scale against which the fifteen microseconds of a rational literal should be read.

M.3.2 `textToInterval`, with outward conversion

Cl. 10.5.8, Cl. 12.12.7

dcf83c3

Defect. Three faults, one per row of Table 6.2. The existing constructor from a string converted each endpoint with the C library, once, and to nearest, so a literal came back as an interval that does not contain the value its text denotes, which is the case Cl. 12.12.7 calls a false negative and forbids outright. It accepted text it had not read to the end, `std::stod` stopping at the first character it cannot use and reporting success all the same. And it failed by throwing an exception rather than by returning `Empty`, so a caller had no way of telling a failure from a result.

Remedy. A free function `tight::textToInterval` with a parser written for the purpose. Each endpoint is converted *in its own direction*, the rounding mode being set explicitly and restored, the number must be consumed entirely, an omitted endpoint is admitted, and the validity of the pair is delegated to M.3.1.

Literal	Old constructor	textToInterval
"[1.2345]"	[0x1.3c083126e978ep+0, 0x1.3c083126e978ep+0]	[0x1.3c083126e978dp+0, 0x1.3c083126e978ep+0]
"[1abc,2]"	[1,2]	Empty
"[-1,]"	throws	[-1,+inf]

Table 6.2: One case for each of the three faults. In the first the answer is a single point and the value the text denotes lies outside it, since the decimal is not a `double` and rounding to nearest moved the only endpoint there is; the conformant answer is the pair of adjacent doubles that straddle it. In the second the text is not a literal and was read as one. In the third an omitted endpoint, which Cl. 12.11.3 admits with the implicit value $-\infty$ or $+\infty$, raised an exception where Cl. 12.12.7 prescribes `Empty` for a failure and an interval here.

Evidence. On the corpus the failures fall from 89 to 75 with the flag on and from 1540 to 1526 with it off, fourteen assertions repaired and none broken.

Cost. Reading a literal costs between 24 nanoseconds and 15 microseconds according to its form, three orders of magnitude, against the one and a half nanoseconds of the constructor from numbers; the surcharge of conformance is between 0.95 and 1.10 on every readable cell. The two forms that go through the rational conversion cost about three microseconds, since they build integers of arbitrary precision, and the one whose value leaves the range costs fifteen, its exponent of 380 producing a numerator of that many digits. One cell of this entry is not readable at all, its `spread` reading 1.84 against the 1.05 typical of the others, and Section 8.3 says why.

M.3.3 The special literals, recognised rather than rejected

Cl. 12.11.3, Cl. 9.7

41de17d

Defect. After M.3.2 the three spellings of `[entire]` gave `Empty`: not being numbers, they were rejected. The interesting case is not that one. `[]` and `[empty]` answered `Empty` before and answer `Empty` now, so on the value nothing changes and the corpus notices nothing; what changes is the reason, since they were being rejected as malformed rather than recognised as the empty interval.

Remedy. Before the separator is looked for, the body of the literal is compared, ignoring case and outer spaces, against the empty word, `empty` and `entire`.

Evidence. On the corpus the failures fall from 75 to 72 with the flag on, three assertions repaired, the three of `[entire]`. The seven of the empty interval do not move, which is the point made above: the corpus measures the value and not the reason.

Cost. Measured with M.3.2: the special form costs 32 nanoseconds against the 341 of the inf-sup one, since nothing is converted, and the surcharge is 1.06.

M.3.4 Rational literals, routed through `fromRational`

Cl. 9.7, Cl. 12.12.7

0665de0

Defect. An endpoint of the form p/q was not rejected: it was read halfway. The C library converted the numerator, stopped at the slash and reported success, so the interval came back built on the numerators alone. That the wrong results happened to contain the right ones is an arithmetic coincidence and not a guarantee, and it is worse than a rejection, because nothing downstream can tell.

Remedy. The parser now tries the rational form when the numeric one does not consume the text entirely, and converts it with `Interval::fromRational`, which TIGHT already had: that function *encloses* p/q instead of rounding it, widening by one *ulp* only where it must. No new conversion was written; what was missing was a caller.

Evidence. On the corpus the failures fall from 72 to 68 with the flag on, four assertions repaired and none broken.

Cost. Measured with M.3.2. The rational form is the expensive one, about three microseconds against the 341 nanoseconds of the inf-sup form, that is nine times on AVX2 and five on SSE2, and the reason is that it builds integers of arbitrary precision rather than that it leaves the range.

M.3.5 The uncertain form, which is integer arithmetic

Cl. 12.11.5, Cl. 9.7.6

657c2db

Defect. The form was not recognised at all: having no brackets, it was rejected with an exception by the constructor from a string. It is 34 lines of the corpus, the largest family of the block, and Cl. 12.11.5 requires that the constructor accept any portable literal, which this is.

Remedy. The observation that guided the work is that the uncertain form reduces to integer arithmetic. With $m = M/10^d$ and a radius of R units in the last place, the two endpoints are

$$\frac{M - R}{10^d} \quad \text{and} \quad \frac{M + R}{10^d},$$

which are exact rationals with a power of ten below, so they are handed to M.3.4 and enclosed. Half an *ulp* is the same with M and the denominator doubled and $R = 1$, and the exponent field adds zeros. Nothing is converted in floating point before the final rounding.

Evidence. On the corpus the failures fall from 68 to 35 with the flag on and from 1519 to 1487 with it off, thirty-three assertions repaired and none broken.

Cost. Measured with M.3.2. The uncertain form costs about three microseconds, like the rational one it delegates to, and its surcharge is 1.01. One row of this entry carries the only crack in the rule that new names do not depend on the flag: the literal whose radius leaves the range answers differently in the two builds, because what turns $[+\infty, +\infty]$ into $[\text{realmax}, +\infty]$ is M.2.19, which is behind it.

M.11.1 The exact representation, and the recovery requirement

Cl. 13.4, Cl. 13.4.1

3fff9dd

Defect. The library had no conformant output at all. Cl. 13.4 requires a representation from which the interval can be recovered exactly, and Cl. 13.4.1 fixes its form.

Remedy. A new header that nobody in the library includes. The form is written from the bits rather than by the C library, and that is not fastidiousness: the recovery requirement is exact, so it must not depend on which normalisation a library chooses for a subnormal or for a signed zero.

Evidence. Zero MISMATCH out of 594 controls, identical with the flag on and off. The recovery requirement is a property that cannot be satisfied by accident, any loss of information in the textual form breaking it, so it is the strongest check of the block, and it comes from the clause rather than from an idea of the person writing the test.

Cost. 1.08 on a finite interval, an operation of 217 nanoseconds, and the whole of it is the two conversions. The entry also carries the row in which the constructor that reads the form back costs less with the flag than without, 0.73, which is one of the six cells of this entry that the registry declares unreadable, set aside in Section 8.3 rather than counted as a result.

M.11.2 The output that must enclose

Cl. 13.3, Cl. 12.11

c763970

Defect. Cl. 13.3 constrains one thing only, and it is not the appearance: the interval the printed literal *denotes* must contain the one printed. Printing to nearest breaks it, since a value one *ulp* above one prints as 1 at six digits; everything else the clause says is a recommendation.

Remedy. The generic output rounds outwards and *proves* it by reading the string back, adding digits until it encloses, seventeen always being enough for this format. The reading back is done in the direction *opposite* to the one being proved, which is the point on which the check could have been written so as to prove nothing at all: to establish that the printed value is below the true one, the string is read back rounding upwards, so that the value read is above the printed one, and if it is still below the true one the inequality holds for certain.

Evidence. The sanity run before the measurement found a defect of our own, and the entry carries its repair. The printing itself honours the rounding mode as much as the reading does, and the mode installed so that the reading could prove containment governed the printing too, which therefore fell on the wrong side: the decimal form was never produced, the code falling back on the exact form after twelve failed attempts, 11.94 attempts per endpoint and 7982 endpoints out of 8154 that never succeeded. With the two modes set separately inside the loop the figures become one attempt per endpoint and no endpoint that fails, and the cost of an interval falls from 8525 to 698 nanoseconds. The loop is *not* removed: its purpose is not to find the digits, which printing in the right direction already gives, but to demonstrate containment without trusting the C library.

Cost. The price of that demonstration is read by difference: the exact form costs 246 nanoseconds, six digits cost 473 more, seventeen cost 840, and the decoration costs 22.8. The surcharge of the flag is within the noise on every row, between 0.99 and 1.09.

6.2 Boolean, set and cancellative operations

These three blocks add operations rather than repairing them, and together they brought about seventeen hundred new assertions into the corpus, all of which passed on the first run. Three observations are worth keeping.

The first is a false friend, and it is the reason the block was written with care. `TIGHT` inherits from the kernel an `operator<` and an `operator<=` which are *not* the `less` of Table 10.3: they compare the upper endpoint of one operand with the lower endpoint of the other, which is exactly `strictPrecedes` and `precedes`. They are not wrong, they are misnamed, and they are what the geometric sign filter is built on. They were left alone and documented, which is the opposite of the decision taken for `operator==` in M.2.22: that one was *incorrect*, answering false to $[1, 2] = [1, 2]$, so nobody could have built anything on it.

The second is that the empty cases of Table 10.4 do not follow from the formulas on the endpoints and have to be written out one by one. The ordering relations of Table 10.3 are stated on the endpoints,

$$\text{less}(\mathbf{a}, \mathbf{b}) \iff a_{\text{lo}} \leq b_{\text{lo}} \wedge a_{\text{hi}} \leq b_{\text{hi}}, \quad (6.3)$$

$$\text{precedes}(\mathbf{a}, \mathbf{b}) \iff a_{\text{hi}} \leq b_{\text{lo}}, \quad (6.4)$$

$$\text{interior}(\mathbf{a}, \mathbf{b}) \iff b_{\text{lo}} <' a_{\text{lo}} \wedge a_{\text{hi}} <' b_{\text{hi}}, \quad (6.5)$$

and two things in them are easy to get wrong. The first is the relation $<'$ of (6.5), which is the strict order *extended* so that $-\infty <' -\infty$ and $+\infty <' +\infty$ hold: without it `interior(Entire, Entire)` would come out false, while the standard declares it true, since a set is topologically interior to itself. The second is the empty case: with $\mathbf{a} = \text{Entire}$ and $\mathbf{b} = \text{Empty}$, whose stored endpoints are $+\infty$ and $-\infty$, (6.3) would evaluate to false where Table 10.4 requires true. Those rows are tabulated in the standard on purpose and are transcribed rather than derived.

The third is that in `convexHull` the empty interval is *not* absorbing, the only place in the library where that is so. Cl. 12.12.6 distinguishes the two cases with a single word, *either* for intersection and *both* for the hull.

G.6 adds two operations whose definitions are worth writing down, because in both cases the definition is what makes the implementation non-obvious.

The first is the two-output division of Cl. 10.5.5. The set it computes is

$$\mathbf{c} /_{\text{set}} \mathbf{b} = \{x \in \mathbb{R} \mid \exists b \in \mathbf{b} : xb \in \mathbf{c}\}, \quad (6.6)$$

which has zero, one or two connected components, and the operation returns them *separately*. Keeping them apart is the entire point: the hull of the two, which is what an

ordinary division returns when the divisor crosses zero, is **Entire** and says nothing. The note to the clause warns that for this reason **mulRevToPair** is not an arithmetic operation, and putting it inside an expression loses containment unless the caller uses both outputs.

The second is the cancellative subtraction of Cl. 12.12.5, which answers the question *which z satisfies $z + \mathbf{b} = \mathbf{a}$* . Such a z exists only if $\mathbf{a}$ is at least as wide as $\mathbf{b}$, that is

$$w(\mathbf{a}) \geq w(\mathbf{b}) \iff a_{\text{lo}} + b_{\text{hi}} - a_{\text{hi}} - b_{\text{lo}} \leq 0, \quad (6.7)$$

and that test cannot be made in floating point: for operands below the unit roundoff both sides collapse onto the same double and the comparison declares the result defined in *both* directions. The sign of the left-hand side is therefore decided exactly, with the error-free transformation **2Sum**, written in the form of (6.7) rather than as $w(\mathbf{a}) - w(\mathbf{b})$ because the direct form overflows. **2Sum** is exact only under round-to-nearest while the library keeps **FE_UPWARD** installed, so the mode is changed and restored *on the ambiguous branch only*, which a comparison with directed rounding isolates first.

M.4.1 The three unary predicates

Cl. 10.5.10, Cl. 10.6.3

51a49f2

Defect. **TIGHT** had none of the three. **isEntire** existed only on the branch of the first merge request and had never been carried into the fork.

Remedy. Three free functions: **isEntire** is true when the two endpoints are infinite in that order, **isCommonInterval** when the interval is non-empty and bounded, and **isSingleton** adds that the two endpoints coincide. The point that is not obvious is that the empty interval must be excluded first, its canonical representation being written with infinities in the reverse order, which is exactly what a naive test on the endpoints gets wrong.

Evidence. Declaring them adds 117 calls to the corpus, and all of them pass with the flag on and with it off. Table 6.3 is the truth table of the manual block.

Interval	isEntire	isCommonInterval	isSingleton
Empty	0	0	0
Entire	1	0	0
[1,2]	0	1	0
[1,1]	0	1	1
[0,0]	0	1	1
[-5,-5]	0	1	1
[-inf,0]	0	0	0
[0,+inf]	0	0	0

Table 6.3: The three predicates of M.4.1 on the cases of the manual block, 1 for true and 0 for false. The first row is the one a naive test gets wrong: the canonical representation of **Empty** has the two infinities in the reverse order, so an **isEntire** written as *both endpoints infinite* answers true on it. The last two rows separate the two halves of **isCommonInterval**, which asks for non-empty *and* bounded.

Cost. These three and the seven of M.4.2 and M.4.3 are what calibrates the margin of uncertainty of the whole campaign. They construct nothing, and their assembly is identical between the two builds instruction by instruction, thirty-six comparisons out of thirty-six: the true ratio of every one of their fifty-four cells is therefore exactly one, and whatever the benchmark reads instead is the noise of the platform. Forty-five cells read 1.00 and the nine that deviate lie between 0.84 and 1.12, which is where the twelve per cent of Section 4.3.3 comes from.

M.4.2 The four order relations, and the false friend

Cl. 10.5.10

738d1cd

Defect. The four relations did not exist. Four operators with similar names did, and that is the discovery of the entry: `operator<` and `operator<=` delegate to the kernel, where they mean $\bar{a} < \underline{b}$ and $\bar{a} \leq \underline{b}$, which are exactly `strictPrecedes` and `precedes` of Table 10.3 and not `less`. They are not wrong, they answer a different question from the one their name announces.

Remedy. The four relations are added as free functions under the names of the standard, and *the operators are not touched*. The reason is that they are correct and the geometry relies on them, unlike the equality of M.2.22, which was wrong and which nobody could therefore have built upon. The cases with an empty operand do not follow from the formula on the endpoints and are written out one by one, as Table 10.4 gives them.

Evidence. Declaring them adds 251 calls to the corpus, and all of them pass in both builds.

Cost. Part of the calibration described above. One figure of substance comes out of it: the strict order of Table 10.3 is paid for, since `strictLess` costs 1.6 times `less` and `interior` 1.76 times `subset`, the helper being called twice and each comparison against an infinity taking four instructions because of the parity flag. It is below the threshold of action, so it is recorded and not optimised.

M.4.3 The three set relations

Cl. 10.5.10, Cl. 11.3

85d744f

Defect. None of the three existed under the name of the standard. A method answered a neighbouring question, with the arguments the other way round, and covered one of the three.

Remedy. Three free functions. Here Table 10.4 is not symmetric, and that is the difficulty: the empty interval is a subset of everything and lies in the interior of everything, so `subset` and `interior` are true as soon as the first operand is empty and false when the first is not empty and the second is; `disjoint` is true as soon as either is empty. `interior` also needs the strict order of the table, in which $-\infty$ is below itself, or `interior` of `Entire` in itself would come out false where note 4 of the standard declares it true.

Evidence. Declaring them adds 168 calls to the corpus, and all of them pass in both builds, and Table 6.4 is the truth table of the manual block. The negative cases of `subset`

are fixed in the manual test rather than left to the corpus, because the plugin uses that operation as scaffolding and a degenerate one would hide other failures.

First	Second	subset	interior	disjoint
[1,2]	[1,3]	1	0	0
[2,3]	[1,4]	1	1	0
[1,4]	[2,3]	0	0	0
[1,2]	[3,4]	0	0	1
[1,2]	[2,3]	0	0	0
[1,1]	[1,1]	1	0	0
Entire	Entire	1	1	0
[1,2]	Entire	1	1	0
Entire	[1,2]	0	0	0
Empty	[1,2]	1	1	1
[1,2]	Empty	0	0	1
Empty	Empty	1	1	1

Table 6.4: The three relations of M.4.3, 1 for true and 0 for false. The fifth row shows that a single point in common is enough to make two intervals not disjoint, and the sixth that a point is a subset of itself without lying in its interior, there being no margin. The seventh is note 4 of the standard, which declares **Entire** to lie in its own interior and requires the strict order of Table 10.4, in which $-\infty$ is below itself. The last three are the asymmetry of that table: **Empty** is a subset of everything and lies in the interior of everything, while nothing that is not empty is a subset of it.

Cost. Part of the calibration of the margin.

M.5.1 intersection

Cl. 10.5.7, Cl. 12.12.6

50a0004

Defect. The intersection did not exist in the library, neither as a free function nor as a method.

Remedy. A free function: the greater of the two lower endpoints and the lesser of the two upper ones, with **Empty** when either operand is empty. The case worth a line is that of disjoint operands: if the endpoints come out crossed the result is **Empty** and not a malformed interval.

Evidence. Measured with M.5.2, and Table 6.5 gives the manual block. The test also prints the minimum and the intersection of the same two operands on one line, which is how the false friend **operator&** is documented rather than argued about.

Cost. The operation is exact and constructs one interval, so the whole surcharge is the constructor of M.2.19 inherited. It is readable on the scalar path alone, between 1.38 and 1.44; on the two vector paths the cells lie between 0.85 and 1.21, that is inside the margin.

Operands	intersection
[1,4], [2,3]	[2,3]
[1,2], [2,3]	[2,2]
[1,2], [1,2]	[1,2]
[1,2], [3,4]	[inf,-inf]
[-2,2], [4,4]	[inf,-inf]
[1,3], Entire	[1,3]
Entire, Entire	[-inf,inf]
[1,3], Empty	[inf,-inf]
Empty, Empty	[inf,-inf]

Table 6.5: The intersection of M.5.1. The second row is a contact at a single point, which is an interval and not an empty one. The two rows that follow are the disjoint operands, where the endpoints come out crossed and the answer is the canonical pair of **Empty**, **[inf,-inf]**, rather than a malformed interval. On the same two operands of the first row the false friend **operator&** answers **[1,3]**, being the minimum of the endpoints and not the intersection.

M.5.2 convexHull

Cl. 10.5.7, Cl. 12.12.6

294557d

Defect. The convex hull did not exist in the library.

Remedy. A free function: the lesser of the two lower endpoints and the greater of the two upper ones, and above all the rule for the empty interval that *differs* from that of the intersection. With one empty operand the other is returned, and only two empty operands give **Empty**. Cl. 12.12.6 makes the distinction with one word, *either* for the intersection and *both* here, and it is the only place in the library where the empty interval is not absorbing. The hull is not the union, which is why the operation exists: the union of two disjoint intervals is not an interval.

Evidence. Taken together the two entries add 83 assertions to the corpus, all of which pass, and the failures stay at 34 with the flag on and at 1494 with it off. Table 6.6 gives the manual block, whose rows with an empty operand are the ones that separate this operation from the previous.

Cost. The row that measures nothing is the interesting one: on an empty operand the function returns the other operand without constructing anything, and it still costs between 1.16 and 1.21 on all three paths. The expansion of the constructor in the other branch makes the function larger, and the path that constructs nothing carries the weight of code it never executes. It also explains why the intersection costs the same on disjoint operands as on secant ones.

M.6.1 mulRevToPair, the division with two outputs

Cl. 10.5.5, Cl. 12.12.3

ab44575

Defect. The operation did not exist, and no function of the library returned two intervals: this is the first.

Operands	convexHull
[1,2], [3,4]	[1,4]
[1,4], [2,3]	[1,4]
[1,2], [1,2]	[1,2]
[1,3], Entire	Entire
[1,3], Empty	[1,3]
Empty, [1,3]	[1,3]
Entire, Empty	Entire
Empty, Empty	Empty

Table 6.6: The convex hull of M.5.2. The first row is why the operation exists: the union of two disjoint intervals is not an interval, and the hull steps over the hole. The three rows with one empty operand return the other one, and only the last, with both empty, gives **Empty**: it is the single place in the library where the empty interval is not absorbing, and Cl. 12.12.6 marks the difference with one word, *either* for the intersection and *both* here. The empty answer is the reversed pair that Table 6.5 writes out. On the operands of the first row the false friend **operator|** answers [3,4], being the maximum of the endpoints.

Remedy. A free function returning a pair. The set it computes is given by Cl. 10.5.5 as

$$\mathbf{c} /_{\text{set}} \mathbf{b} = \{x \in \mathbb{R} \mid \exists b \in \mathbf{b} \text{ such that } xb \in \mathbf{c}\},$$

that is the reals which, multiplied by *some* element of the divisor, fall inside $\mathbf{c}$. It has zero, one or two connected components, and keeping them apart is the whole point of the operation: their hull is **Entire** and says nothing. The note of the clause warns that this is not an arithmetic operation, and that using one component alone inside an expression loses containment.

Evidence. Table 6.7 gives the manual block, whose rows run through the three possible numbers of components. The two entries of the block add 467 assertions to the corpus, of which 465 pass.

$\mathbf{b}$	$\mathbf{c}$	First component	Second component
Empty	[1,2]	Empty	Empty
[0,0]	[1,2]	Empty	Empty
[0,0]	[0,2]	Entire	Empty
[-2,-0.1]	[-2.1,-0.4]	[0.2...,21]	Empty
[0,1.1]	[-2.1,-0.4]	[-inf,-0.36...]	Empty
[-2,1.1]	[-2.1,-0.4]	[-inf,-0.36...]	[0.2...,inf]
[-1,1]	[1,2]	[-inf,-1]	[1,inf]
Entire	[1,1]	[-inf,-0]	[0,inf]

Table 6.7: The two components of M.6.1, an absent component being **Empty**, which the library writes as the reversed pair [inf,-inf]. The first two rows have no component at all, the four that follow have one, and the last three have two, which happens when the divisor has zero in its interior. The last three are also the rows that show why the operation exists: the hull of their two components is **Entire**, so an answer returned as a single interval would carry no information.

The two that do not are the outputs of the operation applied to a not-an-interval literal, where an equality between two NaN values is false by construction, and they close with G.9. It is the first time in the work that declaring an operation *adds* failures, and it is recorded rather than hidden.

Cost. The highest of its block, between 1.65 and 2.75, and the reason is arithmetically simple: the function constructs *two* intervals instead of one, so it inherits the constructor twice.

M.6.2 cancelMinus and cancelPlus

Cl. 10.5.6, Cl. 12.12.5

68ba034

Defect. Neither of the two cancellative operations existed.

Remedy. Two free functions. `cancelMinus` is the narrowest z with $y + z \supseteq x$ when one exists, and Cl. 12.12.5 turns the *when it does not* into a value, namely `Entire`. Two things follow. `Entire` must stay diagnostic, so a result that overflows must not be `Entire` as well, and rounding each endpoint in its own direction gives that for nothing. And the test for existence cannot be made in floating point: with operands below the unit of roundoff the endpoints collapse and the comparison declares the result defined both ways round, so the sign of the exact expression is decided with an exact sum, whose two-sum is valid only under round to nearest, a mode that is entered and left on the ambiguous branch alone.

Evidence. Measured with M.6.1, the two entries having been committed together, and Table 6.8 gives the manual block.

Operation	Operands	Result
<code>cancelMinus</code>	<code>[-1,5], [-1,5]</code>	<code>[-0,0]</code>
<code>cancelMinus</code>	<code>[-2,6], [-1,5]</code>	<code>[-1,1]</code>
<code>cancelMinus</code>	<code>[-1,5], [-2,6]</code>	<code>[-inf,inf]</code>
<code>cancelMinus</code>	<code>Empty, [-1,5]</code>	<code>[inf,-inf]</code>
<code>cancelMinus</code>	<code>[-1,5], Empty</code>	<code>[-inf,inf]</code>
<code>cancelMinus</code>	<code>Entire, [-1,5]</code>	<code>[-inf,inf]</code>
<code>cancelPlus</code>	<code>[-2,6], [-5,1]</code>	<code>[-1,1]</code>
<code>cancelMinus</code>	<code>[M,M], [-M,-M]</code>	<code>[1.7976...e308,inf]</code>
<code>cancelMinus</code>	<code>[-1,u], [-u/2,1]</code>	<code>[-1,-0.999999999999999888]</code>
<code>cancelMinus</code>	<code>[-1,u/2], [-u,1]</code>	<code>[-inf,inf]</code>

Table 6.8: The two cancellative operations of M.6.2, with M for `realmax` and u for the unit of roundoff. The third row is the one where no z exists, because the second operand is the wider, and Cl. 12.12.5 turns that into `Entire`. The eighth is why `Entire` has to stay diagnostic: the result overflows, and rounding each endpoint in its own direction answers `[realmax, +∞]` rather than `Entire`, so that overflowing is not confused with having no value. The last two rows are the pair on which the test for existence cannot be made in floating point: they differ by half a unit of roundoff and must answer differently, which they do only because the sign is decided by an exact sum.

Cost. On the defined shape the surcharge is 1.81, an operation of three nanoseconds

inheriting one constructor. The ambiguous shape is the one that carries the design: it costs 34 nanoseconds against the 3.3 of the fast path on the vector builds, and 133 on the scalar one, and that is the price of the three calls into the C library that change the rounding mode. The rounding register could be written directly, as M.2.20 does, but the branch is rare by design and the shape that reaches it is built for the purpose, so the figure is recorded and the change is not made.

6.3 Reverse-mode operations

G.7 was the largest block by volume, twelve operations of Table 10.1 and about sixteen hundred lines of corpus, and none of them existed.

A reverse operation recovers an argument from a result. Cl. 10.5.4 states it at Level 1 as a containment rather than as an equality, and the three equations differ only in which argument is being recovered. For a unary φ it is equation (5),

$$\varphi\text{Rev}(\mathbf{c}, \mathbf{x}) \supseteq \{x \in \mathbf{x} \mid \varphi(x) \text{ is defined and lies in } \mathbf{c}\}, \quad (6.8)$$

and for a binary $\bullet$ they are equations (6) and (7),

$$\bullet\text{Rev}_1(\mathbf{b}, \mathbf{c}, \mathbf{x}) \supseteq \{x \in \mathbf{x} \mid \exists b \in \mathbf{b} : x \bullet b \text{ is defined and lies in } \mathbf{c}\}, \quad (6.9)$$

$$\bullet\text{Rev}_2(\mathbf{a}, \mathbf{c}, \mathbf{x}) \supseteq \{x \in \mathbf{x} \mid \exists a \in \mathbf{a} : a \bullet x \text{ is defined and lies in } \mathbf{c}\}. \quad (6.10)$$

Three consequences of those three lines govern the whole block, and none of them was a choice made here. The digit in the name says *which* argument is recovered, and the two forms exist only where they differ, since the clause adds that if $\bullet$ is commutative then $\bullet\text{Rev}_1$ and $\bullet\text{Rev}_2$ agree and may be implemented as a single $\bullet\text{Rev}$: this is why `mulRev` is one operation while `powRev` and `atan2Rev` are two. The last argument $\mathbf{x}$ is optional and defaults to the whole real line; it is what the caller already knows about the argument, and it is what keeps an otherwise unbounded answer bounded. And the relation is $\supseteq$, so a valid implementation may return more than the preimage but never less, which is the licence that makes a search-based implementation legitimate at all.

Table 6.9 lists the twelve beside the forward operation each of them inverts.

The eight searches use one procedure, given as Algorithm 6.2. Two decisions inside it are what make the answer valid, and both are visible in the pseudocode. The first is that the bisection runs on the *integer that encodes* the double rather than on the value: the non-negative doubles are ordered exactly as their encodings, so sixty-three halvings exhaust the format and the invariant holds from the first step, with no starting estimate to compute. The second is that the predicate is decided by the interval evaluation of f , which is already conformant: if $f([x_{\text{lo}}, t])$ does not meet $\mathbf{c}$ then *no* point of that segment has its image in $\mathbf{c}$, and the segment is discarded with certainty rather than with an opinion. Figure 6.1 states the same argument in the other direction.

The procedure returns the last point that still misses $\mathbf{c}$, never the first that meets it, and that asymmetry is what keeps the result an enclosure. It also costs one *ulp* of tightness

Reverse	Inverts	How it is computed
sqrRev	x^2	closed form, two branches
absRev	$ x $	closed form, two branches
pownRev	x^p , p integer	closed form for $ p \leq 2$, bisection otherwise
coshRev	$\cosh x$	bisection
mulRev	ab	the hull of <code>mulRevToPair</code>
sinRev	$\sin x$	bisection, grid of Cl. 10.5.4 for the exact values
cosRev	$\cos x$	bisection, grid
tanRev	$\tan x$	bisection, grid
powRev1	a^y , on a	bisection in the logarithm
powRev2	a^y , on y	bisection in the logarithm
atan2Rev1	$\text{atan2}(y, x)$, on y	bisection
atan2Rev2	$\text{atan2}(y, x)$, on x	bisection

Table 6.9: The twelve reverse operations of Table 10.1, beside the forward operation each one inverts. The names are the ones the standard fixes and were not chosen here. Four have a closed form; the other eight are searches, and the search is the same procedure in all of them.

Algorithm 6.2: The bisection used by the eight searching reverse operations. The pair `bits` and `val` is the bijection between a non-negative double and the integer that encodes it; f is evaluated in interval arithmetic, so the test on line 7 is a proof and not an estimate.

Input: an interval $x = [x_{\text{lo}}, x_{\text{hi}}]$, a target c , a forward function f

Output: the largest t such that $f([x_{\text{lo}}, t])$ misses c

```

1: if  $f(x)$  does not meet  $c$  then
2:   return Empty                                     % no argument at all maps into the target
3: end if
4:  $\ell \leftarrow \text{bits}(x_{\text{lo}})$ ,  $h \leftarrow \text{bits}(x_{\text{hi}})$ 
5: while  $h - \ell > 1$  do
6:    $m \leftarrow \ell + \lfloor (h - \ell)/2 \rfloor$ 
7:   if  $f([x_{\text{lo}}, \text{val}(m)])$  meets  $c$  then
8:      $h \leftarrow m$                                      % the boundary is at or below  $m$ 
9:   else
10:     $\ell \leftarrow m$                                      % every point up to  $m$  can be discarded
11:   end if
12: end while
13: return  $\text{val}(\ell)$                                      % the last point that still misses
```

whenever the true boundary happens to be exactly representable, which is the defect Section 7.4 repairs.

Two things were learned here that generalise. The first is that no starting estimate may be computed in floating point: both $1/q$ and `std::pow` respect the current rounding mode, and with `FE_UPWARD` installed the estimate for one case fell one hundred and eighty *ulp* away from the true value and the corrections ran out, returning $+\infty$. The second is that refining the answer by one *ulp* at the end, which looks obviously safe, is *not*: a point may miss c and still be the limit of points that do not miss it. The naive refinement repaired

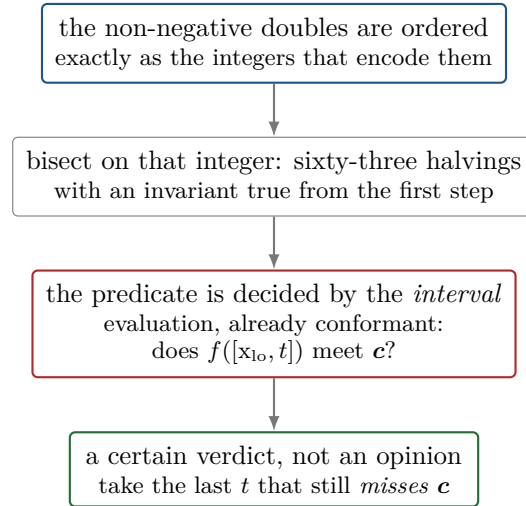


Figure 6.1: The bisection used by the reverse operations. Deciding the predicate with the interval evaluation rather than with an approximate computation is what makes the answer valid: if $f([x_{lo}, t])$ does not meet c , then *no* point of that segment has its image in c , and the segment can be discarded with certainty. Taking the last t that still misses, rather than the first that meets, is what keeps the result an enclosure.

forty-seven endpoints and broke forty-four, and the correct one needs three conditions together, two of which were dictated by cases in the corpus.

M.7.1 `sqrRev`, `absRev` and `coshRev`

Cl. 10.5.4, Cl. 12.12.4

4f2831b

Defect. None of the twelve reverse operations existed: zero occurrences of the suffix in the sources, apart from the two-output division that Cl. 10.5.5 treats separately and that M.6.1 had added.

Remedy. The three operations built on even functions, each in both arities. They go together because they share one structure: the preimage is the union of two branches symmetric about the origin, and only the non-negative branch and the image to start from change. The negative branch is the reflection of the first.

Evidence. Declaring the six keys of the plugin the corpus grows from 6066 to 6137 assertions, and all 71 new ones pass with the flag on and with it off; the totals stay at 36 and 1496, zero repaired and zero broken. Table 6.10 gives the rows of the manual block that show the structure.

Cost. The operations have a closed form and are cheap, so the inherited constructor is visible: 1.88 on `sqrRev` and 1.82 on `absRev`. The assembly says where it goes: with the flag `sqrRev` grows from 193 to 484 instructions and from two calls to seven, the branch actually executed staying the closed form. The library pays for code it does not run, which is the phenomenon of M.5.2 on a larger scale.

Call	Result
<code>sqrRev([-10,-1])</code>	Empty
<code>sqrRev([0,25])</code>	<code>[-5,5]</code>
<code>sqrRev([0,25],[-4.1,6])</code>	<code>[-4.09...,5]</code>
<code>absRev([-1.1,0])</code>	<code>[0,0]</code>
<code>absRev([1,inf],[-inf,0])</code>	<code>[-inf,-1]</code>
<code>absRev([0,1],[7,9])</code>	Empty
<code>coshRev([1,1])</code>	<code>[0,0]</code>
<code>coshRev([1,1],[1,inf])</code>	Empty
<code>sqrRev([1,4],[0,1.2])</code>	<code>[1,1.19...]</code>
<code>sqrRev([1,4])</code> intersected with <code>[0,1.2]</code>	<code>[0,1.19...]</code>

Table 6.10: The three inverses of M.7.1 on the rows that show their structure. The preimage has two branches symmetric about the origin, and the third argument cuts them: the second row keeps both, the third keeps the positive one and part of the negative, the fifth keeps the left branch alone. The first and the sixth are the two ways of answering Empty, no square being negative and the argument being disjoint from the preimage; the fourth and the seventh are where the two branches collapse onto one point. The last two rows are why the operation takes the third argument instead of leaving the caller to intersect afterwards: the answer with it is strictly narrower, the hull of the preimage being wider than the preimage.

M.7.2 pownRev

Cl. 10.5.4, Cl. 12.12.4

de2361c

Defect. The operation did not exist.

Remedy. It generalises the three of M.7.1, an even exponent behaving like the square and an odd one being injective, and adds two cases: the zero exponent, where $t^0 = 1$ for every t , so the answer is the whole argument when $1 \in \mathbf{c}$ and empty otherwise; and the negative exponent, which brings in the reciprocal. The order matters and is root first, reciprocal second: taking the reciprocal first overflows, and with $\mathbf{c} = [0, 2^{-1074}]$ and $p = -3$ the answer comes out as $5.6 \cdot 10^{102}$ instead of 2^{358} .

Evidence. The corpus grows from 6137 to 6317 assertions and 172 of the 180 new ones pass, with zero pre-existing lines broken. The eight that fail are not violations, and it is the clause that says so: Cl. 12.10.2 asks the reverse operations for *valid* and recommends *accurate*, and the eight are one *ulp* wide. M.7.5 closes six of them anyway.

Cost. Two paths that differ by three orders of magnitude, which is why both are kept in the table. With exponent three the operation bisects and costs 34 microseconds, into which the flag disappears, 1.00. With exponent two it has a closed form, costs 21 nanoseconds, and the ratio is 2.26. It is the law of the campaign inside a single entry.

M.7.3 sinRev, cosRev and tanRev

Cl. 10.5.4, Cl. 12.12.4

45b7200

Defect. None of the three periodic inverses existed. They are also the three that justify the third argument of the operation: the preimage of a periodic function is an infinite

union of branches, so without a bounding interval the answer is **Entire** whenever $\mathbf{c}$ meets the image.

Remedy. Enumerating the branches would require reducing the argument modulo π to the last *ulp*, which is a piece of numerics in itself. It is not needed: the value is the hull of the preimage inside the argument, so only its two endpoints matter, and each is found by bisection on the bit pattern, the predicate being decided by the *interval* version of the function, which gives a certain verdict where an approximate evaluation gives only an opinion. The endpoint taken is the last point that still misses $\mathbf{c}$, so π never appears in the code.

Evidence. Table 6.11 gives the rows that carry the entry. The corpus grows from 6317 to 6385 assertions and 52 of the 68 new ones pass. The sixteen that fail are the flat points, where the function does not move: the true half-width of the preimage is the square root of twice the unit of roundoff, so one *ulp* of error on the value displaces the endpoint by about $7 \cdot 10^{-9}$. No predicate built on values accurate to one *ulp* can separate them, and closing them would need π to about a hundred and six bits.

Cost. About 12 and 19 microseconds per operation, sixty-three halvings each evaluating an interval function, and the flag adds 1.05 and 1.04, that is nothing measurable.

Call	Result	What it shows
<code>sinRev([-2,-1.1])</code>	Empty	$\mathbf{c}$ outside the image
<code>sinRev([0,0])</code>	Entire	every multiple of π
<code>tanRev([-156,-12])</code>	Entire	the tangent is onto
<code>sinRev([-0,-0],[3,3.5])</code>	[3.14159265358979312, 3.14159265358979357]	the two doubles that straddle π
<code>sinRev([0,0],[-1,1])</code>	[0,0]	a flat point
<code>cosRev([1,1],[-0.1,0.1])</code>	[0,0]	a flat point

Table 6.11: The three periodic inverses of M.7.3. The fourth row is the one that carries the entry: the answer is the pair of adjacent doubles that straddle π , found by bisecting the bit pattern with the interval function as the predicate, and π appears nowhere in the code that produced it. The last two are the flat points, where the sixteen residual failures of the block live: the true half-width of the preimage is the square root of twice the unit of roundoff, so no predicate built on values accurate to one *ulp* can find its endpoints.

M.7.4 mulRev

Cl. 10.5.4, Cl. 12.12.4

0db9413

Defect. The library provided no way of recovering a factor from the product and the target interval.

Remedy. No new arithmetic: the Level 1 value is the set division of Cl. 10.5.5 followed by a hull, so the operation is M.6.1 with its two components put back together, the fourth reuse of that entry. What matters is *where* the hull is taken: the components must be intersected with the argument *before* being joined, or the answer is **Entire**.

Evidence. The corpus grows from 6385 to 6563 assertions and 177 of the 178 new ones pass.

Cost. Measured with the block; the operation inherits the two constructors of M.6.1 and adds an intersection.

M.7.5 `pownRev` with one rounding instead of two

Cl. 10.5.4, Cl. 12.10.2

abd9b04

Defect. M.7.2 handled a negative exponent by composing the preimage of the positive one with the set reciprocal. The order was right, but the chain carried *two* roundings where *tightest* admits one, and the price was measured: eight lines of the corpus one *ulp* off.

Remedy. The negative exponent no longer goes through the reciprocal. The bisection is made directly on the power with a negative exponent, which M.2.18 had made conformant over its domain, so there is one rounding; and the property that made the composed version correct is preserved, namely that no intermediate value need be representable. Since the map decreases for a negative exponent, the lower endpoint of the branch comes from the upper endpoint of *c*.

Evidence. The corpus does not move, the entry declaring nothing new, and the failures fall from 61 to 55 with the flag on: six of the eight lines are repaired. The two that remain need a root of two to a power that is not a power of two, where the oracle itself is one *ulp* wide.

Cost. 1.01 on the bisecting shape, that is nothing: the entry changes what is bisected on, not how much work there is.

M.7.6 `powRev1` and `powRev2`

Cl. 10.5.4, Cl. 12.12.4

27c87ca

Defect. Both absent. They are the two largest entries of the block, 804 lines of corpus, sixty-two per cent of it.

Remedy. `powRev1` recovers the base given the exponent and `powRev2` the exponent given the base. Neither has arithmetic of its own: both are the bisection written for the periodic inverses, with the power held fixed on the argument that is not being recovered. The domain filter has to be repeated inside the inverse, since inside `pow` it sits behind the flag while these are new names and must answer the same either way.

Evidence. The corpus grows from 6563 to 7367 assertions and 674 of the 804 new ones pass. The 130 that fail were classified one by one: 57 are *accurate*, 70 are *valid*, one has an empty *tightest*, and two are lines where *we* are narrower than the corpus, verified by sampling the direct function over two hundred thousand points. None is a violation of containment, and Cl. 12.10.2 asks these operations for *valid* alone.

Cost. 1.13 and 1.12 on the bisecting shapes, an operation of nine and three microseconds. The interesting row is the one where a Level 1 rule answers without bisecting at all: it

costs seven nanoseconds and the flag shows on it, 1.55, which is the same law seen from the other end.

M.7.7 atan2Rev1 and atan2Rev2

Cl. 10.5.4, Cl. 12.12.4

2bce4d5

Defect. Neither existed, and they are the only two operations of the block with no line of corpus at all: the suite never exercises them, so there is no oracle to read the expected answers from.

Remedy. The same bisection as the other inverses, applied to an enclosure of the image that does not depend on the flag. That is where the work is: **atan2** in TIGHT returns a *pair*, and the boolean is exactly the information needed, marking the rectangle on which the non-conformant build returns the complement rather than the hull.

Evidence. Verified by properties, there being no corpus: completeness above all, that is that no point whose image meets *c* can be missing from the answer, checked over 24,012 points per inverse against the direct function, and over 2310 more against the standard library as an *external* oracle, since comparing only against the direct function would miss an inverse and a direct function wrong together. Table 6.12 gives the answers on the pairs used for completeness.

Arguments	atan2Rev1	atan2Rev2
[0,0], [0,0], [0,0]	—	Empty
[0,0], [0,0], Entire	—	[0,inf]
[1,1], [0,0], Entire	[-9.88...e-324, 9.88...e-324]	—
[1,2], [0.1,0.5], [-5,5]	[0.10...,1.09...]	[1.83...,5]
[-2,-1], [2,3], [-5,5]	[-4.9e-324,4.37...]	Empty
[-1,1], [-0.5,0.5], [-3,3]	[-0.54...,0.54...]	[-3,3]
[0,0], [1,2], [-4,4]	[0,4]	[0,-0]

Table 6.12: The two inverses of M.7.7, which recover *y* and *x* respectively from **atan2**(*y*, *x*); a dash marks a call the block does not make. The first row is the origin, the one point outside the domain, so the preimage is empty; the second shows that the origin removed from an unbounded argument still leaves the non-negative half-line. The third answers an interval two subnormals wide, which is the narrowest the format allows and not a degenerate case. These are the pairs on which completeness was checked point by point, no corpus being available.

Cost. The only rows of the block that diverge, 1091 and 706 operands out of 4096, and they diverge because the two builds disagree inside **atan2**: it is M.2.23 inherited whole, the assembly of the two inverses being identical between the builds, nineteen instructions on all three paths. The conformant answer is both narrower and faster, 0.64 and 0.23, and checked line by line the conformant result is contained in the other in all 458 diverging lines, never the reverse and never by more than one step of representation. It is the exact converse of M.2.23: there conformance cost width, here the same conformance gains both

width and time, because an inverse can only work from a hull and the complement is not an interval.

Chapter 7

Exceptions, Decorations and the Conformance Claim

What remains of the roadmap is the part of IEEE 1788-2015 that is not arithmetic, and it is the part a library is most likely to leave undone, because nothing in a numerical test suite fails while it is missing.

Four things are required and were absent. An operation that cannot return a value has to be able to say so, which is the exception mechanism of Cl. 7.5.3 and Cl. 12.1.3. A result has to be able to carry what is known about the way it was obtained, which is the decorated type of Cl. 11 and the rule by which a decoration propagates. The four reductions of Cl. 12.12.12 have to be correctly rounded, which cannot be done by accumulating in the format they return. And an implementation has to *publish* two things rather than compute them: the tightness of each of its operations, under Cl. 12.10.3, and the claim itself with the questionnaire of Cl. 4.4.

The chapter then closes the account. G.13 is the one block that answers to no obligation and pursues tightness for its own sake, which is why it is the only one for which stopping was a judgement rather than a criterion; the audit that follows checks the standard against the library from the opposite end, sentence by sentence rather than block by block; and the last section says what was deliberately left alone, and why.

7.1 Exceptions and decorations

G.8 adds the exception mechanism of Cl. 7.5.3 and Cl. 12.1.3. The argument for its existence is that at Level 2 an operation always returns a value, so **Empty** cannot distinguish a failure from a legitimately empty result: `[]` and `[empty]` denote the empty set on purpose. The thirty-three signalling lines of the corpus, which had been failing by construction until then, all passed.

Half of that test lies outside the corpus, and it is worth saying why. A register that raised **UndefinedOperation** on every empty result would pass every line the corpus contains

and be useless. The check that a legitimate answer signals *nothing* was therefore measured as well, by turning on the corresponding key in the plugin: the corpus doubles, and the failures stay where they were.

G.9 and G.10 add the decorated type. This is the only block that adds a *type* rather than a branch, so the axis of measurement changes to the second one of Table 4.5, decorated against bare. Two headers were added and the existing header was not touched at all, which is the argument for why no flag is needed: a caller who does not include them pays nothing and sees nothing.

A decoration is a statement about what happened while the interval was being computed, and Cl. 11.2 defines five of them. Table 7.1 lists what each one asserts.

Decoration	What it asserts about f and x
com	x is non-empty, bounded and inside the domain, f is continuous at <i>every point</i> of x , and the computed interval is bounded
dac	x is non-empty and inside the domain, and the <i>restriction</i> of f to x is continuous
def	x is non-empty and inside the domain
trv	always true; it asserts nothing
ill	the object is not an interval at all: Nal

Table 7.1: The five decorations of table (10) of Cl. 11.2, in the propagation order (19), which is also the order of strength. Each one is implied by the one above it, so the list is a chain and not a set of labels.

Because they form a chain, they can be ordered:

$$\text{ill} \prec \text{trv} \prec \text{def} \prec \text{dac} \prec \text{com}, \quad (7.1)$$

and propagation through an operation is then the *min-rule* (18) of Cl. 11.6: the decoration of a result is the weakest among those of the operands and the one the operation itself earns,

$$d(f(x_1, \dots, x_n)) = \min(dv_0, d(x_1), \dots, d(x_n)), \quad (7.2)$$

where dv_0 , the *local decoration* of (17), is the only term that has to be computed: it is the strongest of the five that holds for f on the interval parts of the operands, judged as Table 7.1 states.

That is the whole of the propagation, and writing it as a minimum is not a convenience. The five decorations are declared as an enumeration whose values increase along (7.1), so the comparison operators of the language *are* the order of Cl. 11.5.3 and the min-rule is a numeric minimum: there is no table to keep in step with the standard, and no way for the two to drift apart. The same choice pays a second time in the constructor, which normalises a forbidden combination instead of asserting against it, so that a pair Cl. 11.4 forbids is not representable; and that in turn gives **setDec** for nothing, since the special cases of Cl. 11.5.2 are exactly those three.

The corpus corrected the code three times in this block, and each correction is a lesson about reading the standard rather than about programming.

Overflow is not a discontinuity. A sum whose result exceeds the largest finite double must lose `com`, not because the function is discontinuous but because the tabulated definition asks for three things at once, one of which is that the *computed* interval be bounded. The first version answered `trv`, reasoning about Weierstrass.

`com` and `dac` do not ask for the same continuity. The first asks that f be continuous at *every point* of x , the second that the *restriction* of f to x be continuous. The ceiling of $[1.1, 2]$ has a constant restriction but jumps at 2, so it is `dac`.

Touching the cut is not crossing it. For `atan2` the criterion is not whether the second operand is the zero singleton but the sign of its lower endpoint, and $-0 < 0$ is false, which is to say the zero is read as Level 1 reads it.

M.8.1 The exception register

Cl. 7.5.3, Cl. 12.1.3

049475a

Defect. The library had no signalling mechanism at all, and a caller had no way of knowing what had happened. That is what makes the entry necessary rather than decorative: at Level 2 an operation *always* returns a value, so the value alone does not distinguish a failure from a legitimate result, a constructor that fails returning `Empty`, which is also what a literal denoting the empty set returns.

Remedy. A register with the five exceptions of the standard and the operations to use it: signal, query, clear in bulk or one at a time, save and restore the whole state, install an optional handler. The form is a choice, not an obligation: Cl. 12.1.3 says in so many words that the handler mechanism is language or implementation defined, and the choice is recorded as a choice. The state is per thread.

Evidence. The manual block checks the register on itself, because at this step nothing signals yet, and Table 7.2 is the trace of those steps. Half of the test lies outside the corpus, and it is the half that matters: that a legitimate answer signals *nothing*. A register raising an undefined operation on every empty result would pass every line of the corpus and be useless. Measured by turning on the no-exception checks of the plugin, the corpus doubles from 7367 to 14,495 assertions and the failures stay at 152: zero spurious signals over some seven thousand new checks.

Cost. The cleanest control of the whole campaign. The register compiles identically in the two builds and on the three paths, five instructions to write, seven to query, seventeen to raise with the handler, and the times confirm it to the third digit in all six configurations: 0.292, 0.583 and 1.454 nanoseconds. Raising costs 0.238 nanoseconds and the optional handler 0.871, three and a half times as much, and it is an indirect call that whoever does not install one does not pay.

Call	Register afterwards	Note
<code>clearExceptions()</code>	(nothing)	
<code>signal(UndefinedOperation)</code>	Undefined	
<code>signal(UndefinedOperation)</code> again	Undefined	the flag is sticky
<code>signal(IntvlOverflow)</code>	Undefined Overflow	
<code>clearException(Undefined)</code>	Overflow	the other one stays
<code>clearExceptions()</code>	(nothing)	
<code>signal(InvalidOperand)</code>	InvalidOperand	
<code>saved = exceptionState()</code>	unchanged	
<code>clearExceptions()</code>	(nothing)	
<code>signal(IntvlPartOfNaI)</code>	PartOfNaI	
<code>restoreExceptionState(saved)</code>	InvalidOperand	the whole register at once
<code>signal(PossiblyUndefined)</code>	handler calls = 1	with a handler installed, and it sees its own flag already raised
<code>signal(UndefinedOperation)</code>	handler calls = 1	with the handler removed: the flag rises all the same

Table 7.2: The register of M.8.1 followed through the steps of the manual block. That block checks properties and does not print the state, so the trace was obtained by running the same steps again. The names of the exceptions are shortened to their first field, so that `Undefined` stands for `UndefinedOperation` and the other three read the same way. The three groups are the three questions: that a second signal does not cancel the first and that clearing one leaves the other, that the state can be saved and restored whole, and that the optional handler is called exactly once and that removing it stops the calls without stopping the flags.

M.8.2 The constructors signal

Cl. 12.12.7, Cl. 7.5.3

8f60ad7

Defect. The constructors of G.3 already returned the right value, but silently. The comment left in M.3.3 had announced exactly this step: `[]` and `[empty]` denote the empty set, `[nai]` fails, and all three answer `Empty`. Without a second channel that difference is not recoverable by the caller.

Remedy. The two constructors raise an undefined operation on the failing paths and only on those. The possibly-undefined exception turned out to be needed, against the assumption written in the gap analysis: the inf-sup branch kept only two of the four conversions, and that pair does not prove that the endpoints are in order, two literals inside the same *ulp* giving the same pair even when exchanged.

Evidence. The 33 signalling lines of the corpus all pass, the failures falling from 185 to 152 with the flag on and from 1645 to 1612 with it off, zero broken. With this entry no line of the corpus can be moved by a change to the operations any more: what remains needs the decorations. Table 7.3 is the manual block, and its second half is the one the corpus does not look at.

Cost. On the path that succeeds the constructor from numbers costs 1.01, which answers

Call	Answer	Signal
<code>numsToInterval(1,-1)</code>	Empty	Undefined
<code>numsToInterval(+inf,-inf)</code>	Empty	Undefined
<code>numsToInterval(NaN,NaN)</code>	Empty	Undefined
<code>"[+infinity]"</code>	Empty	Undefined
<code>"[1abc,2]"</code>	Empty	Undefined
<code>"[nai]"</code>	Empty	Undefined
<code>numsToInterval(1,2)</code>	[1,2]	none
<code>"[1,2]"</code>	[1,2]	none
<code>"[]"</code>	Empty	none
<code>"[empty]"</code>	Empty	none
<code>"[entire]"</code>	<code>[-inf,inf]</code>	none
<code>"[,2]"</code>	<code>[-inf,2]</code>	none
<code>"3.56?1"</code>	<code>[3.5499...,3.5700...]</code>	none
<code>"[1.0000000000000001, 1.0000000000000002]"</code>	<code>[1, 1.0000000000000002]</code>	Possibly
<code>"[1.5, 2.5]"</code>	<code>[1.5,2.5]</code>	none
<code>"[0.5, 0.75]"</code>	<code>[0.5,0.75]</code>	none

Table 7.3: The sixteen controls of the manual block of M.8.2. The names of the exceptions are shortened as in Table 7.2, **Possibly** standing for **PossiblyUndefinedOperation**. The first group fails and says so. The second is the half the corpus does not look at: every one of those answers is legitimate and none of them signals, and among them are the two literals that *denote* the empty set, which answer **Empty** exactly as the failures do and must be told from them by the second channel alone. The last group is case (b) of Cl. 12.12.7: the same inexact decimal written twice gives a pair whose order the implementation cannot decide, so the interval is returned and the possibly-undefined exception is raised, while two literals that are far apart stay silent.

the question the block asks: whoever does not fail does not pay for the mechanism. The assembly says why. Without the flag the function has two thread-local accesses, both on the branch that fails, and the private constructor has none; the five that appear with the flag belong to M.12.2, the operand check read on every construction. Six cells of this entry are among the thirteen the registry declares unreadable, and their bimodality is described in Section 8.3.

M.9.1 The decorated type, the five decorations and **Nal**

Cl. 11.2, Cl. 11.3

34b2d0b

Defect. The library had no decorations and no **Nal**. In the test bench the not-an-interval literal was approximated by an interval of **NaN** values, and the lines that use it failed because an equality between two of those is false by construction.

Remedy. Two new headers; **Interval.hpp** is not touched, which is the argument that makes a flag unnecessary here: a translation unit that does not name the new headers pays nothing and sees nothing. The order of Cl. 11.2 is put in the values of the enumeration, so that the comparisons of the language *are* those of the standard and the min-rule is a numeric minimum, with no table to keep in step. The constructor normalises instead of

asserting, so a forbidden combination is not representable, as in M.2.19, and that gives `setDec` its special cases for nothing.

Evidence. Measured with M.9.3, the block being measured once.

Cost. The type carries one byte beside two doubles, and what that costs is read in M.9.3.

M.9.2 The operations of Cl. 11.5

Cl. 11.5, Cl. 12.12.11

9e9db41

Defect. The five operations on and with decorations did not exist, nor did the predicate that recognises `Nal`.

Remedy. The five, plus `isNaI`. The comparisons needed no code, the underlying values being ordered along the propagation order. `newDec` is a reuse and not a second definition: the equation of the standard says `com` when the interval is non-empty and bounded, which is `isCommonInterval` of M.4.1 word for word. `intervalPart` of `Nal` is the first real use of the register of M.8.1: it returns `Empty` and signals, because the value alone cannot tell that failure from a genuinely empty decorated interval.

Evidence. The operations that exist only in decorated form contribute 72 lines of corpus, and the entry is what makes the four not-an-interval lines close, which had been the residue since G.6.

Cost. Between 1.01 and 1.02, that is nothing: these operations read or write one byte. Querying a decoration costs a third of a nanosecond and building one costs five, which is the only figure of the block a caller needs.

M.9.3 The propagation

Cl. 11.6, Cl. 11.7

cbfc568

Defect. With the type in place, no operation propagated a decoration: the whole of Cl. 11.6 was missing.

Remedy. The decorated extension evaluates the bare one on the interval parts and decorates the result with the min-rule, the weakest of the decorations of the operands and the local decoration, which is the only part to be computed. The margin that makes the block feasible is that Cl. 11.6 requires only that the local decoration *hold*, the note weakening it deliberately from the strongest possible: a decoration that is too weak is conformant, merely less informative.

Evidence. The corpus grows from 7367 to 8894 assertions and the failures from 152 to 166. The total alone would say the wrong thing and is taken apart: four assertions are *repaired*, the not-an-interval lines, and the eighteen new failures are not new lines but the eighteen tightness failures of M.7.3 counted again in decorated form. Zero decoration failures over some 1900 new lines. Table 7.4 gives the manual block, whose expected decorations are copied from the corpus file by file.

Call	Answer	Note
add [1,2]_com [5,7]_com	[6,9]_com	the min-rule
add [1,2]_com [5,7]_def	[6,9]_def	
add [1,2]_com [5,realmax]_com	[6,inf]_dac	it overflows, it does not jump
add empty_trv [1,2]_com	empty_trv	
add NaI [1,2]_com	NaI	it absorbs
div [-2,-1]_com [0,10]_com	[-inf,-0.09...]_trv	it touches zero
div [-200,-1]_com [2 ⁻¹⁰⁷⁴ ,10]_com	[-inf,-0.09...]_dac	it does not touch it
sqrt [1,4]_com	[1,2]_com	
sqrt [-5,25]_com	[0,5]_trv	outside the domain
sqrt [0,25]_def	[0,5]_def	the operand is the weaker one
log [0,1]_com	[-inf,0]_trv	the domain is open at zero
tan [0,0]_com	[0,0]_com	
tan entire_dac	[-inf,inf]_trv	a pole inside
tan [0,inf]_dac	[-inf,inf]_trv	
pow [0.1,0.5]_com [0,1]_com	[0.1...,1]_com	
pow [0.1,0.5]_com [-2.5,inf]_dac	[0,316.22...]_dac	
floor [1.1,2.0]_com	[1,2]_def	the image is not one point
floor [-1.2,-1.1]_com	[-2,-2]_com	no integer inside
ceil [1.1,2.0]_com	[2,2]_dac	2 is a jump
floor [-1.1,0.0]_dac	[-2,0]_def	
intersection [1,3]_com [2.1,4]_com	[2.1...,3]_trv	Cl. 11.7.1
cancelMinus entire_dac [-1,5]_def	[-inf,inf]_trv	
equal(NaI,NaI)	false	Cl. 12.12.9

Table 7.4: The manual block of M.9.3, each expected decoration copied from the corpus. The groups are the rules: the min-rule and NaI absorbing, the local decoration of the division and of sqrt, whose domain is not a rectangle, the poles recognised from the image rather than from π , and the integer functions. Two rows are the ones that corrected the code. The third says that overflow is not a discontinuity: at Level 1 nothing has happened, and com falls only because table (10) asks for three things at once, the last of which is that the *computed* interval be bounded. And the ceiling of [1.1, 2] is dac and not com, because com wants f continuous at every point of x while dac wants the restriction continuous, and the restriction here is the constant 2 although the function jumps at 2.

Cost. Measured on the second axis, decorated against bare, as Table 7.5 gives it, and the law of M.2.25 holds here too: the cost of a decoration grows the cheaper the bare operation is, from 1.10 on tan, which costs 93 nanoseconds bare, to 3.27 on the addition, which costs three. ceil is the exception at 5.32, and the cause is structural rather than arithmetic: the local decoration evaluates the bare function *twice*, on the operand and on the operand widened by one *ulp*, to tell com from dac, and its assembly goes from 91 instructions and one call to 491 and eight. The floor is set by the decorated conversion of M.10.1, 1.54 even though it is the identity: no decorated operation can cost less than about one and a half times its bare one, which is the price of one byte beside two doubles.

Operation	Path	Bare	Decorated	Ratio
add	AVX2	3.273	10.703	3.27
	SSE2	3.244	11.206	3.45
	scalar	5.369	25.725	4.79
mul	AVX2	6.659	20.285	3.05
	SSE2	20.388	40.916	2.01
	scalar	30.436	54.887	1.80
div	AVX2	8.760	18.854	2.15
	SSE2	17.826	30.275	1.70
	scalar	26.082	43.253	1.66
sqrt	AVX2	9.075	15.266	1.68
	SSE2	9.335	15.612	1.67
	scalar	30.338	39.243	1.29
tan	AVX2	92.644	102.184	1.10
	SSE2	93.035	102.381	1.10
	scalar	94.132	111.183	1.18
ceil	AVX2	3.882	20.663	5.32
	SSE2	6.559	25.885	3.95
	scalar	7.929	32.579	4.11

Table 7.5: The second axis of the campaign, in nanoseconds per operation. What is put side by side here is not the two builds but the bare and the decorated version of the same operation in the build with the flag, since the question is what decorating costs rather than what the flag costs, and the two rows of a pair share their operand: the addition, the product and `sqrt` on two finite intervals, the division on a finite one and a divisor of definite sign, `tan` and `ceil` on a finite one. The six pairs cover the three ways in which the local decoration is computed: the common one for the operations defined on the whole plane, the private one of the division and of `sqrt`, whose domain is not a rectangle, and the one of the integer functions. The divergence count is zero on all eighteen cells and is not deduced from the code: the two rows of each pair were compared operand by operand, and on all 4096 the decorated version returns the same interval part as the bare one, which is the condition under which the ratio measures a decoration and not a difference in the work done.

M.9.4 The decorated reverse operations, and the one that only looks like them

Cl. 11.7.1, Cl. 10.5.5

20ae9cc

Defect. The twelve reverse operations had no decorated form, so the corpus could not exercise the decorated part of the library at all: the plugin turns them on with a single switch.

Remedy. The twelve receive the trivial decoration, one line each. It is the step that shows why Cl. 11.7.1 states the rule *by family*: the twelve cost seven entries and differ deeply in how they compute, yet their decoration is the same, none of them being the interval extension of a point function. The two-output division is *not* in that clause, which lists three others, so it takes the min-rule with its first argument treated as a divisor.

Evidence. Measured with M.9.3. Without this entry no figure of the block could have

been obtained at all, which is why it exists. Table 7.6 is the manual block, and its last three rows are the two-output division, which is where the family ends.

Operation	Arguments	First output	Second output
sqrRev	[0,25]_def	[-5,5]_trv	—
sqrRev	[-1,25]_dac	[-5,5]_trv	—
absRev	[1.1,2.1]_com	[-2.1...,2.1...]_trv	—
sqrRev	Nal	Nal	—
mulRevToPair	[-2,-0.1]_com [-2.1,-0.4]_com	[0.2...,21]_com	empty_trv
mulRevToPair	[-2,0]_dac [-2.1,-0.4]_com	[0.2...,inf]_trv	empty_trv
mulRevToPair	Nal [-2.1,-0.4]_com	Nal	Nal

Table 7.6: The manual block of M.9.4. The two-output division takes two arguments, written one below the other, of which the first is the divisor, and it is the only operation here that answers with a pair, which is why the last two columns exist. The first four rows say the whole of the family rule: a `com` input does not buy a `com` output, because a reverse operation is not the interval extension of a point function and the min-rule has nothing to propagate, while `Nal` goes through as it does everywhere. The last three are the two-output division, which Cl. 11.7.1 does not list and which therefore takes the min-rule with its first argument treated as a *divisor*: a divisor that touches zero costs everything, exactly as in the division itself, and the empty component is `trv` whatever the other one does, Cl. 11.4 granting the empty interval no other decoration. Writing `trv` for both, by analogy with the reverse operations beside it, would have been wrong on every row whose answer is bounded.

Cost. One line per operation, against operations that cost microseconds: nothing measurable.

M.9.5 atan2 decorated

Cl. 11.6, Cl. 11.2

d6a89ae

Defect. `atan2` had been left out of M.9.3, its bare version returning a pair.

Remedy. It is also the only operation in which domain and continuity are two distinct questions with three distinct answers, and the corpus asks all three. The rectangle that contains the origin has no value there and gets the trivial decoration; the one that touches the cut without crossing it is continuous on its restriction; and the criterion is not whether the second operand is the zero singleton but the sign of its lower endpoint, so that the negative zero is read as Level 1 reads it.

Evidence. Declaring the operation, the corpus grows from 8894 to 9288 assertions and the failures stay at 166: all 394 lines pass, bare and decorated. TIGHT already knew how to compute them from M.2.23; what was missing was the decoration. Table 7.7 is the manual block, and its three regimes are the three answers.

Call	Answer	Note
atan2 entire_dac [-2,-0.1]_com	[-pi,pi]_def	it crosses the cut
atan2 entire_dac [0.1,1]_com	[-pi/2,pi/2]_dac	far from the cut
atan2 entire_dac [-2,0]_dac	[-pi,pi]_trv	the origin is inside
atan2 [0,0]_com [-2,-0.1]_dac	[pi,pi]_dac	it lies in the cut
atan2 [-0,1]_com [-2,-0.1]_com	[1.67...,pi]_dac	it only touches it
atan2 [-2,0]_com [-2,-0.1]_dac	[-pi,pi]_def	it crosses it
atan2 [0,0]_com [0.1,1]_com	[0,0]_com	
atan2 empty_trv entire_dac	empty_trv	
atan2 NaI entire_dac	NaI	

Table 7.7: The manual block of M.9.5, where the endpoints written π and $\pi/2$ are the two `double` values that straddle them, as the corpus writes them. The first group is the three questions the corpus asks and the three different answers: a rectangle that crosses the cut along the negative axis is defined everywhere on itself but jumps from π to $-\pi$, so `def` and not `dac`; one that contains the origin has no value there and gets `trv`; one far from the cut keeps `dac`. The second group is the pair that corrected the code. The criterion is not whether y is the zero singleton, which the fifth row is not, but the *sign of the lower endpoint* of y : as long as y stays at zero or above, the restriction is continuous, and that reading treats -0 as Level 1 treats it, since $-0 < 0$ is false.

Cost. Measured with M.9.3. The entry adds no divergence of its own: the 1172 diverging operands are exactly those of the bare version.

7.2 The four reductions, and why they must be exact

Cl. 12.12.12 asks, for the parent format of every 754-conforming interval type, for four reductions over vectors of `binary64`, namely `sum`, `dot`, `sumSquare` and `sumAbs`, and they must be *correctly rounded*. They are the only requirement of Cl. 12.12 that is not an interval operation at all, and they concern this work because TIGHT claims conformance to IEEE 754-2019 through Cl. 12.6.1. None of them existed, and the corpus file that exercises them was executing *zero* assertions, because the plugin did not declare the four operations and the generator therefore skipped every case.

One line of that file states the difficulty better than any argument:

$$(2^{52} + 1) \cdot (2^{52} - 1) + 2^{104} \cdot (-1) = -1, \quad (7.3)$$

where every running sum in `binary64` answers zero: the exact product is $2^{104} - 1$, which is not representable, and rounding it to the nearest double gives 2^{104} , which the second term then cancels exactly. Correct rounding of the result is thus not a property of the last step but of all of them together, and it cannot be obtained by rounding as one goes.

The accumulation is therefore carried out in *fixed point* over the whole exponent range of the format, which is the accumulator of Kulisch [KM86]. Writing each term as an integer significand times a power of two, $t_i = m_i 2^{e_i}$ with $|m_i| < 2^{106}$ for a product of two doubles,

the sum is the integer

$$S = \sum_i m_i 2^{e_i - B}, \quad B = -2148, \quad (7.4)$$

held in a fixed array of 64-bit words: every addition into it is exact by construction, and rounding happens *once*, at the end, in the current direction. Algorithm 7.1 is the procedure.

Algorithm 7.1: The exact reduction. Special values are settled before any arithmetic, because a NaN or two infinities of opposite sign determine the answer regardless of the other terms; for `sumAbs` and `sumSquare` an infinity *wins* over a NaN, since there every term is non-negative and nothing can cancel it.

Input: a vector of terms $t_1, \dots, t_n$, a rounding direction

Output: the correctly rounded value of $\sum_i t_i$

- 1: settle the special values and return if one of them decides the answer
 - 2: $A \leftarrow 0$ % the fixed-point accumulator, one bit per exponent
 - 3: **for** $i \leftarrow 1, n$ **do**
 - 4: $(m_i, e_i) \leftarrow$ the integer significand and exponent of t_i % exact
 - 5: add m_i into A shifted by $e_i - B$, propagating the carries
 - 6: **end for**
 - 7: $(h, r, s) \leftarrow$ the leading bits of A , the rounding bit, and whether anything below it is non-zero
 - 8: **return** the double built from h , rounded once using r and s
-

Two alternatives were tried and rejected, and each for a numerical reason rather than for taste. An error-free transformation, `2Sum`, which the library already had from `M.6.2`, gives an exact expansion in the sense of Shewchuk [She97], but leaves the final rounding to be extracted from it, which is the hard half of the problem and not the easy one. And `long double` is not enough: the exact product of two doubles needs 106 bits of significand and it has 64. The products are consequently formed from the integer significands, $53 \times 53 \rightarrow 106$ bits, so that even a product that underflows to subnormal stays exact, and without `__int128`, which one of the supported compilers does not have.

Overflow, underflow and the sign of an exact zero are settled as 754 Cl. 7.4 and 754 Cl. 7.5 require, which is the same reading that `N.9` established for the kernel: a directed mode never passes to infinity on the side it is rounding away from, and the subnormals come out of the same truncation of the significand that serves everything else rather than out of a special case. An exact zero is `+0` in every direction except `roundTowardNegative`.

One mistake is recorded here because the way it showed is worth more than the mistake. The first version *lost the value one entirely*. One falls on bit 2096 of the accumulator, where the low word of the shifted significand is zero and the high word is not, and the carry loop stopped at the first zero word. Every test on the special values passed while every arithmetic test failed, and it is that profile, special values sound and arithmetic uniformly wrong, that points at the accumulator rather than at the rounding. The oracle was kept independent: with integer terms below 2^{20} the exact sum and dot product fit in

a 64-bit integer, so the reference is computed in integers and not by another floating-point accumulation, which would have compared two conjectures.

The audit of Section 7.5 later found that these reductions must also raise the flags of IEEE 754-2019, which they did not; M.10.6 added them. Three of the four arise on paths the accumulator leaves before any arithmetic, that is invalid operation on infinities of opposite sign, on $\infty \times 0$ and on a signalling NaN; only inexact and underflow come from the rounding itself, where the bits r and s of Algorithm 7.1 already answer the question *is this exact*. Underflow is tiny *and* inexact together, so an exact subnormal raises nothing.

M.10.1 The decorated constants and the type conversion

Cl. 12.12.1, Cl. 12.12.10

4bc9b40

Defect. The bare constants had existed since M.1.1; their decorated versions did not, and the type conversion existed in neither form.

Remedy. Two functions and the two conversions. They are functions and not constant objects, and the reason is one the work had already paid for: a constant of namespace scope has internal linkage and dynamic initialisation per translation unit, which is exactly what broke the test bench in August. The clause also calls them constant functions.

Evidence. Measured with the block. The `com` to `dac` branch of the conversion cannot fire, and the entry says why rather than writing an unreachable line: Cl. 11.4 already forbids `com` on an unbounded interval, which is the third time that reading proved necessary.

Cost. The conversion with a single type is the identity, and its decorated form is what sets the floor quoted in M.9.3: 1.54 to 1.70, the price of carrying one byte beside two doubles.

M.10.2 The decorated constructor from numbers

Cl. 12.12.7, Cl. 10.2

99e9af7

Defect. The decorated constructors of Cl. 12.12.7 did not exist, and the corpus has a hundred lines for them.

Remedy. One function. On the name: the prefixes of the standard are notation and not names, the two versions differing only in return type, so the suffix follows `newDec` and `setDec`, whose names the standard does give. Whether the construction failed is read from the value here, and the reason is written down: Cl. 10.2 makes an empty result impossible on success.

Evidence. The manual block carries all nine lines of the corpus, copied and not deduced, and Table 7.8 gives them. Half of the test lies outside the corpus here as well: that a call which *succeeds* signals nothing.

Cost. Measured with the block; it is the bare constructor of M.3.1 plus one decoration, 3.977 nanoseconds bare against 7.520 decorated, that is from 1.34 to 1.89. The ratio is above the floor of M.10.1, and the difference is the work proper to the entry: the decorated version has to decide whether the construction succeeded.

Call	Answer	Signal
numsToIntervalDec(-1,1)	[-1,1]_com	none
numsToIntervalDec(-inf,1)	[-inf,1]_dac	none
numsToIntervalDec(-1,inf)	[-1,inf]_dac	none
numsToIntervalDec(-inf,inf)	[-inf,inf]_dac	none
numsToIntervalDec(2,1)	Nal	Undefined
numsToIntervalDec(1,-1)	Nal	Undefined
numsToIntervalDec(NaN,NaN)	Nal	Undefined
numsToIntervalDec(-inf,-inf)	Nal	Undefined
numsToIntervalDec(inf,inf)	Nal	Undefined

Table 7.8: The nine lines of the corpus for M.10.2. The last word of the clause is the whole difference between the two constructors: where the bare one answers **Empty** the decorated one answers **Nal**, because the empty interval is a legitimate interval and **Nal** is not an interval at all. The first group also shows why an unbounded argument costs **dac** and not **com**, and the signal is the one the bare constructor already raises from M.8.2, the note of Cl. 12.12.7 adding that it will normally be ignored, **Nal** being information enough.

M.10.3 The decorated constructor from text

Cl. 12.11.4, Cl. 12.12.7

b99bb32

Defect. The decorated literal, with its suffix, was not accepted, and it is ninety-one of the hundred lines.

Remedy. The two clauses are read at different levels, and the corpus shows it with two opposite lines: a literal whose interval overflows still costs only **com**, while a literal naming a forbidden combination has no value and is rejected. The permitted combination of Cl. 11.4 is judged on the Level 1 value, whereas overflow is Level 2. Whether the interval is unbounded is a syntactic question and is asked of the literal, in one place only: when **com** is requested and the computed interval is unbounded. The failure of the text constructor cannot be read from its value, **Empty** being a legitimate answer, so this is the second real use of the register of M.8.1.

Evidence. Together with M.10.2 and M.10.4 the corpus grows from 9288 to 9433 assertions, and the hundred constructor lines and the thirty signalling checks all pass. Table 7.9 is fifteen of the thirty-seven rows of the manual block, one for each different decision.

Cost. Measured with the block, and the decoration costs 22.8 nanoseconds on a constructor that costs hundreds.

M.10.4 The four reductions, correctly rounded

Cl. 12.12.12, Cl. 12.6.1

274bfc5

Defect. The four reductions did not exist, and the corpus file that tests them ran *zero* assertions, the plugin not declaring them, so the generator skipped every case. They are required of a 754-conforming type, which is the claim the whole analysis of NFG rests on.

Literal	Answer	Signal	Note
"[]"	empty_trv	none	
"[entire]"	[-inf,inf]_dac	none	
"[-1.0,1.0]"	[-1,1]_com	none	no suffix: it is newDec
"[1,1E3]_COM"	[1,1000]_com	none	case does not matter
"[-4/2, 10/5]_com"	[-2,2]_com	none	
"10?3_com"	[7,13]_com	none	
"2.5??u_def"	[2.5,inf]_def	none	
"[1.0E+400]_com"	[realmax,inf]_dac	none	it overflows: com falls to dac
"10?3e380_com"	[realmax,inf]_dac	none	the row the flag decides
"[-1.0,]_com"	Nal	Undefined	x unbounded with com
"[-1.0 , 1.0]_fooo"	Nal	Undefined	no such suffix
"[nai]"	Nal	none	a literal: it does not signal
"[Nai]_trv"	Nal	Undefined	but with a suffix it is not one
"[1.0,2.0]"	Nal	Undefined	not a literal at all
"[1.00000000000000002, 1.00000000000000001]"	[1,1.00000000000000002]_com	Possibly	the order cannot be decided

Table 7.9: Fifteen of the thirty-seven rows of the manual block of M.10.3, one for each decision the parser has to take, with `realmax` for the largest finite double. The two clauses are read at different levels and the two middle rows are where that shows: a literal whose interval overflows keeps its value and only loses `com`, which is Level 2, while a literal naming a combination Cl. 11.4 forbids has no Level 1 value at all and is rejected. Whether x is unbounded is a syntactic question asked of the literal and not of the computed interval, in one place only, which is when `com` is requested. The two `Nal` rows in the middle of the last group are the pair worth reading together: `[nai]` is a literal and returns `Nal` without signalling, while the same word with a suffix is not a literal and therefore fails. The last literal, written across two lines, is the one case where the order cannot be decided, as in Table 7.3.

Remedy. A new header, on the principle of the decorated ones: whoever does not name it pays nothing. The accumulation is exact, in fixed point over the whole exponent range, and one line of the corpus justifies it better than any argument: the dot product of two vectors whose exact value is -1 and which any running sum of doubles flattens to zero. The products are formed from the integer significands, 53 by 53 into 106 bits.

Evidence. The fifteen new assertions pass, and the oracle is independent: with integer terms below 2^{20} the exact sum and dot product fit in a 64-bit integer, so the result is not compared against another floating-point guess.

Cost. The ratio against the naive sum stood in the hundreds, above the threshold at which the work acts, so the assembly was read. The cause was a single loop scanning the accumulator one bit at a time, about 2100 turns where one per word does 33; rewritten, a reduction of one term falls from 1423 to 148 nanoseconds and one of sixteen from 1644 to 374, while a reduction of 4096 terms does not move, which is the control that the cost removed was the fixed one. What the remaining ratio measures is the vectorisation of the naive sum rather than the slowness of the accumulator, which costs the same on all three paths because it does not vectorise.

M.10.5 The sign of zero in the numeric functions

Cl. 12.12.8, Cl. 14.4

b0b3769

Defect. The lower endpoint of the zero singleton came back as a positive zero where Cl. 12.12.8 requires a negative one, and it was not one case but five, across four functions. *The corpus cannot see this:* it compares by value, and the two zeros are equal. It took Cl. 14.4, where the canonical representative of that interval is written with the two signed zeros, to make the bit observable at all, and it is the second real defect beyond the reach of the bench.

Remedy. The rule is applied in the seven functions that carry the names of the standard and *not* in the accessors of the representation. Those have 174 internal calls and feed the correctly rounded power, where the sign of a zero base produced the malformed result of M.2.12. The distinction is the one Cl. 14.2 itself makes: the accessors are Level 3, the named operations are Level 2.

Evidence. The invertibility required by Cl. 14.4 gives the property to check, and it is an identity of *datums* and not of bits, so the comparison is written on the endpoints rather than with the equality operator, which would answer differently in the two builds. Table 7.10 names the five one by one, so that a regression says *which*, and carries with them the half of the test that could have been left out: that nothing other than a zero is touched, since a normalisation firing too often would pass every positive check there is.

Cost. The entry is why the seven named functions do not forward for nothing: the standard name costs 0.584 nanoseconds against the 0.494 of the accessor it forwards to, eight instructions against three, and the difference is the comparison against zero, four instructions for the parity flag. It is the fourth appearance of that pattern in the work.

M.10.6 The 754 flags in the reductions

Cl. 12.12.12, 754 Cl. 7.2

1c52037

Defect. M.10.4 gave the right values and raised no flag at all: seven checks out of eleven failed on the first measurement.

Call	Before	Now
<code>inf([0,0])</code>	+0	-0
<code>sup([-1,-0])</code>	-0	+0
<code>sup([-0,-0])</code>	-0	+0
<code>mid([-0,-0])</code>	-0	+0
<code>mag([-0,-0])</code>	-0	+0
<code>inf([-0,-0])</code>	—	-0
<code>inf([-0,1])</code>	—	-0
<code>inf([0,1])</code>	—	-0
<code>sup([-1,0])</code>	—	+0
<code>wid([0,0])</code>	—	+0
<code>rad([0,0])</code>	—	+0
<code>mig([-0,1])</code>	—	+0
<code>lower([0,0])</code>	—	+0
<code>inf([-1,2])</code>	—	-1
<code>sup([-1,2])</code>	—	2

Table 7.10: The rule of Cl. 12.12.8: a Level 1 value of zero comes back as -0 from `inf` and as $+0$ from every other function of that subclause. The first group is the five calls that were wrong, out of the thirteen tried; the second group is there for what those calls must answer and not for what they used to answer, which is why its first column is empty. The last group is the distinction the entry rests on. `lower` is the Level 3 accessor of the representation and is deliberately *not* normalised, it having 174 internal calls and feeding the correctly rounded power, where the sign of a zero base produced the malformed result of M.2.12; and the last two rows are the half of the test that says nothing which is not a zero is touched.

Remedy. All in the same header, and the rounding flags in one place: the two bits that decide the rounding *are* the question whether the result is exact, already computed there, so there is nothing to recompute. Reading 754 Cl. 9.4 widened the repair, which is why it was read rather than assumed: three of the four flags arise on paths the reduction left before doing any arithmetic, namely an invalid operation on discordant infinities, on $\infty \times 0$ and on a signalling NaN. Underflow is tiny *and* inexact, so an exact subnormal raises nothing.

Evidence. Table 7.11 is the manual block, and it is read in both directions, the flag that must rise and the one that must not. The two standards disagree in two places and both are resolved in favour of IEEE 1788-2015: 754 Cl. 9.4 leaves inexactness unspecified for its own reductions, which are an implementation-defined approximation, while Cl. 12.12.12 requires them correctly rounded; and the sign of an exact zero under rounding toward negative is legislated by IEEE 1788-2015. The corpus stays at 9433 assertions and 166 failures, zero repaired and zero broken.

Cost. Not measured separately: the flags are raised on paths the reduction already takes, and the entry adds no loop.

Call	Result	Flags raised	Note
sum {1,2,4,8}	15	none	exact
sum {1,2 ⁻⁶⁰ }	1	Inexact	
sum {realmax,realmax}	+inf	Overflow Inexact	
dot {2 ⁻¹⁰³⁰ ,2 ⁻⁵⁴⁵ }{1,2 ⁻⁵⁴⁵ }	8.69e-311	Underflow Inexact	
dot {2 ⁻⁶⁰⁰ } {2 ⁻⁶⁰⁰ }	0	Underflow Inexact	
sum {2 ⁻¹⁰⁷⁴ ,2 ⁻¹⁰⁷³ }	1.48e-323	none	an exact subnormal
sum {+inf,-inf}	NaN	Invalid	
sum {+inf,+inf}	+inf	none	
dot {0,1} {+inf,1}	NaN	Invalid	infinity times zero
sum {quiet NaN,1}	NaN	none	
sum {}	0	none	the empty vector, 754 Cl. 9.4

Table 7.11: The manual block of M.10.6, read in both directions: half of it is the flag that must rise and half the flag that must not. The sixth row is the whole of 754 Cl. 7.5 in one line, since underflow is tiny *and* inexact together and an exact subnormal therefore raises nothing, and the two rows above it are the same quantity reached from the two sides. The second group is the invalid operation, which arises on paths the reduction leaves before doing any arithmetic: discordant infinities and $\infty \times 0$ have no value, whereas two infinities of the same sign and a quiet NaN arriving as a term are answers and not faults.

7.3 The conformance claim

G.12 was reformulated twice before it was closed, and the reformulation is the interesting part. The block was described as *naming conventions and conformance claim*; Cl. 12.2, the clause on names, contains no *shall* at all, so the aliases are a convenience and not conformance. Meanwhile Cl. 4.2, which lists what a set-based implementation must satisfy, refers on to Cl. 14.1 and Cl. 14.2, and the questionnaire of Cl. 4.4 asks for an account of Cl. 14.3 and Cl. 14.4. Three obligations that no line of the roadmap named came out of that: an interchange encoding, which is due in force of the 754 claim itself; a means for the user to ask that `InvalidOperand` be signalled; and the documentation of the tightness of every operation.

The interchange encoding of Cl. 14.4 brings its own property to verify, and it is a strong one: the encoding must be exactly invertible on the datums. It is also what made the sign of a zero observable, and so what exposed M.10.5, one of the four defects the corpus could not see, listed in Table 4.4.

M.12.1 The names of the standard, as a forwarding layer

Cl. 12.2, Cl. 12.12

6620195

Defect. Which names were missing was *measured* rather than recalled: of the operations of the four tables, fifteen had no declaration for the bare type under the name the standard gives.

Remedy. A new header that nobody includes. The layer computes nothing: each new name is one line forwarding to the function that already exists.

Evidence. That the layer does nothing is the whole claim, so recomputing an expected value by hand would have verified the arithmetic a second time and the alias not at all. It is verified instead by comparing the assembly: identical line by line to that of the function forwarded to, in all six configurations with the addresses normalised. Table 7.12 puts the two spellings side by side on the values, which is where the two rows that do *not* answer the same become visible.

Name	Call	Answer	Forwards to	Its answer
add	[1,2] [-3,-1]	[-2,1]	a+b	identical
div	[1,2] [-3,-1]	[-2,-0.33333...]	a/b	identical
sup	[1,2]	2	upper	identical
mag	[-3,-1]	3	magnitude	identical
inf	[0,0]	-0	lower	0
equal	[1,2] [1,2]	true	a==b	true
isMember	1.5 [1,2]	true	—	—

Table 7.12: Seven of the thirteen rows of the manual block of M.12.1, themselves chosen among 740 comparisons, all of which agree. On the six rows not shown the last column reads *identical* as it does on the first four here, and that is the claim of the entry: the layer computes nothing. The two rows in the middle are the exceptions and neither is a defect. *inf* is the only name whose forward is not bit for bit, and the difference is the rule of M.10.5, the Level 2 operation normalising the sign of a zero where the Level 3 accessor does not. The answers are those of the build with the flag; without it *equal* answers *false* on both columns, which is the row that carries its 2048 divergences. *isMember* closes the table because it forwards to nothing: the name of the standard is the only spelling the library has for it, from M.2.21.

Cost. Exactly what an inlined forward should cost: 3.17 against 3.22 nanoseconds for the addition, and the times agree to the third digit. The seven numeric names are the exception and carry M.10.5, which is why they cost 0.584 against 0.494.

M.12.2 Level 3 objects that are not valid

Cl. 14.3, Cl. 14.2

d32be02

Defect. Cl. 14.3 requires an implementation to provide the user with a means of asking that an invalid operand be signalled. The exception was already in the enumeration from G.8: there was the slot and not the means.

Remedy. Three new names, and a per-thread switch that is off by default. Which objects are not datums is not a matter of taste: the representation is the inf-sup form of Cl. 14.2, so the pairs that denote nothing are the reversed ones other than the empty interval, and those with a NaN endpoint. The check runs after the repair of M.2.19, an infinity arriving from an overflow not being an invalid operand.

Evidence. The manual test uses the pair of NaN values, which is the only malformed object the constructor lets through in every build: a reversed pair aborts a build with the assertions alive. With the switch off nothing happens; with it on the register of M.8.1 records the invalid operand and the object proceeds unchanged, the clause constraining the *signal* and not the result.

Cost. Measured with M.2.19, the two entries living in the same constructor, and the cost of this one *is* the difference between the two conformant columns: 1.8 times on the isolated constructor, that is one thread-local read costing nearly twice the two comparisons of M.2.19, and between 1.6 and 2.6 per cent inside a call to `exp`. With the switch off it is one instruction, and whoever turns it on chooses to pay.

M.12.3 The interchange encoding

Cl. 14.4, Cl. 14.2

4ab9788

Defect. Cl. 14.4 requires a 754-conforming implementation to provide an interchange encoding, and it is one of the three obligations no line of the roadmap had named. It is owed by force of the 754 claim, which is the thread the analysis of NFG hangs from.

Remedy. A new header: sixteen octets for a bare interval, seventeen for a decorated one, the decoration being one octet with the values of the clause. The normalisation of zero is done here and not borrowed from the named accessor, because correcting that one is a change to an existing name and sits behind the flag, while this is a new name and must answer the same in both builds. The byte order has no default, and that is deliberate: the clause says the parameters shall be communicated, and a default would put a silent assumption where an explicit agreement is needed.

Evidence. The clause gives the property for nothing: the encoding must be exactly invertible on datums. It is an identity of *datums* and not of bits, the zero singleton coming back with a negative lower endpoint, so the comparison is written on the endpoints. Table 7.13 is twelve cases of the manual block.

Cost. Encoding reads and decoding builds, which the figures say plainly: 1.02 to encode and 1.36 to decode a decorated interval, the latter being the inherited constructor on the scale of ten nanoseconds.

M.12.4 The documentation of tightness

Cl. 12.10.3, Cl. 12.10.2

1407f73

Defect. Cl. 12.10.3 requires an implementation to *document* the tightness of each of its interval operations. It is an obligation like any other, and it is discharged with a document rather than with code.

Remedy. A section of `CONFORMANCE.md` in the fork. The point of method is that measurement and argument are kept apart, and a conformance document is worth exactly that distinction: the measurement is the corpus, whose expected values are the *tightest* ones, and for every row of the table the number of assertions is given, so that a reader can tell a claim supported by nine thousand lines from one supported by three.

Datum	Octets, big-endian	Read back
[1,2]	3F F0 .. 40 00	the same
[empty]	7F F0 .. FF F0	the same
[entire]	FF F0 .. 7F F0	the same
[0,0]	80 00 .. 00 00	[-0,0], the same datum
[-inf,2]	FF F0 .. 40 00	the same
[u,2u]	00 00 .. 00 00	the same
[-1,3]_com	BF F0 .. 40 08 10	the example of Cl. 14.4
[nai]	7F F8 .. 7F F8 00	the same
[-1,3]_com, little	00 00 00 ... 10	the fields are reversed, not the octet
a reversed pair	40 00 .. 3F F0	Empty and InvalidOperand
sound octets	3F F0 .. 40 00	[1,2], and no signal

Table 7.13: Twelve cases of the manual block of M.12.3, with u for the smallest subnormal. Of the octets only the first two of each endpoint are shown, and for the decorated type the one that carries the decoration. The first eight rows encode a datum and read it back, and the fourth is the one that shows what the clause means by exactly invertible: [0,0] comes back with a negative lower endpoint, so it is an identity of *datums* and not of bits, which is why the check is written on the endpoints and not with the equality operator. The ninth row is the same datum in the other byte order, where what is reversed is the order of the fields and not the octet. The last two start from the octets instead, with the switch of M.12.2 on, which is the condition Cl. 14.3 sets for the signal to be raised at all.

Evidence. Nothing executable: the verification *is* the measurement, and it is the one the registry reports block by block. The two inverse operations with no line of corpus are documented as verified by properties, with the external oracle named.

Cost. None: the entry adds no code.

M.12.5 The claim and the questionnaire

Cl. 4.3, Cl. 4.4

45eb3a4

Defect. The library made no conformance claim, which is what Cl. 4.3 provides for and Cl. 4.4 asks to be answered.

Remedy. The first two sections of the same document. The claim is for the set-based flavor with 754 conformance for the inf-sup type derived from **binary64**, without compressed arithmetic and without non-standard flavors. The questionnaire answers twenty-one questions; two of its parts do not arise. The operation of Cl. 10.6.4 is declared *not provided*, that clause being among the recommended ones, and declaring it is itself part of answering the questionnaire.

Evidence. The claim is the point at which the work becomes checkable by someone else, and it is why the audit of Section 7.5 was done afterwards: closing the thirteen blocks does not prove coverage, it proves that the blocks are closed.

Cost. None: the entry adds no code.

7.4 Tightness of the reverse operations

G.13 is unlike the other twelve, and the difference had to be faced rather than glossed over: it closes no *shall*. Cl. 12.10.2 asks of the reverse operations only that they be *valid*, and recommends *accurate*. Work on tightness is therefore done because tightness is the declared purpose of the library, not because the standard demands it, and this reverses the acceptance criterion. In the other blocks an error would have cost conformance; here an error *removes* containment, since the code is making the answer smaller. Every step of the block accordingly carries a sampled completeness check alongside the comparison with the corpus, and it was one of those checks that caught the only defect this work introduced.

Four changes took the residue from one hundred and sixty-six failing lines to twenty-six. Three of them carry the argument, and each is a rule that replaces a search.

Refining the answer by one *ulp*, and why it is almost always wrong. Algorithm 6.2 returns the last point that misses $\mathbf{c}$, so it overshoots by exactly one *ulp* whenever the true boundary is representable, and the boundaries the corpus asks about are full of exact values: `powRev1([2,4],[1,1])` is $[1,1]$, and the library answered $[0.9999999999999999, 1.0000000000000002]$.

The obvious repair is to drop the endpoint when the *point* itself misses $\mathbf{c}$. It is not sound, and the reason is that a point may miss $\mathbf{c}$ and still be the limit of points that do not: measured on the corpus, the naive refinement repaired forty-seven endpoints and broke forty-four. Let a be the last point that misses and a' the next double. The boundary equals a' only if no real of the open interval (a, a') lies in the preimage, and where f is monotone across one *ulp* the image of that open interval lies between $f(a)$ and $f(a')$. Writing γ for the endpoint of $\mathbf{c}$ that is met, three conditions are needed *together*:

$$f(a') \cap \mathbf{c} = \{\gamma\} \text{ and } \gamma \text{ is an endpoint of } f(a'), \quad (7.5)$$

$$f(a) \text{ lies strictly beyond } \gamma, \quad (7.6)$$

$$\text{the endpoint of the image that meets } \mathbf{c} \text{ moves beyond } a'. \quad (7.7)$$

Neither of the last two is caution: each was added after a measurement. Without (7.6) the case $\mathbf{c} = [1,1]$ with exponent $[0,0]$ passed, where f is constantly one over the whole *ulp* above zero, so *every* real of the open interval is in the preimage; $f(a)$ did not contradict it, because $a = 0$ is outside the domain of `pow` and its image is empty rather than beyond $\mathbf{c}$. Without (7.7) thirteen lines of the corpus broke, all of the form `powRev1([0,4],[1,1],[0,1])`, whose preimage is the *open* interval $(0,1]$: there $f(a) = [0,0]$ does lie below $\mathbf{c}$, and yet every interior point meets it, since the exponent interval contains zero and $t^0 = 1$ whatever t is. The contact is not isolated, and it is the third condition that says so.

The proof rests on monotonicity across one *ulp*, which `pow` has in each argument separately and `sin`, `cos` and `tan` do not. The refinement is therefore requested by the caller rather than always on, and the three periodic functions do not request it.

Two sets of measure zero, recognised instead of searched. A search cannot find an answer that is a single point, and two of them occur:

$$\{t \geq 0 \mid t^e = 0\} = \{0\} \quad \text{for } e > 0, \quad \{y \mid a^y = 1\} = \{0\} \quad \text{for } a \neq 1. \quad (7.8)$$

Neither is reachable numerically: t^e underflows to zero long before t does, and a^y rounds to one for every $|y|$ below about $1.6 \cdot 10^{-16}$. A bisection on the representation therefore stops at the wrong double in both cases, while a comparison of the operands settles them exactly. Base $[2, 4]$ with $\mathbf{c} = [1, 1]$ activates both and now answers $[0, 0]$.

The grid of $\pi/2$, where no function is evaluated. Sine, cosine and tangent attain the only exactly representable values of their images, zero and ± 1 , on the grid $G = \{k\pi/2 \mid k \in \mathbb{Z}\}$, and they attain them exactly. Between two consecutive grid points each is monotone, so the image of an open cell is the open interval between two values that are known without computing anything:

$$f((k\frac{\pi}{2}, (k+1)\frac{\pi}{2})) = (f(k\frac{\pi}{2}), f((k+1)\frac{\pi}{2})), \quad f(k\frac{\pi}{2}) \in \{0, 1, -1, \pm\infty\}. \quad (7.9)$$

Deciding whether a cell can contribute to the answer is then a comparison against those five values, and *no function is evaluated at all*. The pole, which had been the obstacle, becomes the easy case: a pole belongs to the preimage of no real $\mathbf{c}$, so the grid steps over it without ever computing it. What the code needs is not π but two doubles that enclose $\pi/2$; doubling them is exact, so those two generate the whole grid.

The completeness check earned its place here. The first draft of the grid rule accepted t as an interior point of a cell even when t coincided with the grid point itself. For every $k \neq 0$ that cannot happen, since a non-zero multiple of $\pi/2$ is irrational and lies strictly inside its own enclosure, but the origin is exactly representable. So `tanRev([-1,0],[0,2.1])` judged the cell $(0, \pi/2)$, found it incompatible with $\mathbf{c}$ and stepped over the origin, discarding $t = 0$ where $\tan 0 = 0$ lies in $\mathbf{c}$: a violation of containment, that is of Cl. 10.4, and the only one this work introduced. The corpus has no such line and could not have seen it. What caught it was the check written in the sound direction: if the image of a *point*, computed by the library, is *contained* in $\mathbf{c}$, then the true value lies in $\mathbf{c}$ and the point must appear in the answer. Containment and not intersection, because the cosine of every $|t|$ below about $1.5 \cdot 10^{-8}$ comes back with one as its upper endpoint, so an image that merely touches $\mathbf{c}$ may be touching it by overestimation. Three point seven million points were tested this way, of which nine hundred and twenty-eight thousand were proved to be in the preimage, and none is lost.

Table 7.14 accounts for what is left, and the first group is the reason the block was stopped rather than continued.

Case	Verdict	Lines
cosRev at π	TIGHT is the <i>tightest</i>	2
tanRev, two cases	TIGHT is the <i>tightest</i>	4
pownRev, two cases	TIGHT is the <i>tightest</i>	4
powRev2, two cases	the corpus is not <i>tightest</i>	2
cosRev, two cases	exact bound, one and two <i>ulp</i> away	4
sinRev	exact bound, one <i>ulp</i> away	2
cosRev, three arguments	flat point at $k\pi$	6
sinRev	flat point at $\pi/2$	2
total		26

Table 7.14: The twenty-six residual lines of the corpus, which are thirteen cases counted also in their decorated form. Twelve of them, nearly half, fail because the answer of TIGHT is *tighter* than the one the corpus expects and still encloses the exact preimage: aligning with the corpus would make them worse. The eight flat points would need π to about a hundred bits, which is a separate piece of work.

M.13.1 The endpoint that falls on an exact double

Cl. 12.10.2, Cl. 10.5.4

3e76f16

Defect. The bisection returns the last point that still misses $\mathbf{c}$, which is one *ulp* outside the true border whenever that border is exactly representable. It is structural and not a matter of precision: eleven more bits of mantissa do not move a single endpoint, which was measured before anything was written.

Remedy. A refinement that takes the successor, under three conditions that must hold together: the image of the successor meets $\mathbf{c}$ at an endpoint of $\mathbf{c}$ which is also an endpoint of its own image; the image of the point itself lies strictly beyond that endpoint; and that endpoint of the image moves past the successor. The two endpoints are refined separately, with a guard against their crossing.

Evidence. Two of the three conditions were dictated by the corpus rather than by the reasoning, and the entry carries the figure that justifies the caution: the naive refinement repairs 47 endpoints and *breaks* 44, while the demonstrated one gives 195 correct endpoints out of 258. Failures on the corpus fall from 166 to 109. Table 7.15 carries the two borders the refinement must *not* close beside those it does.

Cost. Measured on the third axis, one commit against the next. The refinement *costs*, between 1.04 and 1.12, and is paid even where it finds nothing: a shape on which it moves no endpoint at all still pays twelve per cent, since it adds four evaluations of the function after every sweep.

Operation	Arguments	Answer	Note
powRev1	[2,4] [1,1]	[1,1]	exact
powRev1	[2,inf] [1,2]	[1,1.4142135...]	exact below
powRev1	[-inf,-2] [1,inf]	[0,1]	exact above
powRev1	[-inf,-2] [1,2]	[0.70710678...,1]	irrational: it can only be enclosed
powRev1	[0,0] [1,1]	[0,inf]	an open endpoint
powRev2	[2,4] [1,1]	[0,0]	encloses; M.13.3 closes it
sinRev	[0,0] [-1,1]	[0,0]	not touched

Table 7.15: Seven of the thirteen cases of the manual block of M.13.1, with the long endpoints truncated. The first group is what the refinement is for: a border that is exactly representable, which the bisection used to miss by one *ulp*. The second group is why the three conditions are needed together. An irrational border can only ever be enclosed, so there is nothing to take the successor of; and `powRev1([0,0], [1,1])` has the open preimage $(0, +\infty)$, where the naive refinement answered $[2^{-1074}, +\infty)$ and lost every real below the smallest subnormal. That is the shape of the 44 endpoints the naive form broke against the 47 it repaired. The last two rows are controls: `powRev2` still merely encloses at this step, and a periodic inverse does not ask for the refinement and must answer as before.

M.13.2 The two rules the bisection cannot find

Cl. 10.5.4, Cl. 10.5.3

22339eb

Defect. Two sets of measure zero. A power is zero for one base alone, and the sign of the exponent is fixed when base and image lie on known sides of one. A bisection cannot find either: numerically the power underflows to zero, and a power arrives at one for every exponent below about $1.6 \cdot 10^{-16}$.

Remedy. The two rules are placed at the head of the two operations and intersected with the argument before the bisection runs.

Evidence. Two of the conditions were corrected by the corpus, and it is the reason the measurement is made at every step: containing zero is not the same as having a non-positive lower endpoint, and the exponents that carry a zero base to zero form the *open* half-line, zero to the zero being outside the domain. Each of the two mistakes broke one line of the corpus and was found there. Failures fall from 109 to 79. Table 7.16 is the manual block, one group per rule.

Cost. Where a rule answers, the operation does not bisect at all, and that is the measurement of what recognising a set of measure zero is worth: the Level 1 rule of `powRev1` costs 0.06 times the direct power it avoids, that is some nine hundred and fifty times less than the sweep.

Operation	Arguments	Answer	Note
powRev1	[1,2] [0,0]	[0,0]	$t^e = 0$ for $t = 0$ alone
powRev1	entire [0,0]	[0,0]	
powRev1	[0,inf] [-inf,0]	[0,0]	
powRev1	[-inf,0] [-inf,0]	[empty]	no exponent above zero
powRev2	entire [0,0]	[0,inf]	
powRev2	[0,0] [0,0] [-inf,0]	[empty]	the set is open
powRev2	[0,0] [-inf,-0.1]	[empty]	zero is not in $\mathbf{c}$
powRev2	[2,4] [1,1]	[0,0]	the sign is fixed
powRev2	[0,0.25] [0,1]	[0,inf]	it starts exactly at zero
powRev2	[0.25,0.5] [1,2]	[-1,0]	it ends exactly at zero
powRev2	[0.5,2] [1,1]	[-inf,inf]	the base contains one

Table 7.16: The manual block of M.13.2, one group per rule. The first is the rule on the side of the base, a power being zero for one base alone; the second and third are the rule on the side of the exponent, which fixes its sign when base and image lie on known sides of one. Neither is reachable by a bisection: numerically the power underflows to zero, and a power arrives at one for every exponent below about $1.6 \cdot 10^{-16}$. The two rows that corrected the code are the sixth, where the set of exponents carrying a zero base to zero is the *open* half-line, zero to the zero being outside the domain, and the seventh, where containing zero is not the same as having a non-positive lower endpoint. The last row is the control that must *not* fire: a base containing one fixes no sign, so the answer is Entire.

M.13.3 The predicate decided in the logarithm

Cl. 10.5.4, Cl. 10.5.3

7358a29

Defect. Every endpoint that remained came from bases and images that are powers of two, where the logarithm is exact and the power is not.

Remedy. The identity $a^y \in \mathbf{c} \iff y \log_2 a \in \log_2 \mathbf{c}$, exact at Level 1, which changes where the rounding falls: the bisection then works on a product of intervals, which is *tightest* by M.2.4, and lands on the border. Two things the identity does not carry with it were found by the corpus: the base that is only zero, where a product with an infinity reads as enormous rather than as undefined; and the rule $0 \cdot \infty$, which lives behind the flag in the multiplication, so the product is rewritten inside the detail namespace of the inverses. **powRev1** is *not* touched: it has no failing line, and changing the predicate of an operation that answers correctly is risk without gain.

Evidence. Failures fall from 79 to 38, and **powRev1** and **powRev2** go from 57 and 73 failing lines to none and two.

Cost. Not separable on the first axis; measured as the block is, one commit against the next.

M.13.4 The periodic inverses decided on the $\pi/2$ grid

Cl. 10.5.4, Cl. 12.10.2

f8a300f

Defect. The three periodic functions reach 0 and ± 1 , the only exactly representable values of their image, on the grid of integer multiples of $\pi/2$, and there they reach them *exactly*. The bisection could not get there for two different reasons. On those points the predicate is blind, one *ulp* of error on the value displacing the argument by about $7 \cdot 10^{-9}$. And on the tangent it does not even start: the argument contains a pole, so the interval evaluation is unbounded and meets any $\mathbf{c}$, and the sweep stops at the poles and returns the whole interval where the true answer is a single point.

Remedy. Between two consecutive grid points each function is monotone, so its image over the open cell is the open interval between the two grid values, and deciding whether a whole cell can contribute is a comparison between $\mathbf{c}$ and numbers known exactly: *no function is evaluated*. The pole turns from the obstacle into the easiest case, since a pole belongs to the preimage of no real $\mathbf{c}$ and the grid crosses it without computing it.

Evidence. The completeness check caught a violation of our own before the commit, and it is the fourth defect outside the reach of the corpus, the first introduced by this work. The first draft accepted a point as interior to a cell even when it coincided with the grid point, which cannot happen for any non-zero multiple of $\pi/2$, that being irrational, but the origin is exactly representable: so a case throwing away the origin, where the tangent is zero and zero is in $\mathbf{c}$, lost containment. The check is written in the sound direction, containment and not intersection, over 126,333 points in the manual test and 3.7 million in a separate probe, zero lost. Failures fall from 38 to 26. Table 7.17 is the manual block, and half of it is what the rule must leave alone.

Cost. Tightness is free here: 4096 answers out of 4096 change and the time does not move. It is the converse of M.2.23, where conformance cost width, and of M.13.1, where it cost time.

7.5 The exhaustive audit of the obligations

Closing thirteen blocks does not prove coverage: it proves that thirteen blocks are closed. Clauses that no line of the roadmap happened to name would stay outside, and that had already happened four times, with **fma**, with the reductions, and with the three obligations of Cl. 14 recovered in G.12.

The audit therefore started from the opposite end. All 192 sentences of the body of the standard containing the word *shall* were extracted from the document and judged one at a time, with the evidence named: Table 7.18 is the result. A verdict of *satisfied* is not an opinion but a pointer, to a file, to an entry of the registry, or to the lines of the corpus that exercise it.

Operation	Arguments	Answer	Note
cosRev	[1,1] [-0.1,0.1]	[0,0]	a grid value
sinRev	[0,0] [-1,1]	[0,0]	
tanRev	[0,0] [-1.5708,1.5708]	[0,0]	through the poles
sinRev	[0,1] [-0.1,1.58]	[0,1.58]	the zero of the sine
sinRev	[0,1] [-0.1,3.15]	[0,pi+]	
cosRev	[-1,-1] [3.14,3.15]	[pi-,pi+]	tighter than the corpus
sinRev	[-0,-0] [3,3.5]	[pi-,pi+]	not touched
sinRev	[-0,-0] [2,2.5]	[empty]	not touched
cosRev	[-1,1] [-1.2,12.1]	[-1.2,12.1]	not touched
tanRev	[0,0], no x	Entire	not touched
cosRev	[c,c'] [1.57,1.58]	[0x1.921fb54442d17p+0, 0x1.921fb54442d19p+0]	not touched
sinRev	[c,c'] [3.14,3.15]	[0x1.921fb54442d17p+1, 0x1.921fb54442d19p+1]	not touched
tanRev	[-c',-c] [3.14,3.15]	[0x1.921fb54442d17p+1, 0x1.921fb54442d19p+1]	not touched

Table 7.17: Thirteen of the fifteen cases of the manual block of M.13.4, where pi- and pi+ are the two double values that straddle π . In the last three rows c is 0x1.1a62633145c06p-54 in the first and 0x1.1a62633145c06p-53 in the other two, and c' is the double after c . The first group is the borders the grid closes, including the tangent *through* the poles, which the sweep could not even start on, and the row that answers more tightly than the corpus asks. Everything below is the other half of the test, and it is the half that matters in an entry that tightens: these are the cases the rule must not disturb, the last three being where the bisection already arrives by itself.

Verdict	Sentences	Meaning
satisfied	149	with the file, the entry or the corpus lines named
not applicable	30	they define the flavour, not the implementation
not applicable	10	compressed arithmetic, other flavours, multiple types or formats, none of which is claimed
uncovered	3	two of code, one documentary

Table 7.18: The audit of the 192 obligations of the standard. The three uncovered sentences were closed by the audit itself: two of them asked the reductions to raise the flags of IEEE 754-2019, which became M.10.6, and the third was a statement in our own documents that claimed more latitude than Cl. 12.12.2 allows.

Two further passes were run alongside it, because the first gap had been found in a table rather than in a sentence. One is a translation unit that names every row of Tables 9.1, 10.1, 10.2, 10.3 and 10.5 and of the Cl. 12.12 clauses, separately for the bare and the decorated type, and whose only requirement is that it compile: 234 rows, and it compiles. The other is an empirical probe of a hundred and twenty-six checks on the properties the clauses state in prose, which passes with the conformant build and *aborts* with the non-conformant one, on the square root of an empty interval, after having already failed on the product, the division, the reciprocal and the power. That abort is the most direct evidence available that conformance is carried by the flag.

The third uncovered sentence deserves its own remark, because it was not a defect of the code. Cl. 12.12.2 narrows at Level 2 the latitude that Cl. 11.6 leaves at Level 1: a decoration must be the strongest possible whenever the accuracy mode of the corresponding bare operation is *tightest*. The registry and the gap analysis both stated, without that reservation, that a weaker decoration remains conformant. The code was and is conformant; it was the written justification that was too generous, and it was corrected in both documents.

7.6 What remains open

Three things were left alone deliberately, and each is left alone for a reason that is worth stating rather than for lack of time.

Twelve of the twenty-six residual lines are a result, not a residue. In each of them the answer of TIGHT is strictly tighter than the one the corpus expects and still encloses the exact preimage, which was verified case by case: in one instance by computing the value to sixty digits, in another by sampling the forward function densely around the pole. In two rows of `tanRev` the answer does not even contain the one the corpus expects, which is the same fact seen from the other side and which a first reading of the measurement had reported as two violations. Changing the library to agree with those lines would make it less tight, and Cl. 12.10.2 asks only for *valid*.

The eight flat points need extended precision. Where c is one *ulp* wide around ± 1 , the true half-width of the preimage is of the order of $\sqrt{2u}$, and an error of one *ulp* on the value displaces the endpoint by about $7 \cdot 10^{-9}$: near an extremum the function does not move. No predicate built on values accurate to one *ulp* can separate those points. Closing them means π to about a hundred and six bits and an argument reduction to match, which is an entry of its own and is not opened.

The four design decisions of the kernel stay as they are. N.2, N.6, N.7 and N.8 would cost the kernel the performance it was written for. Two of the four do not occur through TIGHT, the dispatch taking the conformant path on exactly the operands that produce them, and the other two are the spurious flags, which an interval layer whose standard rejected global flags does not expose.

One direction is open in a different sense, being a decision rather than a piece of work: the advisor proposed extracting the interval arithmetic out of NFG and incorporating it

into TIGHT, so that the library becomes a thing in itself, detached from the geometric predicates, with conformant and non-conformant behaviour chosen by the flag it already has. That is discussed in Chapter 9, where it belongs, since it concerns what should be done next rather than what was done.

Chapter 8

Experiments

This chapter reports what the platform of Chapter 4 measured on the changes of Chapters 5 to 7. It has two halves, answering the two questions that are still open after a change has been written: how far conformance actually went, and what it cost. The second half is the answer to **RQ3**, and it is the question the research group asked at the outset, in the form of a doubt: adding a branch for the non-finite values to a branch-free kernel might be expensive, and nobody knew how expensive.

Two conventions are used throughout and are stated once. Every ratio is the conformant time divided by the non-conformant one, so a value above one is a surcharge and a value below one means the conformant build is the faster of the two. Every ratio is read against the margin of Section 4.3.3: within twelve per cent of unity there is nothing to read.

8.1 Conformance results

The first question is whether the work repairs what it claims to repair, and the evidence is of two kinds because the two pieces of software are judged against two different standards. Section 8.1.1 answers for the kernel, where the instrument is a probe and not a corpus, and reports it across the five versions. Section 8.1.2 follows the external corpus from the first run to the last, block by block of the roadmap. Section 8.1.3 accounts for the twenty-six assertions that still fail, none of which is a violation of the standard.

8.1.1 The kernel, commit by commit

The defects of Chapter 5 were established with a probe rather than with a corpus, there being no external suite for a kernel that is not an interval library in the ordinary sense, and the probe is described in Appendix C. Its verdict is a count of checks passed over checks run, and Table 8.1 follows that count across the five versions of `numerics.h` that the repairs produced. The five were compiled and run in one session, so the rows are comparable with each other, and each row was run on the three compilation paths at two optimisation levels.

Commit	Repairs	division			cube	fmsub	isNAN
		AVX2 /	SSE2 /	scalar	3 paths	AVX2	3 paths
52a1d3d	none	7/7	2/7	0/7	6/11	0/6	6/9
a10c1a0	N.1	7/7	7/7	7/7	6/11	0/6	6/9
9ec464d	N.3	7/7	7/7	7/7	11/11	0/6	6/9
e18a495	N.4	7/7	7/7	7/7	11/11	6/6	6/9
b5e9dd8	N.5	7/7	7/7	7/7	11/11	6/6	9/9

Table 8.1: The probe across the five versions of the kernel, as checks passed over checks run; the cells in bold are those in which the defect is still present. The counts do not depend on the optimisation level, `-O2` and `-O0` agreeing on all five rows. The last row is the head of the fork: zero containment violations and zero failed checks over all six configurations. On the same version the division matrix, which is seven cases over three paths and two optimisation levels, reports 42 enclosures out of 42.

Two things in that table are worth reading aloud. The first is that the division was wrong on *finite ordinary operands* on two paths out of three, and right on the third: a defect that a user of the vector build could not have met, and that a user of a machine without AVX2 met on every sufficiently wide quotient. The second is that each repair moves exactly one column and leaves the others where they were, which is what a sequence of local repairs should look like and is the evidence that the four are independent.

The probe also counts the exception flags that IEEE 754-2019 requires, which is a weaker instrument and is reported as such. On the complete AVX2 run at the head of the fork the flags raised are `INVALID` twenty-six times, `INEXACT` twenty-nine, `OVERFLOW` four, `DIVBYZERO` once and `UNDERFLOW` twice. Up to `9ec464d` the same run gave `INVALID` twenty-seven, `INEXACT` thirty-three and `DIVBYZERO` three, and the difference is exactly the spurious flags that `N.4` removed: a subtraction written as a multiplication by a reciprocal divided by zero and signalled an invalid operation, in an operation that contains no division at all.

8.1.2 The corpus, block by block

Table 8.2 follows the external corpus from the first run to the last. Two columns are needed because the denominator moves: the plugin declares only the operations the library provides, so a block that adds operations also enlarges the corpus it is measured against. The shape of the progression is the one plotted in Figure 5.2.

Three things in that table are worth reading aloud.

The single largest movement is `G.2`, which takes the failures from over a thousand to one. Within it, one change accounts for almost half: the domain filter of the elementary functions repaired seventy-nine assertions, and the domain of `pow` another four hundred and ninety-nine. That the arithmetic of an interval library should be the bulk of its conformance is not surprising; that a single missing filter should be worth five hundred assertions is a measure of how much of the standard rests on `Cl. 10.4`.

After	Assertions	Failing	What happened
first run	4358	1084	the baseline, before any change
G. 2	4409	1	the arithmetic itself
G. 3	4543	35	the constructors, and the thirty-three signalling lines that could not pass yet
G. 4	4864	34	the eleven booleans, with <code>equal</code> among them; all 321 new assertions passing
G. 5	4947	34	set operations, all new assertions passing
Tab. 9.1	5599	34	<code>fma</code> and the six integer functions
G. 6	6066	36	two-output division; the two failures are <code>[nai]</code>
G. 7	7367	185	the twelve reverse operations, eight hundred new assertions in one block
G. 8	7367	152	the exception register; the thirty-three signalling lines now pass
G. 9	9288	166	the decorated type, and <code>atan2</code> declared
G. 10	9433	166	decorated constructors and reductions
G. 13	9433	26	the tightening of the reverse operations

Table 8.2: The external corpus, block by block. At no step did a line that passed before fail afterwards: every step was verified with `comm` on the two sorted lists, case by case and never by the difference of the totals. The last measurement taken with the flag off, after G. 8, gives 1612 failures against 152 with it on.

The two rises, at G. 7 and G. 9, are not regressions. Each of those blocks brought several hundred assertions that had never been exercised before, and the new failures are new lines, not broken ones. This is exactly the case that a comparison of totals would misreport, and the reason the platform compares the two lists line by line.

The last row is the one that is easy to misread in the other direction. G. 13 closes no obligation: it repairs one hundred and forty lines on which the library was already *valid*, and merely less tight than the corpus expected.

8.1.3 What the residue is

Twenty-six lines out of 9433 still fail, and they are classified one by one in Table 7.14. The distinction that matters is that **a failure against the corpus is not by itself a violation of the standard**. Cl. 12.10.2 requires *tightest* of the basic operations, of the integer functions and of the `absmax` family; of the reverse operations it requires only *valid*, recommending *accurate*. All twenty-six residual lines are *valid*, and twelve of them fail because the answer of TIGHT is *tighter* than the one the corpus expects and still encloses the exact preimage.

That group was not asserted but verified, case by case and by different means according to the case: one bound was computed to sixty digits, another was established by sampling the forward function densely on both sides of a pole, and in one instance it was shown that

π lies strictly between the two doubles the library returns, so that no narrower interval with representable endpoints exists. Aligning the library with those lines would make it less tight, which is the opposite of what the library is for.

The failure is therefore an artefact of the way the corpus decides. It compares the computed interval with the expected one by equality, so an answer narrower than expected is counted as a failure even where Cl. 12.10.2 asks only for *valid*. A comparison by containment would follow the standard instead. The point has been discussed with one of the authors of TIGHT, and the cases will be collected and proposed to the maintainers of the corpus.

The remaining fourteen are the eight flat points, where the preimage cannot be separated by any predicate built on values accurate to one *ulp*, and six exact bounds one or two *ulp* away.

That the residue stops there rather than at zero is itself a measured result, and Table 8.3 is the measurement. Before the tightening of G.13 was designed, the sweep of the reverse operations was rewritten in four variants and each was run over the 129 bare lines of `powRev`, which carry 258 endpoints. The natural hypothesis was that the bisection decides with an interval power one *ulp* wide, so that more precision would close the gap. The table refutes it: eleven extra bits of significand move *not one endpoint*. The cause is structural rather than numerical, the sweep answering with the last point that still misses the target, which is one *ulp* outside whenever the true bound is exactly representable, and the bounds the corpus asks for are full of exactly representable values.

Variant of the sweep	Endpoints right	Not enclosing
double, as it was	109	2
long double	109	2
double plus one <i>ulp</i> , naive	156	44
double plus one <i>ulp</i> , proved	195	2
long double plus one <i>ulp</i> , proved	214	2

Table 8.3: Four variants of the sweep, over the 258 endpoints of the 129 bare lines of `powRev`. The second column counts the endpoints that match the corpus; the third counts those that fail to enclose, which is the only column in which a number may not grow. Extended precision buys nothing on its own. The naive refinement, which drops an endpoint because the function misses the target at that single point, buys forty-seven and *breaks forty-four*, which is why Section 7.4 proves its three conditions instead of assuming them.

The last line of that table is also the measure of what was left on the table: extended precision, on top of the proved refinement, would move a further nineteen endpoints. It is not an entry of this work, and Chapter 9 says why.

Conformance itself was not established by the corpus alone, and this is the point of Section 7.5: the 192 obligations of the standard were judged one at a time, 149 of them satisfied with the evidence named and the rest not applicable to the flavour and the claim of this implementation. The audit found three uncovered sentences, and closing them was

the last change of the work. The complementary evidence is the property probe, which passes its hundred and twenty-six checks with the conformant build and *aborts* with the other one.

8.2 The cost of conformance

The second question is what that conformance costs, and it is answered over the cells of the benchmark rather than on a single figure of merit. Section 8.2.1 reads the 332 cells on which the two builds return the same answer on every operand, which are the only ones where a ratio is a cost at all. Section 8.2.2 takes the cells on which they answer differently and says why those must not be read as costs, one group of them being a saving. Section 8.2.3 measures the decorated type against the bare one, which is the second axis rather than the first.

8.2.1 Results by block

The benchmark tables of the registry hold 495 cells on the first axis. In 345 of them the divergence column is zero, which is to say that both builds returned the same answer on every operand and the ratio is therefore the cost of the added checks in pure form; in the other 150 the two builds answer differently, and the ratio compares two procedures rather than measuring a surcharge. The thirteen cells the registry declares unreadable are set aside, and Section 8.3 says why. What is left is the 332 cells that measure a cost, and they are grouped by the time of the non-conformant build in Table 8.6.

Before the shape of those cells is described, two aggregations say where the surcharge fell. Table 8.4 groups the cells by the block of the roadmap that produced them, and answers the question a reader of Chapters 5 to 7 arrives with: which part of conformance was expensive. The answer is that only three blocks are visible at all. The arithmetic itself, **G.2**, carries more than half of the cells and the highest median; the reverse operations and the cancellative ones follow; and everything from the exception register onwards is free, with medians at unity and three quarters of the cells inside the margin. That is not a coincidence of measurement but a consequence of where the work happened: the later blocks add names that did not exist, so the conformant build runs code that the other one does not have, and the cost they report is the cost of the constructor they inherit rather than of a check.

Table 8.5 cuts the same cells the other way, by compilation path. The three differ in their base cost by a factor of three, which Table 3.4 explains: the vector path computes a product without branching, and the scalar one walks a nine-case analysis. The surcharge follows the law stated below rather than the path: it is largest in relative terms where the operation is cheapest, so the AVX2 build, which is the fastest of the three, is also the one on which conformance is most visible. A reader looking for the worst case of this work should look at the AVX2 column and not at the scalar one, which is the opposite of the intuition that a slower path suffers more.

Block	Cells	Of which A	Lower q.	Median	Upper q.
G.2	276	138	1.04	1.20	1.57
G.3	17	14	1.01	1.02	1.06
G.5	15	15	0.97	1.16	1.38
G.6	39	39	1.01	1.10	1.65
G.7	45	39	1.04	1.20	1.50
G.8	21	21	1.00	1.00	1.01
G.9	15	15	1.00	1.00	1.02
G.11	27	27	0.99	1.00	1.02
G.12	27	24	1.00	1.06	1.17
482		332			

Table 8.4: The surcharge by block of the roadmap. The second column counts all the cells the block produced on the first axis, once the thirteen unreadable ones have been set aside; the third counts those of class A, on which both builds answer identically, and the three quartiles are computed over those alone. G.1 and G.4 do not appear because their operations construct no interval and were used to calibrate the margin; G.10 and G.13 are measured on the other two axes. This grouping is ours; every cell in it is a figure already published in the registry.

Path	Cells	Median time, flag off	Lower q.	Median	Upper q.
AVX2	111	3.30 ns	1.01	1.08	1.28
SSE2	111	5.23 ns	1.01	1.05	1.19
scalar	110	10.19 ns	1.00	1.12	1.45

Table 8.5: The surcharge by compilation path, over the cells of class A alone. The third column is the median time of the non-conformant build on that path, and is the reason the three rows are not comparable as absolute times: they are three different programs. This grouping is ours; every cell in it is a figure already published in the registry.

Those 332 cells have a shape, and it is the main quantitative result of this chapter: **the surcharge is a fixed quantity of work, so it weighs in proportion inversely to the cost of the operation it is added to.** Table 8.6 states it as numbers.

That statement can be made precise, and doing so is what turns five medians into a law. Suppose the conformant build performs the same work as the other one plus a fixed quantity c , independent of the operation: the checks are a few comparisons on the endpoints, and comparisons do not know how expensive the operation they guard is. Then for an operation costing t without the flag,

$$t_{\text{ON}} = t + c \quad \implies \quad r(t) = \frac{t_{\text{ON}}}{t} = 1 + \frac{c}{t}, \quad (8.1)$$

a hyperbola that falls towards one. The model has one parameter, so each band of Table 8.6 estimates it: taking t at the geometric centre of the band and r at its median, $c = t(r - 1)$ gives

Cost of the operation	Cells	Lower q.	Median	Upper q.
under 2 ns	94	1.00	1.15	1.50
2 to 5 ns	65	1.00	1.17	1.62
5 to 20 ns	61	1.02	1.11	1.45
20 to 100 ns	40	1.03	1.09	1.29
over 100 ns	72	1.00	1.02	1.05

Table 8.6: The surcharge against the cost of the operation, over the 332 cells on which both builds return the same answer. The quartiles are given because the median alone would hide how wide the first two bands are: an operation of one nanosecond can be made half as fast again by two comparisons. Above a hundred nanoseconds three quarters of the cells lie within five per cent of unity, which is inside the margin. This grouping is ours; every cell in it is a figure already published in the registry.

Band	t (ns)	Median r	$c = t(r - 1)$ (ns)
under 2	1.0	1.15	0.15
2 to 5	3.2	1.17	0.54
5 to 20	10	1.11	1.1
20 to 100	45	1.09	4.0
over 100	300	1.02	6.0

The estimate is not constant: it grows with t , from about a sixth of a nanosecond on the cheapest operations to six nanoseconds on the most expensive. So (8.1) is the right shape and the wrong constant, and the discrepancy is itself informative. A fixed c of one sixth of a nanosecond would predict a ratio of 1.0005 in the last band, where the measurement gives 1.02; a fixed c of six nanoseconds would predict 7 in the first, where the measurement gives 1.15. Neither is observed, and what separates the bands is that the expensive operations are not the cheap ones with a check bolted on: they call an elementary function, so they also construct their result, and Section 4.3.3 sets twelve per cent as the level below which a ratio says nothing at all. In the last band the whole surcharge lies inside that margin, which is the same statement as c being unmeasurable there rather than large.

Counted against the margin of (4.3) rather than against a model, the 332 cells fall as follows: **188 of them lie inside the band**, that is to say measure nothing at all, 141 lie above it and 3 below. So on fewer than half of the cells of class A is there a surcharge to report, and the reader who wants one figure for how often conformance is free on this axis has it there.

The useful form of the law is therefore the weaker one, and it is the one that holds over three orders of magnitude: the surcharge is bounded work, so the ratio it produces is bounded by $1 + c_{\max}/t$ and becomes invisible once t grows past a hundred nanoseconds. That is enough to answer the question the research group asked, since it says where the cost can matter and where it cannot.

The law is visible entry by entry as well, and the two extremes of the library make it concrete. The most expensive family in relative terms is the product with an infinite

endpoint, **M.2.2**, whose median ratio over its twenty-one cells is 2.32: the operation costs about one and a half nanoseconds without the flag, and the dispatch that decides whether $0 \cdot \infty$ can arise adds four comparisons to it. At the other end, **tan** on an ordinary argument costs about a hundred and fifty nanoseconds, of which the conformance checks are within the noise: its three cells give 1.02, 1.00 and 1.00.

Read against the concern that opened the question, the answer to **RQ3** is therefore not a single number, and any single number would have been misleading. On everything that calls an elementary function conformance costs nothing measurable. On the cheapest operations of the library it costs from nothing to about half as much again, that being the third quartile of the first band, and the one family that goes well past it is the product, at 2.3, where four comparisons are added to an operation of a nanosecond and a half. In a geometric predicate, where the operations that dominate are additions and products of intervals, the first figure is the one that applies.

One further observation belongs here, because it changes how the first figure should be read. In several entries the surcharge is not the check itself but the *public constructor* the operation calls to build its result: this was established by reading the generated assembly, where the constructor expands inline and the function doubles in size, and confirmed by the entries whose operations return a **double** rather than an interval, which give exactly 1.00. Where that constructor was made cheaper, by recognising the degenerate infinite pair from the bits instead of by four floating-point comparisons, the ratio of the constructor in isolation fell from 2.78 to 1.38 on the vector path and the whole family moved with it.

8.2.2 Where conformance is not a cost

The remaining 150 cells of the first axis are the ones on which the two builds answer differently, and they are the ones that must not be read as costs. Two groups deserve to be stated, because they are the most instructive measurements of the campaign.

A ratio in the tens usually means the other build computed nothing. The largest ratio of the whole work is 45.47, on **pow** with a base that leaves the domain. Without the flag the answer is a **NaN** produced in about two and a half nanoseconds; with it, the base is restricted to the domain and the power is actually evaluated. The temptation is to read that as the cost of conformance, and the entry answers it with a measurement instead of an argument: on the same operands, the whole operation costs 115.535 nanoseconds, the domain filter alone costs 1.747, and **pow** on an already restricted base costs 114.019. The conformance code is one and a half per cent of what is being measured; the rest is the elementary function that the non-conformant build never calls. Removing the filter entirely would move the ratio from 45.5 to about 45.0.

A ratio below one means the conformant path is genuinely shorter. Fifty-one of the cells on which the two builds disagree fall below unity, and the extreme is 0.029, on the empty operand of **pow** in the scalar path. The cause was traced and is worth recording: the non-conformant build answers with a constant built from a *signalling NaN*, and in the pure scalar path the doubles travel through the x87 registers, where merely

loading a signalling NaN raises the masked invalid exception and triggers a microcoded assist of about a hundred and fifteen nanoseconds. The conformant build recognises the empty operand and returns in three. The same constant explains four other cells of the same size in three different entries, which is how it was identified as a single cause rather than a handful of coincidences.

There is also one case in which conformance costs neither time nor nothing, but *width*, and it is the only one in the library. For `atan2` over a rectangle that crosses the cut along the negative axis, the non-conformant build returns the complement of the image, which is narrower and is what the surrounding geometry wants; the standard requires an interval, so the conformant build returns the enclosure, of width 6.28 against 4.71. It is also faster, because a constant needs no evaluation. The entry states the price rather than hiding it, and it is the clearest example of why conformance sits behind a flag instead of replacing the previous behaviour.

8.2.3 The cost of the decoration

The decorated type is measured on the second axis, the decorated operation against the bare one within the same build. Table 8.7 covers the three ways in which a decoration is computed.

Operation	Path	Bare	Decorated	Ratio
sum	AVX2	3.273	10.703	3.27
	SSE2	3.244	11.206	3.45
	scalar	5.369	25.725	4.79
product	AVX2	6.659	20.285	3.05
	SSE2	20.388	40.916	2.01
	scalar	30.436	54.887	1.80
division	AVX2	8.760	18.854	2.15
	SSE2	17.826	30.275	1.70
	scalar	26.082	43.253	1.66
sqrt	AVX2	9.075	15.266	1.68
	SSE2	9.335	15.612	1.67
	scalar	30.338	39.243	1.29
tan	AVX2	92.644	102.184	1.10
	SSE2	93.035	102.381	1.10
	scalar	94.132	111.183	1.18
ceil	AVX2	3.882	20.663	5.32
	SSE2	6.559	25.885	3.95
	scalar	7.929	32.579	4.11

Table 8.7: The cost of carrying a decoration, in nanoseconds per operation. Both columns come from the conformant build, and the divergence column of the original table is zero on all eighteen cells: the decorated operation returns the same interval as the bare one, and what is being compared is the price of computing the decoration alongside it. Transcribed from the table of M.9.3.

The same law holds on this axis, and more sharply. The sum is the cheapest operation and pays the most, from 3.27 to 4.79 times; `tan` costs ninety-three nanoseconds bare and pays about ten per cent. Between them the product, the division and the square root fall in order of their bare cost.

`ceil` is the exception, and its cause is structural rather than one of scale: it is cheap, but the decoration of an integer function has to distinguish `com` from `dac`, which requires knowing whether the value is unchanged over a neighbourhood, so the bare function is evaluated *twice*, on the argument and on the argument widened by one *ulp*. The assembly confirms it, going from ninety-one instructions and one call to four hundred and ninety-one and eight.

Two further results of that block bound the axis from below and from above. Type conversion, which is the identity when source and target are the same type, still costs between 1.54 and 1.70 decorated: that is the floor, the price of a byte carried beside two doubles, and no decorated operation can cost less than about one and a half times its bare form. At the other end, the five operations that only inspect or build a decoration compile to identical assembly in both builds and are not affected at all: reading a decoration costs a third of a nanosecond, building one costs five.

8.3 Threats to validity

The measurements above have five limits, and it is better to state them than to let a reader find them.

Thirteen cells are not readable, and the cause is known. They all measure `textToInterval` in its inf-sup form, or a constructor that forwards to it, and they are spread over three entries of the registry: six belong to the constructors that signal, six to the exact representation and one to the text constructor itself. Their timing is *bimodal*: it falls either around three hundred and thirty nanoseconds or around six hundred, and which of the two a given compilation lands on changed between the measurement and a later probe, in three cells out of six, with the only difference in the source being a function added to the benchmark and never called from that path. The three alignments do not absorb it, since the reported figure is already the minimum over them, and the one cell of the text constructor declares itself with a **spread** of 1.84 against the 1.05 typical of its neighbours. The assembly rules out size as the cause: with the flag the function is *smaller* and yet slower in three of the six configurations. Every cell that the registry marks this way is excluded from the counts of this section, which is why they are 332 and not 345.

The margin of uncertainty is a floor, not a proof. It was calibrated on operations whose two builds compile to identical assembly, so it measures the noise of this platform on this machine. It does not license reading a ratio of 1.13 as a real cost; it only forbids reading 1.11 as one. Where a difference near the threshold mattered, the entry re-measured it in a short controlled probe rather than arguing about it, and in one case that probe moved a cell from 0.84 to 1.00.

Absolute times are not comparable across sessions. They depend on the instantaneous frequency of the machine, so no figure from one run is set beside a figure from another; ratios are far more stable, since the two builds are interleaved within one run. When a change made an earlier measurement obsolete, the measurement was repeated rather than carried forward. One consequence is visible in the registry and is left there deliberately: two entries that quote each other's figures had to be corrected when the constructor they both call was made cheaper, because a ratio whose denominator falls rises without anything else changing.

The flag counts of Section 8.1.1 measure 754 and not 1788. The two standards do not ask the same thing here, and the probe answers to the first of them only. Cl. 1.4 of IEEE 1788-2015 puts the behaviour of interval operations with respect to the flags of IEEE 754-2019 outside its scope, and does not require an implementation to expose the inexact flag at all, whereas 754 Cl. 7.6 prescribes how a conforming implementation handles them. The probe therefore checks the kernel as a floating-point program, which is what it is, and none of its flag counts is evidence for or against conformance to IEEE 1788-2015. This is the same distinction that answers RQ1 in Chapter 9, seen from the side of the measurement.

The corpus is an oracle with a known blind spot. It measures only what it knows how to exercise, and the four defects of Table 4.4 were found outside it. Its expected values are also not always the tightest ones: in two lines of `powRev2` the corpus itself is not *tightest*, which was established by sampling the forward function on two hundred thousand points on either side of the bound. A test suite written by other people is the strongest evidence available for conformance, and it is still evidence about the intersection of what it covers with what the library does.

Chapter 9

Conclusions and Future Work

This work set out to establish whether an interval library written for speed inside a geometric predicate system can be made conformant to IEEE 1788-2015 without giving up what makes it useful, and to say what that conformance costs. It can, and the price is now a measured quantity rather than a matter of opinion. This chapter states the answers, what the work leaves behind, what it did not close and why, and the direction that remains open.

9.1 Answers to the research questions

RQ1: the two standards govern different objects. IEEE 754-2019 governs the numbers that serve as endpoints, IEEE 1788-2015 the intervals built on them, and the second refers to the first rather than repeating it. Conformance to one therefore says nothing about conformance to the other, and the two obligations meet at only one point: Cl. 12.6.1 defines a 754-conforming interval type, and an implementation may claim it if the accuracy requirements of Cl. 12.10 are met. That claim, it turned out, is about *values* and not about flags, which settled a question that had been open at the start of the work: the spurious exception flags of the kernel, N.7 and N.8, are genuine departures from IEEE 754-2019 taken by themselves and yet do not stand in the way of the interval claim, because IEEE 1788-2015 rejected global flags in favour of decorations.

The practical consequence is visible throughout Chapters 5 to 7. The representation of the kernel settles rounding once and for all, so an error in one of its operations is not an error of rounding but an error in the choice of which endpoints to combine: scalar arithmetic is not where those defects live, and looking for them there would have found nothing.

RQ2: the wrapper is almost, but not quite, enough. The criterion that decides is stated in Section 5.1: a wrapper repairs what it recognises from the operands *before* calling, and a result that is wrong on ordinary finite operands has to be repaired at the source. Applied to what the analysis found, the criterion assigns almost everything to the

wrapper. Seventy-one changes to TIGHT closed thirteen blocks of the roadmap, and of the nine defects of the kernel, four are design decisions. Two of them are produced only by operands the dispatch recognises, so through TIGHT they do not occur at all; the other two raise exception flags that an interval layer does not expose.

Two defects fall on the other side, and both deliver a wrong value from ordinary data: the division, which pairs the endpoints as if it were a multiplication and returns intervals that do not contain the quotient, and the conversion of an exact rational to a double, which loses the sign on overflow. The first of the two is in the official repository of NFG; the second is implemented and published on the fork. So the answer to RQ2 is that the wrapper carries the overwhelming majority of conformance, and that a small, identifiable and repairable core of it cannot be carried anywhere but in the kernel.

RQ3: the cost is a fixed amount of work, and it weighs where the operation is cheap. Over the 332 benchmark cells in which both builds return the same answer on every operand, the median surcharge is fifteen per cent below two nanoseconds per operation, and two per cent above a hundred, with three quarters of the cells in that last band within five per cent of unity. The extremes are the product with an infinite endpoint, whose dispatch costs about 2.3 times the bare operation, and the elementary functions, on which conformance is not measurable at all.

The concern that opened the question was that adding a branch to a branch-free kernel would be expensive. It was well founded in form and misplaced in target. The kernel is untouched; the checks live in the wrapper; and where the surcharge is large, the assembly shows that a good part of it is not the check but the public constructor the operation calls to build its result, which is also why making that constructor recognise a degenerate pair from the bits moved a whole family of entries at once. What the measurement does not support, and what would have been the comfortable answer, is a single number: between an operation of one nanosecond and one of a hundred and fifty, the same added work is a surcharge of half as much again, or none at all.

9.2 What the work leaves behind

Table 9.1 lists what exists at the end that did not exist at the beginning. Two of the four are of use beyond this thesis, and it is worth saying which.

The platform is the part most likely to outlive the specific changes. It decides correctness against a criterion this work did not write, it reports what a change repaired and what it broke case by case rather than by the difference of two totals, and it states the margin below which its own numbers are not readable. A later change to the library can be judged by the same instrument without renegotiating what counts as evidence.

The four repairs accepted upstream are of a different kind, and their value is one of provenance. That the kernel had those defects is no longer a statement a reader must take on trust from a fork: it is checkable against a history that is not ours.

Artefact	What it is
a conformant library	TIGHT satisfies IEEE 1788-2015 in the build with the flag, verified against 9433 assertions of an external corpus and against the 192 obligations of the standard read one at a time. The previous behaviour survives in the other build
a verification platform	the corpus of ITF1788 connected to the library by a plugin, plus the test blocks and the property checks that cover what the corpus cannot reach
a benchmark on three axes	a program that measures the conformant build against the non-conformant one, the decorated value against the bare one, and one commit against the preceding one, with a calibrated margin of uncertainty
four repairs upstream	N.1, N.3, N.4 and N.5 are in the official repository of NFG, so they now benefit every user of the kernel and not only this fork

Table 9.1: What the work leaves behind. The registry of the changes, with one clause, one commit and one test block for each of the seventy-one entries, is summarised in Appendix A; the commands that reproduce any measurement are in Appendix B.

9.3 Limitations

Three things were left open, and each was a decision rather than an omission.

Twenty-six lines of the corpus still fail, and twelve of them are a result. In those twelve the answer of TIGHT is strictly tighter than the one the corpus expects and still encloses the exact preimage, which was verified case by case. Cl. 12.10.2 asks of the reverse operations only that they be *valid*; aligning with the corpus would make the library less tight, which is the opposite of its purpose. Of the remaining fourteen, eight are flat points where no predicate built on values accurate to one *ulp* can separate the preimage, and six are exact bounds one or two *ulp* away. The criterion itself will be raised with the maintainers of the corpus: comparing by containment rather than by equality would follow Cl. 12.10.2, which asks of these operations only that they be *valid*.

The block on tightness was stopped rather than finished. G.13 is the only block that closes no obligation of the standard, so no clause says when it is done; it would be finished only when every line of the corpus agrees, which is not a conformance goal and for twelve lines is not even desirable. Where to stop was therefore a judgement, and it is recorded as one.

The four design decisions of the kernel stay as they are. Repairing them inside NFG would cost the performance the kernel exists for. Two of them do not occur through TIGHT, the dispatch taking the conformant path on exactly the operands that produce them, and the other two are exception flags that an interval layer does not expose.

Two of the limits stated in Section 8.3 are not repeated here, except to note that they are limits of measurement rather than of the changes: thirteen benchmark cells are not

readable because their timing is bimodal, and the corpus is an oracle whose blind spots were found by other means, four times.

9.4 Future work

Extracting the interval arithmetic. The direction proposed by the advisor in August 2026, and the one that would change the most, is to take the interval arithmetic out of NFG and incorporate it into TIGHT, so that TIGHT becomes a library in its own right rather than a wrapper over the numeric core of a geometric predicate system. Figure 9.1 contrasts the two arrangements. The flag this work introduced is what makes the proposal tractable: the two behaviours are already separated, already tested against each other on every operand shape, and already measured, so the question that remains is one of packaging and of agreement with the maintainers rather than of semantics.

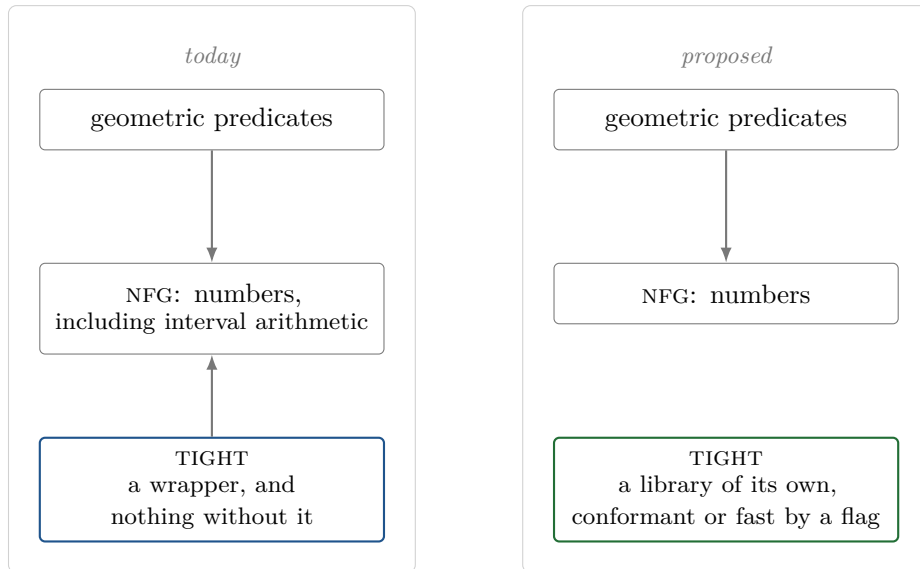


Figure 9.1: The direction that remains open. The interval arithmetic moves from the kernel into the library above it, which then stands on its own; the geometric predicates keep the numbers they need. What this work contributes to the proposal is that the two behaviours are already separated by a flag, and that the cost of choosing one over the other is known operation by operation.

Extended precision where the library is accurate but not tightest. The eight flat points of Table 7.14 need π to about a hundred and six bits and an argument reduction to match. The same machinery would close the two `pownRev` cases whose root is not a power of two, where the present method is limited by an oracle that is itself one *ulp* wide. This is a self-contained entry and it is not opened.

The entries of the kernel that remain open. N.9 is implemented and published on the fork, and its request has not been opened. The other three are N.6, N.7 and N.8, that is the design decisions minus N.2. Whether TIGHT meets them or not, in NFG they stand, and they should be discussed with the maintainer rather than proposed as patches, since the trade-off they embody is his to make.

The decorated operations that were measured and not optimised. One margin was found and deliberately left: the decorated addition writes the bare sum twice, once inside an argument and once inside the call, and the compiler eliminates neither. A variant written in the benchmark, and therefore outside the library, answers identically on four thousand and ninety-six rows in six configurations and costs between eighteen and forty per cent less. It sits below the threshold that this work set for intervening in the library, so it is recorded rather than applied.

Appendix A

Summary of the Change Registry

The registry of the changes, `modifiche_effettuate.tex`, is a document of its own: two hundred and forty-three pages, one page or two per entry, with the clause, the commit, the test block, the expected output, the limits and the benchmark of each. This appendix is its index, so that the thesis is self-contained on *what* was done and on where the detail is to be found.

The table below is generated from the registry rather than typed, which matters for a reason of method: an index maintained by hand drifts from the document it indexes, and the drift is invisible. Every row carries three things read from the entry itself. The label `M.N.K` identifies the change, where N is the block of the roadmap it closes in whole or in part and K counts within it; the clause is the obligation the entry answers to; the commit is the one that carries it in the fork `Berrutz/tight`. The titles are translated from the registry, and are otherwise the ones under which an entry is filed there.

Three conventions are worth recalling before reading it. The numbering of K is *chronological*, so the entries of one block appear in the order in which the work reached them and not in order of importance. An entry may close only part of a block, and a block may be reopened by a later clause: `G.2` carries twenty-seven entries for that reason, and it was reopened twice after having once been declared closed, by `M.2.25` and by `M.2.27`. And a change that adds an operation the library never had cites the clause that *requires* the operation, not one it was violating.

The nine defects of the kernel are not listed here. They are not changes to `TIGHT`, they carry a different label, and Table 5.1 gives them in full with the site of their repair.

Table A.1: The seventy-one changes to TIGHT, in the order in which they were made, with the clause each one answers to and the commit that carries it in the fork. Every figure in the table is read from that document rather than typed by hand.

Entry	What it changes	Clause	Commit
M.1.1	The constants empty and entire	Cl. 10.5.2	7c6b978
M.2.1	Empty absorbing in the arithmetic operations	Cl. 12.10.2	dd7cd72
M.2.2	Conformant product with infinite endpoints	Cl. 10.4	e0afc8c
M.2.3	Conformant division with a divisor touching zero	Cl. 10.4	cae156a
M.2.4	<i>tightest</i> division, dividing the endpoints directly	Cl. 12.10.2	6c174a5
M.2.5	Empty outside the four arithmetic operators	Cl. 12.10.2	2a44570
M.2.6	The domain filter in the elementary functions	Cl. 10.4	0975740
M.2.7	The domain filter as a public operation	Cl. 10.4	1201e35
M.2.8	The missing numeric functions: rad , mag , mig	Cl. 10.5.9	2f2071a
M.2.9	tan at the poles	Cl. 10.4	a354578
M.2.10	The domain of pow	Cl. 10.4	7f88aaf
M.2.11	The lower endpoint of pow rounded below zero	Cl. 10.4	392536f
M.2.12	The base of pow with a negative zero	Cl. 10.2	ca28443
M.2.13	Empty absorbing in the elementary functions	Cl. 12.10.2	5d0f187
M.2.14	Exponentials rounded below the floor of their image	Cl. 10.4	6f2b476
M.2.15	Exponentials overflowing past realmax	Cl. 10.4	1a0642c
M.2.16	Exponentials underflowing to zero	Cl. 10.4	3cf7f86
M.2.17	Empty absorbing in pown and hypot	Cl. 12.10.2	af2cc96
M.2.18	pown with a negative exponent	Cl. 10.5.3	d3b0cfd
M.2.19	The degenerate infinite interval	Cl. 10.2	328e5f5
M.2.20	The midpoint rounded to nearest	Cl. 10.5.9	f04e68a
M.2.21	isMember and non-finite arguments	Cl. 10.5.10	b173a8f
M.2.22	Equality as equality of sets	Cl. 10.5.10	d225098
M.2.23	The enclosure of atan2 instead of the complement	Cl. 10.4	6119d8f
M.2.24	rsqrt took the same endpoint twice	Cl. 10.2	d4fc879
M.2.25	fma , the last missing row of Table 9.1	Cl. 10.5.3	f8eb980
M.2.26	The six integer functions, and Table 9.1 closes	Cl. 10.5.3	cb6e12b
M.2.27	Containment of pow with a tiny exponent	Cl. 10.4	0f3b040

continued on the next page

continued from the previous page

Entry	What it changes	Clause	Commit
M.3.1	<code>numsToInterval</code> , the constructor that must be able to fail	Cl. 10.5.8	99feab5
M.3.2	<code>textToInterval</code> : outward conversion, and refusing what is not a literal	Cl. 10.5.8	dcf83c3
M.3.3	The special values of a literal, recognised instead of refused	Cl. 12.11.3	41de17d
M.3.4	Rational literals, routed through <code>fromRational</code>	Cl. 9.7	0665de0
M.3.5	The uncertain form, which is arithmetic on integers	Cl. 12.11.5	657c2db
M.4.1	The three unary predicates	Cl. 10.5.10	51a49f2
M.4.2	The four order relations, and the false friend	Cl. 10.5.10	738d1cd
M.4.3	The three set relations	Cl. 10.5.10	85d744f
M.5.1	<code>intersection</code>	Cl. 10.5.7	50a0004
M.5.2	<code>convexHull</code> , where the empty interval does not absorb	Cl. 10.5.7	294557d
M.6.1	<code>mulRevToPair</code> , the two-output division	Cl. 10.5.5	ab44575
M.6.2	<code>cancelMinus</code> , where Entire must stay diagnostic	Cl. 10.5.6	68ba034
M.7.1	<code>sqrRev</code> , <code>absRev</code> , <code>coshRev</code> : the even functions	Cl. 10.5.4	4f2831b
M.7.2	<code>pownRev</code> , and the q -th root the library did not have	Cl. 10.5.4	de2361c
M.7.3	<code>sinRev</code> , <code>cosRev</code> , <code>tanRev</code> : the periodic ones	Cl. 10.5.4	45b7200
M.7.4	<code>mulRev</code> , where the hull is taken last	Cl. 10.5.4	0db9413
M.7.5	<code>pownRev</code> with a single rounding	Cl. 10.5.4	abd9b04
M.7.6	<code>powRev1</code> and <code>powRev2</code>	Cl. 10.5.4	27c87ca
M.7.7	<code>atan2Rev1</code> and <code>atan2Rev2</code>	Cl. 10.5.4	2bce4d5
M.8.1	The exception register	Cl. 7.5.3	049475a
M.8.2	The constructors signal	Cl. 12.12.7	8f60ad7
M.9.1	The type, the five decorations, <code>Nal</code>	Cl. 11.2	34b2d0b
M.9.2	The operations of Cl. 11.5	Cl. 11.5	9e9db41
M.9.3	Propagation, and the min-rule	Cl. 11.6	cbfc568
M.9.4	The decorated reverse operations	Cl. 11.7.1	20ae9cc
M.9.5	<code>atan2</code> : touching the cut is not crossing it	Cl. 11.6	d6a89ae
M.10.1	The decorated constants and type conversion	Cl. 12.12.1	4bc9b40
M.10.2	The decorated constructor from numbers	Cl. 12.12.7	99e9af7
M.10.3	The decorated constructor from text, and the decorated literal	Cl. 12.11.4	b99bb32

continued on the next page

continued from the previous page

Entry	What it changes	Clause	Commit
M.10.4	The four reductions, correctly rounded	Cl. 12.12.12	274bfc5
M.10.5	The sign of a zero, and the entry reopened	Cl. 12.12.8	b0b3769
M.10.6	The IEEE 754-2019 flags in the reductions	Cl. 12.12.12	b18fea6
M.11.1	The exact representation, and the recovery requirement	Cl. 13.4	3fff9dd
M.11.2	Output, and the string that must enclose	Cl. 13.3	c763970
M.12.1	The names of the standard, as a forwarding layer	Cl. 12.2	6620195
M.12.2	Invalid Level 3 objects	Cl. 14.3	d32be02
M.12.3	The interchange encoding	Cl. 14.4	4ab9788
M.12.4	The documentation of tightness	Cl. 12.10.3	1407f73
M.12.5	The claim and the questionnaire	Cl. 4.3	45eb3a4
M.13.1	The bound that falls on an exact double	Cl. 12.10.2	3e76f16
M.13.2	The two rules the bisection cannot find	Cl. 10.5.4	22339eb
M.13.3	The predicate decided in the logarithm	Cl. 10.5.4	7358a29
M.13.4	The periodic functions decided on the $\pi/2$ grid	Cl. 10.5.4	f8a300f

Appendix B

Reproducing the Measurements

Every figure of Chapter 8 can be produced again, and this appendix gives the commands. Three things are needed before starting, and they are stated because each of them was at some point the reason a run did not work.

The first is a Linux toolchain with C++20 and AVX2. The library is built and measured under WSL Ubuntu, which is where its dependencies expect to be. The second is that the build must happen *outside* the source tree of the library: the directory `tight/build` carried in the repository is pinned to a commit that is no longer reachable, and an incremental build in it would try to fetch a dead object. The third is that `libmpfr-dev` and `libboost-test-dev` are *not* required: the accuracy suite of the upstream project is made optional by `TIGHT_BUILD_TESTS`, and the macros of `Boost.Test` are replaced by a small stand-in kept beside the plugin. Both were deliberate, so that reproducing a measurement does not depend on administrative rights.

B.1 Building the two compilations

The two builds differ by one macro and by nothing else. Listing B.1 produces them. `TIGHT_FETCH_WORST_CASES` switches off the download of the reference data the accuracy suite uses, which is not needed here.

Two variations of that recipe are used elsewhere in the work and are worth recording. Passing `-DNFG_GIT_TAG=<hash>` builds against another commit of the kernel, which is how the defects N.1 and N.9 are reproduced through the library. Omitting `CMAKE_BUILD_TYPE` leaves the assertions live, which is the build in which the constructor rejects a malformed interval: with the flag on it runs to the end, and with the flag off it aborts inside the block of M.2.6, which is the defect that entry repairs rather than a broken build.

```

# the conformant build
cmake -S tight -B ~/tight-build-strict \
      -DTIGHT_IEEE1788_STRICT=ON \
      -DTIGHT_BUILD_TESTS=OFF -DTIGHT_FETCH_WORST_CASES=OFF
cmake --build ~/tight-build-strict

# the non-conformant one
cmake -S tight -B ~/tight-build-plain \
      -DTIGHT_BUILD_TESTS=OFF -DTIGHT_FETCH_WORST_CASES=OFF
cmake --build ~/tight-build-plain

# the test blocks of the registry: one line per M.N.K marker
~/tight-build-strict/bin | grep 'test M.'
```

Listing B.1: Building the conformant and the non-conformant library. The only difference between the two is `TIGHT_IEEE1788_STRICT`; everything else, including the commit of the kernel, is the same.

B.2 Running the conformance corpus

The corpus is prepared once and then run as often as needed. Listing B.2 does both. The preparation clones ITF1788, creates the virtual environment, applies the two one-line adaptations without which a framework written in 2018 does not start under Python 3.12, and installs the plugin inside the copy of the framework; it is idempotent, and skips what is already there.

```

# once, from the directory of the verification platform
bash scripts/setup_itf1788.sh

# every time
bash scripts/run_corpus.sh
```

Listing B.2: Preparing and running the external corpus. The run takes about ten minutes, almost all of it in the forty invocations of the compiler.

The script prints the failures of each build, the residue by operation, and the two numbers that matter: how many assertions fail only without the flag, which are the ones the flag repairs, and how many fail only with it, which are the ones it breaks. The second number is zero throughout this work, and it is computed with `comm` on the two sorted lists rather than as a difference of totals, for the reason given in Section 4.1.1.

To measure two versions of the library against each other instead of the two builds of one version, `compare_libs.sh` takes two commits and does the same thing across them. That is the third axis of Table 4.5, and it is how the entries of G.13 were measured.

B.3 Relaunching a benchmark

The benchmark takes the measurement mode as its last argument, one per family of operations, and the shape of the run as the first four. Listing B.3 shows the invocation used for the tables of Chapter 8 and what each number means.

```
# from the directory of the benchmark
# 4096 distinct operands, 32 passes per round, 9 rounds, 5 interleaved runs
bash run_casi.sh 4096 32 9 5 casi          # addition
bash run_casi.sh 4096 32 9 5 casi-mul     # product
bash run_casi.sh 4096 32 9 5 casi-div     # quotient
bash run_casi.sh 4096 1 9 5 casi-costr2   # constructors: inner must drop,
                                           # a literal costs microseconds

python3 agg.py out/casi.tsv              # aggregate: minimum, and the
                                           # median as a noise indicator
```

Listing B.3: Relaunching one benchmark. The four numbers are the size of the operand set, the passes per round, the rounds, and the interleaved runs; the mode selects the family of operations. Every combination is compiled three times, once per instruction alignment, and the figure reported is the minimum over all of them.

Three habits of use are worth stating, because a figure taken without them is not comparable with the ones in this thesis. The `inner` parameter has to fall when the operation is expensive: with a text literal at about three microseconds an operation, the default would run for hours. The aggregation reports the minimum and, beside it, the ratio between median and minimum: when that ratio leaves the neighbourhood of one the cell was disturbed and is measured again rather than averaged. And the raw output of every run is kept in `out/`, one file per mode plus one of the per-operand digests, which is what the divergence column of every table is counted from.

Finally, the property probe that verifies the clauses stated in prose rather than by value is a translation unit of its own, compiled against both builds. It is the check described at the end of Section 7.5: a hundred and twenty-six assertions that pass with the conformant library and abort with the other one, which is the most direct evidence available that conformance is carried by the flag and not by the rest of the code.

Appendix C

The Probe on the Kernel

The defects of NFG reported in Chapter 5 were not established by reading the kernel alone, and not by a corpus either, there being no external suite for a library that provides interval arithmetic as one of several number types for geometric predicates. They were established with a probe: a single translation unit that includes `numerics.h` from the submodule, exercises it, and prints what it found. This appendix says how it is built, what rules govern it, and what its complete run reports, so that every figure of Section 8.1.1 can be obtained again rather than believed.

C.1 What the probe is

The probe is `nfg_probe.cpp`, about a hundred and ten cases arranged in ten blocks, one per family of operations: `repr` for the representation, `inf` for the infinities, then `add`, `sub`, `mul`, `div`, `sqrt`, `fma` and `aux` for the accessory operations, and `rat` for the conversion of an exact rational to a double, which is where `N.9` is seen. Every case is a block marked with a comment naming it, and prints its lines with that name as a prefix, so that a claim in the text and the evidence for it are connected by a string that can be searched for.

It lives in the repository of the thesis and not in the fork of NFG, which is a deliberate separation: the submodule stays as the upstream wrote it, and the instrument that judges it is ours.

C.2 Building it on the three paths

The kernel selects its implementation from the compiler macros, so the probe has to be compiled three times to see all of it. The three commands are those of Listing C.1, and the figures of Table 8.1 were obtained by running each of them at two optimisation levels, `-O2` and `-O0`. The counts do not depend on the level, which is worth checking rather than assuming: a defect that disappeared at `-O0` would be a defect of the compiler and not of the kernel.

```
# AVX2
g++ -O2 -std=c++20 -frounding-math -mavx2 -mfma \
    -I NFG/include nfg_probe.cpp -o /tmp/p && /tmp/p

# SSE2:      replace the SIMD flags with  -msse2
# pure scalar: replace them with  -mno-avx -mno-avx2 -mno-sse2

# the full matrix of N.1, three paths by two levels, in one command
bash run_div_matrix.sh
```

Listing C.1: Building the probe on the three compilation paths, from the directory that holds the probe. The flag `-frounding-math` is not optional: without it the compiler is entitled to assume the default rounding direction, and the kernel installs another one.

C.3 The rules the probe obeys

Four rules make its output evidence rather than an anecdote, and they were adopted one at a time, each after something had gone wrong without it.

A block that has been analysed is never removed. The probe only grows. Every statement about the kernel, in this thesis and in the analysis document, must remain reproducible by recompiling the current probe, which would not be the case if a block were deleted once its defect had been repaired. The cost is that the probe reports on things nobody is looking at any more; the benefit is that a reader who fetches the repository today can check a claim written months ago.

A claim that can be checked by the program is checked by the program. The stronger statements are not left to the reader's comparison of two printed intervals but are decided inside the probe by a predicate that prints its own verdict. There are six: the enclosure of the quotient, which is N.1; the test $\underline{y}^2 \leq 2 \leq \bar{y}^2$ for the square root, which is enclosure written without trusting any other square root; a reference computed in extended precision for the square and the cube, which is N.3; the identity that a fused multiply-subtract must satisfy, which is N.4; the report of a NaN hidden in either endpoint, which is N.5 and is the last column of Table 8.1; and the round trip of an exact rational through a double, which is N.9. A verdict of this kind is what makes a row of Table 8.1 a count of checks and not a count of lines.

The exception flags are cleared before an operation and read after it. The helpers that do so use the complete mask of the five flags of IEEE 754-2019: invalid operation, division by zero, overflow, underflow and inexact, under 754 Cl. 7.2 to 754 Cl. 7.6. An earlier version watched the invalid operation alone, and missed the spurious division by zero that N.4 was raising in an operation containing no division.

The scope of the flag checks is declared. Cl. 1.4 of IEEE 1788-2015 puts the behaviour of interval operations with respect to those flags outside its scope, and does not require an implementation to expose the inexact flag; 754 Cl. 7.6 does require it of a conforming floating-point implementation. Since what is being judged here is the kernel

against IEEE 754-2019, the probe checks the inexact flag too, and that check is separate from the enclosure requirements of IEEE 1788-2015. A reader must not take a flag count of this appendix as evidence about interval conformance, and Section 8.3 says so again where the counts are used.

C.4 What a complete run reports

At the head of the fork, on the AVX2 build, the complete run raises the invalid operation twenty-six times, inexact twenty-nine, overflow four times, division by zero once and underflow twice. Those counts are occurrences of the marker in the output, not distinct causes, and they are quoted in Section 8.1.1 against the counts of the version before N.4, which were twenty-seven, thirty-three and three respectively.

Two of those lines are worth transcribing, because they are the clearest demonstration in the whole work that the kernel rounds outwards as it claims. With M the largest finite binary64 number,

$$\begin{aligned} [0, M] + [0, M] &\Rightarrow [0, +\text{inf}] && \text{with overflow and inexact,} \\ [-M, -M] - [M, M] &\Rightarrow [-\text{inf}, -M] && \text{with overflow and inexact.} \end{aligned}$$

Neither result is a failure. An enclosure that cannot be represented must grow, never shrink, and 754 *Cl.* 7.4 prescribes exactly that the overflowed endpoint be carried to infinity in the direction the rounding attribute demands. The flags say that something was lost; the intervals say that nothing false was gained.

Appendix D

The Operation Map

Section 4.2 describes the plugin that binds the external corpus to the library: for each operation the corpus knows, a fragment of C++ saying which function of TIGHT answers it. Table D.1 is that binding written out, with one column added. Reading a row from left to right takes you from a line of the corpus, through the function that runs it, to the entries of the registry that made that function conformant, and from there, through Table A.1, to a commit. It is the path a reader has to walk to check any claim of Chapters 5 to 7 without taking our word for anything.

Three conventions govern the third column. A tick means that TIGHT provides the operation and that its semantics is the one the standard prescribes. A circle means that the standard *recommends* the operation rather than requiring it, and that the conformance claim of M.12.5 states in writing that it is not provided: this is the case of **overlap**, whose Cl. 10.6.4 says *should*, and of the overlap literals that go with it. An asterisk means that the corpus exercises something the standard does not require at all, so that a failing line there is not a non-conformance: the unary plus is the clearest case, Table 9.1 not listing it.

D.1 What the table does not prove

The table is built on the interface of TIGHT, and an argument that runs along it can only show that TIGHT does what TIGHT has. That is not conformance, and the distinction is worth making sharply because the table is persuasive to look at.

Of the seventy-one changes of the registry, fifty-three appear in the last column of the table and eighteen do not, and the two reasons for an absence are worth keeping apart.

Seven are absent because the corpus has no line that reaches them: M.2.7, the domain filter exposed as a public operation; M.2.24, which repairs a defect inherited from upstream; M.11.1 and M.11.2, the textual output of Cl. 13; M.12.2 and M.12.3, which answer Cl. 14.3 and Cl. 14.4; and M.12.4, the documentation of tightness that Cl. 12.10.3 requires. A reader who took this table for the whole picture would conclude that they did not need doing.

The other eleven are exercised by the corpus and simply do not appear in print. The map names, for each operation, the entries that decided its behaviour rather than every entry that touched it, which leaves nine standing behind a row without being printed on it: M.2.14 and M.2.15 behind the exponentials, M.7.2 to M.7.6 behind the reverse operations, and M.13.2 and M.13.3 behind their tightness. The last two, M.13.1 and M.13.4, are cut by the rule stated in the caption, which prints at most four entries to a row.

Coverage is established by the opposite procedure, and it is the subject of Section 7.5: the 192 sentences of the standard containing *shall*, judged one at a time, together with a translation unit that names every row of the tables of operations, separately for the bare and the decorated type, and whose only requirement is that it compile. This table says how the corpus reaches the library; that audit says whether the library reaches the standard.

D.2 Three names that still mislead

Three operators of TIGHT have names that suggest one operation of the standard and perform another. All three were left as they are, deliberately, and each has the conformant operation beside it under its own name; they are listed here because a reader of the source will meet them before meeting any explanation.

`operator&` and `operator|` are `min` and `max`, not intersection and convex hull, which are spelled out in full and were added by M.5.1 and M.5.2.

`operator<` and `operator<=` are not `less` and `precedes` of Table 10.3. Measured against that table, in M.4.2, they turn out to be `strictPrecedes` and `precedes`: they are not wrong, they are misnamed, and the surrounding geometry uses them as a sign filter. That is why they were not changed, whereas `operator==`, which was genuinely wrong in that $[1, 2] = [1, 2]$ came out false, was.

`atan2` in its bare form returns a pair of an interval and a boolean, the boolean being the one of M.2.23; the conformant enclosure is a separate function, and the decorated version of M.9.5 returns a single decorated interval.

Table D.1: The 62 operations the corpus exercises, the function of TIGHT the plugin binds each one to, and the entries of the registry that made it conformant. ✓ is provided and conformant, ○ is recommended by the standard and deliberately not provided, and * is exercised by the corpus without being required. **none** in the last column means that the operation was already conformant and needed no change; in the second it means that TIGHT does not provide the operation at all. Where more than four entries touch one row only the first four are listed.

Corpus operation	TIGHT	Entries
Constants and constructors		

continued on the next page

continued from the previous page

Corpus operation	TIGHT	Entries
empty_interval	constants::empty	✓ M.1.1
entire_interval	constants::entire	✓ M.1.1
inf_sup_interval I(a, b)	Interval(double, double)	✓ M.2.19, M.8.2
nai_interval	DecoratedInterval::nai()	✓ M.9.1
numsToInterval	numsToInterval	✓ M.3.1
textToInterval	textToInterval	✓ M.3.2, M.3.3, M.3.4, M.3.5
decorated_*	DecoratedInterval	✓ M.9.1, M.10.1, M.10.2, M.10.3

Basic arithmetic operations (Table 9.1)

op_pos	none	* none
op_neg	neg, operator-	✓ M.2.1
op_add	add, a + b	✓ M.2.1
op_sub	sub, a - b	✓ M.2.1
op_mul	mul, a * b	✓ M.2.1, M.2.2
op_div	div, a / b	✓ M.2.1, M.2.3, M.2.4
op_recip	recip, inv	✓ M.2.4, M.2.5
op_sqr	sqr	✓ M.2.5
op_sqrt	sqrt	✓ M.2.5, M.2.6
op_fma	fma	✓ M.2.25

Powers, exponentials and logarithms

op_pown	pown(x, int)	✓ M.2.17, M.2.18
op_pow	pow(x, y)	✓ M.2.10, M.2.11, M.2.12, M.2.27
op_exp / op_exp2 / op_exp10	exp / exp2 / exp10	✓ M.2.13, M.2.16
op_log / op_log2 / op_log10	log / log2 / log10	✓ M.2.6, M.2.13

Trigonometric and hyperbolic functions

op_sin op_cos op_tan	sin cos tan	✓ M.2.9, M.2.13
op_asin op_acos	asin acos	✓ M.2.6, M.2.13
op_atan	atan	✓ M.2.13
op_atan2	atan2	✓ M.2.23, M.9.5
op_sinh op_cosh op_tanh	sinh cosh tanh	✓ M.2.13
op_asinh	asinh	✓ M.2.13
op_acosh	acosh	✓ M.2.6
op_atanh	atanh	✓ M.2.6

Integer and rounding functions*continued on the next page*

continued from the previous page

Corpus operation	TIGHT	Entries	
op_sign op_ceil op_floor op_trunc	sign ceil floor trunc	✓	M.2.26
op_roundTiesToEven	roundTiesToEven	✓	M.2.26
op_roundTiesToAway	roundTiesToAway		
Absolute value, minimum and maximum			
op_abs	abs(x)	✓	none
op_min	min(a,b)	✓	M.2.5
op_max	max(a,b)	✓	M.2.5
Numeric functions of intervals			
op_inf	inf(x)	✓	M.12.1, M.10.5
op_sup	sup(x)	✓	M.12.1, M.10.5
op_mid	mid(x)	✓	M.2.5, M.2.20, M.10.5
op_wid	wid(x)	✓	M.2.5, M.10.5
op_rad	rad(x)	✓	M.2.8, M.10.5
op_mag	mag(x)	✓	M.2.8, M.10.5
op_mig	mig(x)	✓	M.2.8, M.10.5
Boolean functions of intervals			
op_isMember	isMember(m, x)	✓	M.2.21
op_equal	equal(x, y)	✓	M.2.22, M.12.1
op_subset	subset(a, b)	✓	M.4.3
op_interior	interior(a, b)	✓	M.4.3
op_disjoint	disjoint(a, b)	✓	M.4.3
op_isEmpty	isEmpty(x)	✓	M.2.1
op_isEntire	isEntire(x)	✓	M.4.1
op_isNaI	isNaI(v)	✓	M.9.2
op_less op_strictLess	less strictLess	✓	M.4.2
op_precedes op_strictPrecedes	precedes strictPrecedes	✓	M.4.2
op_isCommonInterval	isCommonInterval(x)	✓	M.4.1
op_isSingleton	isSingleton(x)	✓	M.4.1
op_overlap	none	○	M.12.5
Set operations			
op_intersection	intersection(a, b)	✓	M.5.1
op_convexHull	convexHull(a, b)	✓	M.5.2
Cancellative, reverse mode, decorations, exceptions, reductions			
op_cancelMinus op_cancelPlus	cancelMinus cancelPlus	✓	M.6.2

continued on the next page

continued from the previous page

Corpus operation	TIGHT	Entries
reverse mode (*Rev)	sqrRev, absRev, coshRev, pownRev, sinRev, cosRev, tanRev, mulRev, mulRevToPair, powRev1, powRev2, atan2Rev1, atan2Rev2	✓ M.6.1, M.7.1, M.7.7, M.9.4, ...
decorations (com/dac/def/trv/ill, setDec, newDec, decorationPart)	Decoration.hpp, DecoratedInterval.hpp	✓ M.9.1, M.9.2, M.9.3
overlap_* (overlap literals)	none	○ none
exceptions (UndefinedOperation, IntvlOverflow, ...)	signalException, testException, setExceptionHandler	✓ M.8.1, M.8.2
reductions (sum, sumAbs, sumSquare, dot)	Reduction.hpp	✓ M.10.4, M.10.6

Bibliography

- [Att20] Marco Attene. Indirect predicates for geometric constructions. *Computer-Aided Design*, 126:102856, 2020. doi:10.1016/j.cad.2020.102856.
- [Att26] Marco Attene. NFG: Numbers for geometry. <https://github.com/MarcoAttene/NFG>, 2026. Header-only C++ library. Version analysed: commit 52a1d3d, 16 June 2026.
- [BFR24] Luis Benet, Luca Ferranti, and Nathalie Revol. A framework to test interval arithmetic libraries and their IEEE 1788-2015 compliance. *Concurrency and Computation: Practice and Experience*, 36(1), 2024. arXiv:2307.06953, doi:10.1002/cpe.7856.
- [BMP06] Hervé Brönnimann, Guillaume Melquiond, and Sylvain Pion. The design of the Boost interval arithmetic library. *Theoretical Computer Science*, 351(1):111–118, 2006. doi:10.1016/j.tcs.2005.09.062.
- [HK04] Werner Hofschuster and Walter Krämer. C-XSC 2.0 – a C++ library for extended scientific computing. In *Numerical Software with Result Verification*, volume 2991 of *Lecture Notes in Computer Science*, pages 15–35. Springer, 2004. doi:10.1007/978-3-540-24738-8_2.
- [IEE15] IEEE Microprocessor Standards Committee, IEEE Computer Society, New York, NY, USA. *IEEE Standard for Interval Arithmetic*, 2015. doi:10.1109/IEEESTD.2015.7140721.
- [KM86] Ulrich W. Kulisch and Willard L. Miranker. The arithmetic of the digital computer: A new approach. *SIAM Review*, 28(1):1–40, 1986. doi:10.1137/1028001.
- [LTWvG⁺06] Michael Lerch, German Tischler, Jürgen Wolff von Gudenberg, Werner Hofschuster, and Walter Krämer. filib++, a fast interval library supporting containment computations. *ACM Transactions on Mathematical Software*, 32(2):299–324, 2006. doi:10.1145/1141885.1141893.
- [Mic19] Microprocessor Standards Committee, IEEE Computer Society, New York, NY, USA. *IEEE Standard for Floating-Point Arithmetic*, 2019. doi:10.1109/IEEESTD.2019.8766229.
- [Moo66] Ramon E. Moore. *Interval Analysis*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, NJ, USA, 1966.

- [Neh14] Marco Nehmeier. libieeep1788: A C++ implementation of the IEEE interval standard P1788. In *2014 IEEE Conference on Norbert Wiener in the 21st Century (21CW)*, pages 1–6, 2014. doi:10.1109/NORBERT.2014.6893854.
- [RR05] Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11(4):275–290, 2005. doi:10.1007/s11155-005-6891-y.
- [SAP25] Federico Sichetti, Marco Attene, and Enrico Puppo. TIGHT intervals for provably correct geometric computation. In M. Comino Trinidad, C. Mancinelli, D. Cabiddu, and D. Giorgi, editors, *Proceedings of Smart Tools and Applications in Graphics (STAG)*, 2025. URL: <https://fsichetti.gitlab.io/papers/tight.pdf>.
- [SBF⁺22] David P. Sanders, Luis Benet, Luca Ferranti, et al. IntervalArithmetic.jl. Zenodo, 2022. doi:10.5281/zenodo.6337817.
- [She97] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997. doi:10.1007/PL00009321.
- [SZG22] Alexei Sibidanov, Paul Zimmermann, and Stéphane Glondou. The CORE-MATH project. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, 2022. doi:10.1109/ARITH54963.2022.00014.