**SORBONNE UNIVERSITÉ**
**Faculté des Sciences et Ingénierie**
Laboratoire d'Informatique de Paris 6

HABILITATION À DIRIGER DES RECHERCHES

**spécialité : Informatique**

présentée et soutenue publiquement le 22 mai 2019 par

Christoph Quirin LAUTER

# Au-delà de l'arithmétique flottante IEEE754

**(Beyond IEEE754 Floating-Point Arithmetic)**

| | | |
|---|---|---|
| Après avis de : | Sylvie BOLDO | Directrice de Recherches INRIA |
| | Miloš ERCEGOVAC | Distinguished Professor, UCLA |
| | Philippe LANGLOIS | Professeur à l'Université de Perpignan |

et devant le jury composé de :

| | |
|---|---|
| Sylvie BOLDO | Rapportrice |
| Jean-Guillaume DUMAS | Examinateur |
| Miloš ERCEGOVAC | Rapporteur |
| Stef GRAILLAT | Examinateur |
| Philippe LANGLOIS | Rapporteur |
| Siegfried RUMP | Examinateur |

*To Olya*

# Contents

# Introduction

At the time these lines are written, the world's largest computer in terms of processing power is the Summit computer [1]. It provides a peak Linpack performance of $200.795$ petaflops, meaning that it is able to execute $200795000000000000$ Floating-Point operations *per second*.

This is certainly an impressive number and it would be easy to announce other impressive "performance" numbers of this kind: numbers of all floating-point operations executed by all supercomputers in the world, all computers of all kinds, including Internet-Of-Thing devices and so on.

However, impressing numbers do not tell the whole story. We should ask ourselves: what is the value in the results computed with all these operations? Is there any human being able to verify the correctness of these computations? We would hope that all these operations do serve a purpose and end up computing meaningful results. However, are they all necessary? Or are there parts of the different algorithms that yield to results that get immediately overwhelmed by other terms?

In a couple of words, Floating-point numbers are the Computer Science view of what engineers call *scientific notation* for a real number: quantities are expressed as an order of magnitude, given as an exponent of a fixed base power, and a significand term that contains the first decimals of the quantity right to the decimal point. For example, huge quantities such as Avogadro's constant will be expressed with larger powers, $6.02214086 \cdot 10^{23}$ for instance. Tiny quantities, such as the charge of an electron, will have the same number of decimals in their significand – the same precision – but will be expressed with a negative exponent, $1.60217662 \cdot 10^{-19}$ for the example of an electron's charge.

Computing with Floating-point numbers is intrinsically inexact: a computer's memory is finite, so only a finite number of decimals in the significand may be maintained. This requires rounding; for example, in decimal, the result of the operation *divide* $1.0$ *by* $3.0$ will not yield exactly one third but some approximated number like $10^{-1} \cdot 3.33333333$. In general, floating-point numbers are hence approximations to unknown mathematical quantities, affected by errors.

The Summit supercomputer executes roughly $200 \cdot 10^{15}$ floating-point operations per second. An overwhelming majority of these operations will provoke a rounding error. All errors combine, in ways that may be easy or hard to analyze, and finally affect the results. So is there any hope to prove correctness for the algorithms run on these computers?

There is hope but things aren't simple. Floating-point arithmetic is mainly defined by the IEEE754 Standard on Floating-Point Arithmetic [117]. This standard, first published in 1985 for the so-called microcomputers of that time, is now commonly accepted and supported on a wide range of systems. It has been the first standard to precisely define what the expected

1. `https://www.top500.org/green500/lists/2018/11/` retrieved 2019-02-25

results for a certain set of well-defined floating-point operation are and it has enabled proofs on floating-point programs [197]. For instance, the standard requires the result of the division example above – *divide* $1.0$ *by* $3.0$ – to be correctly rounded, i.e. rounded as if the result were computed infinitely precisely and then rounded to a floating-point number.

However, we do think that even if the IEEE754 Standard has been a major breakthrough towards reliable arithmetic and computing in general, in a world striving for exaflop computing, it has become too fine-grained and almost not wide enough in scope. The IEEE754 Standard defines single operations, such as addition between two floating-point numbers, whence induces a bottom-up approach: more complex algorithms are to be built from simpler bricks. Algorithms are first written, then analyzed for their resulting accuracy. In an exaflop world, a top-down approach, where algorithms are generated to just the right accuracy, might be more appropriate. We think we should conceive a modern floating-point environment that is firmly based on IEEE754 but that extends it where needed to allow for this top-down approach to high level operations, *a priori* accuracy specifications and currently uncommon mixtures of numbers in floating-point and fixed-point formats and radices of several kinds.

This manuscript is dedicated to shedding some light on research done in the recent year to cover these aspects. We started by asking ourselves whether the support for IEEE754 was actually complete on classical platforms, such as Linux. We found that certain parts were still missing, in particular with respect to the 2008 version of IEEE754, which had added quite a few operations to the standard. We tried to find algorithms that allow for easy implementation of so-called heterogeneous operations and went on to look into correct rounding of arbitrary-length decimal string to binary floating-point operations. This first binary-decimal mixed radix experience paved the way to a new challenge: providing mixed-radix operations in an IEEE754 extension. We found ways to realize mixed-radix comparisons but we have to admit that our research on a mixed-radix fused-multiply-and-add proved to be harder than we thought.

In parallel, we worked into other directions of IEEE754 extension: extending precision and high-level operations. Working at extending compute precision in a classical IEEE754-compatible manner is reasonable only up to a certain precision point: when precision attains some threshold, existing multiple-precision library provide state-of-the-art solutions with near-optimal complexities. However, there is a certain range just between hardware-covered IEEE754 double precision and software-emulated multiple-precision where significant performance improvements are possible.

With respect to high-level operations, we chose to work on the case of Euclidian norms of arbitrary length vectors. Such operations may seem challenging from a precision perspective in their summation step. We found that this is actually not the case and that performance improvements with a speed-up of up to $4.5$ times with respect to existing solutions may be obtained – while improving accuracy significantly.

These parts of our research follow a pretty classical way of designing floating-point algorithms: it is the human who performs all the necessary steps of accuracy estimation, precision adaptation, hence proving in an *a posteriori* way the correctness of the algorithms for the proposed high-level operations.

This work can be extremely tedious as the part of repetitive accuracy estimate calculations with respect to the creativity part can be pretty high. Humans are the only ones to be creative. Computers are the best at performing repetitive parts.

The implementation of mathematical –mostly elementary– functions, such as $\exp, \sin, \cos$, is a good example for codes where a lot of the implementation steps are mechanical, tedious work. It is, hence, reasonable to no longer write theses codes by hand but to try to utilize computers to generate these codes. We worked on this subject during the Metalibm project funded by ANR[2]. We are now able to generate implementations of univariate functions for accuracies in the range from a couple to about 120 correct bits. The functions to be implemented can be given as expression trees or in the form of differential equations of a certain kind. Almost all generation steps are fully automated, providing a turn-key solution for the user. We provide a meta-level extension to the classical IEEE754 floating-point environment: the user can freely generate implementations for operations they do not find covered by the standard.

The detailed work on this automatic toolchain for the generation of codes for mathematical functions let us understand that there is another arithmetical object that seems completely different from mathematical functions when first compared but that shares a lot of similarities in terms of implementation steps: Linear Time Invariant (LTI) digital filters. Such filters are used in a wide range of applications, ranging from signal processing in telecommunications to controllers for machinery. A toolchain able to implement a digital filter, starting from a frequency response specification and ending up with code in floating-point or, more commonly, fixed-point arithmetic goes through essentially the same steps as the mathematical function code generator: polynomial or rational approximation, coefficient truncation, choice of evaluation scheme and final implementation with precision adaptation.

We hence worked at porting our toolchain from mathematical functions to digital filters. This work required us to find ways to compute certain measures, for instance the so called Worst Case Peak Gain norm, rigorously and with arbitrary accuracy. This opened another field of activity enhancing the classical IEEE754 arithmetic environment: in contrast to IEEE754 arithmetic, where algorithms suffer rounding effects and accuracy results are mostly an *a posteriori* measure, we strive at providing multiprecision high-level operations that achieve *a priori* error control.

These basic bricks can then be easily exploited to solve other arithmetical problems. With other words, we have a two-level approach, where *a priori* error-controlled arithmetic becomes a convenience to then design reliable arithmetic operations in a second step.

This document is organized with respect to the different fields of work we have been involved in:

— Chapter 1 contains a brief summary of our academic work done since our Ph.D. thesis in 2008. We give a summary of our publications since that moment. We also provide a list of our publications in international journals and in international, peer reviewed conferences.

— Chapter 2 is a short introduction to floating-point arithmetic and, in particular, the floating-point environment provided by the IEEE754 standard. In this Chapter, we will give an overview of the changes that took place at the 2008 revision of the standard and look into issues with non-standard exception handling.

— In Chapter 3, we present our research done on enhancing the IEEE754 floating-point environment. We shall look into the implementation of heterogeneous operations, correctly rounded arbitrary length decimal string conversions to binary, mixed-radix conversions and further mixed-radix operations in general, medium-range extended precision floating-point operations and, finally, our Euclidian norm algorithm.

2. Agence Nationale pour la Recherche - French Research Fund

— Chapter 4 is dedicated to our core work on code generation for mathematical functions. We will start with an overview on elementary function implementation, including argument reduction techniques, optimized domain splitting and reconstruction. We will then look at open-ended ways to implement functions which are less well-known and for which no reference code is available. We will finish this Chapter by highlighting other possible use-cases for mathematical function code generation and digital filter tool-chain similarities.

— Chapter 5 covers *a priori* error control in multiprecision floating-point arithmetic. These reliable floating-point arithmetic approaches allow, in a second step, for reliable implementation of fixed-point arithmetic-based algorithms.

— We shall give a concluding overview of where we stand in our research in Chapter 6. We shall then try to give an outlook into what fields of research we would like to direct our interest in the short, medium and long-term range.

# Chapter 1
# Research Summary

*Jupiter omnipotens, audacibus annue coeptis.*

Virgil, *Aeneid*

The main objective of our academic research, done since our Ph.D. defense in 2008, is the *enhancement of the classical floating-point environment*, in different ways and through various means. In this Chapter, we would like to present this work succinctly, before describing it with more detail in the technical Chapters below.

We tend to divide our work into three topics, even though connections surely exist:

— For the first topic, we are concerned with software to support the IEEE754-2008 Standard, implementing all enhancements the 2008 revision brought to the Standard using existing IEEE754-1985 operations or using integer arithmetic and bit manipulation. The heterogeneous operations, taking, for instance, double precision arguments and returning single precision results with one single rounding are an example of such enhancements of the 2008 version of IEEE754.

Starting from there, we further invest into designing floating-point operations that are still based on the IEEE754 floating-point formats –for input or output– but not foreseen in the Standard as such. For example, we have worked on mixed-radix floating-point comparisons that allow decimal and binary floating-point types to be compared without any intermediate rounding.

Finally, for this topic, we have looked into enhancing the floating-point environment through extended precision floating-point arithmetic formats and faithfully rounded high-level operations, such as a faithfully rounded 2-norms of vectors of arbitrary length.

This first research axis is summarized below in Section 1.1.

— The second topic deals with the generation of floating-point code, using software instead of writing, testing and proving the code manually.

We launched research on numerical software –the code generator– that produces other numerical software –the generated code– at the example of mathematical, mostly elementary functions, such as $\exp, \sin, \operatorname{atan}$. We contributed several research papers concerned with the different steps required to be automatized in the implementation of a mathematical function: polynomial approximation, argument reduction through the exploitation of algebraic properties of the function where possible, domain splitting, argument reconstruction and, most importantly, automatized error analysis.

We looked in particular into the case when the function to be implemented is given without a way to evaluate it. For instance, for the function defined as $f(x) = e^x + 17$, simple expression tree evaluation and usage of an existing implementation of the exponential suffices. For functions defined by differential equations, for example with $f''(x) - 2xf'(x) = 0, \quad f(0) = 0, f'(0) = 2/\sqrt{\pi}$, no such easy scheme exists. We worked in cooperation with other researchers to make code generation possible in this case, too.

Our work on this topic of code generation brought us the understanding that code generation for digital Linear Time Invariant filters is surprisingly similar. For instance, the classical Remez algorithm used for approximation of a function by a polynomial has a filter formulation, called Parks-McClellan algorithm [205]. We started investigating toward which extent reuse of algorithmic techniques for code generation and even software reuse is possible.

This second topic is succinctly described in Section 1.2.

— Starting from the observation that code generation for mathematical functions and filters is similar but that the state of the art was more advanced for functions, we looked into the basic bricks required for reliable and automatic design and code generation for filters. In this third research topic, we discovered that rigor in implementation of digital filters mainly stems from rigor in the arithmetic used for the analysis of the numerical object that represents the filter.

In our approach, we use a multi-step approach where we first implement software for the analysis of filters in a way that this code guarantees that its error reliably stays within error bounds which are set in an *a priori* manner. This is achieved with multiprecision floating-point and interval arithmetic through dynamic precision adaptation. Given *a priori*-safe measures, we can then reliably determine the scales used in the fixed-point code used to implement the filter.

This third research topic is summarized in Section 1.3.

At the end of this Chapter, in Section 1.5, we give a comprehensive list of our publications which we have published since our Ph.D. defense, namely the following work: [27, 28, 32, 40, 41, 64, 66, 98–101, 125, 130, 146–150, 156–159, 161–163, 251–254]. We shall shortly describe them in the following three Sections 1.1 through 1.3 as we see fit. The reader interested in a complete list of all our publications, including those ones we published before our Ph.D. defense may refer to the list we give at the end of this manuscript, on page 229.

## 1.1 Advanced operations for the IEEE754 Environment

In [161], we first gave an overview of the challenges in providing complete support for the IEEE754-2008 Standard, in both single and double precision. We evaluated that the introduction of new operations in the 2008 version of the Standard provoked an increase of the number of operations to be supported for these two formats from about 70 operations in the 1985 version to over 350 operations, which is equivalent to a five-fold increase. In the article, we presented a library, which we called `libieee754`, intended to provide open-source software support for IEEE754-2008. We grouped the operations to be provided into groups: the first one consisted of those operations that directly map to IEEE754-1985 operations, the second one consisted of the operations for which emulation sequences with existing IEEE754-1985 operations could be found. This group included the IEEE754-2008 heterogeneous operations, which take arguments in one, larger precision and return a result rounded once to another, lower precision. A third and final group consisted of all higher-level floating-point operations mandated by the IEEE754-2008 Standard that required the use of integer-based emulation to achieve the mandated floating-point behavior. For example, we presented an integer-based algorithm to read in arbitrary length decimal string sequences and to round them once to an IEEE754-2008 binary format. The novelty of this algorithm is that it works within a fixed, known memory space, although it provides correct rounding for input sequences of any length, even for those sequences that require more space than that fixed amount of memory.

Reconsidering our sequences to implement the heterogeneous IEEE754-2008 operations which we had designed using classical properties of floating-point arithmetic, such as Sterbenz' lemma, we discovered that they were both special cases and extensions to another classical floating-point sequence, called Ziv's rounding test. As the literature on Ziv's rounding test left out certain aspects, such as the tight but reliable computation of a certain constant, called the Ziv rounding constant, we revisited Ziv's rounding test in a broader article [66]. In that article, we detail the computation of a tight and rigorous Ziv rounding constant and detail how Ziv's rounding test can be used to implement correctly rounded elementary functions as well as heterogeneous operations. It is worth remarking that the resulting algorithm for the heterogeneous operations is uncommon as it is fully agnostic to the rounding mode. When run in round-to-nearest, it performs Ziv's rounding test and deduces the rounding mid-point, i.e. the decisive point where the rounding changes, from two intermediate results of Ziv's test, hence finishes by providing the result, correctly rounded once to nearest. When run in a directed rounding mode, Ziv's rounding test is not correct in principle: it always answers "correct" and does no longer allow for any distinction. However, in our application, which are the heterogeneous operations, this behavior is exactly the one we need: in the directed rounding modes, the heterogeneous operations are naturally correct.

The conversion from decimal to binary and vice-versa, as it is frequently needed for input and output conversion, requires integer powers of 5 to be computed fast, accurately and often with very little memory spent, e.g. on tables. In [150], we investigated algorithms to compute such powers to any accuracy quickly, using small and exact tables to speed up the process.

Our work on decimal string sequence conversion to binary led us to an investigation that enhances the classical IEEE754-2008 environment through providing *exact* comparisons between binary and decimal IEEE754-2008 formats. Under an exact comparison we understand that it always returns the true mathematical result, not the result obtained when converting –with a rounding– one of the operands to the radix of the operand and then comparing. We published this work for a special case in [27] and then for all possible combinations of IEEE754-2008 formats and comparisons (less, greater, less-or-equal, greater-or-equal, equal etc.) in [28]. There are two keys to implementing fast and correct exact comparisons: first, precomputation of the operands in the given binary and decimal formats that are closest one to another while not being equal. This search of the hardest input is classically known but slightly complicated as floating-point numbers in the decimal formats may admit non-unique representation, as decimal floating-point arithmetic does not always normalize the floating-point numbers. Second, a very tight error analysis of the fixed-point code used to implement the comparisons, bundled with an approach using two tables computed such that the error partially cancels out.

This work on mixed-radix comparisons led us to an investigation whether general, computation mixed-radix operations could be proposed to enhance the IEEE754-2008 environment.

We considered a mixed-radix Fused-Multiply-And-Add (FMA) operation, where each of the inputs and the output could be one of the IEEE754-2008 formats binary64 or decimal64. We intended this FMA to be correctly rounded once, as if no intermediate radix-conversion took place resp. were performed with infinite precision. Similarly to the case of our exact mixed-radix comparisons, we tried to perform a search for the inputs that would lead to the largest amount of precision loss in an actual implementation. The results of this search would hence allow an implementation to be tailored to the right intermediate compute precision. Together with the Ph.D. student Olga Kupriianova, whom we advised, we spent some large amount of time on this problem, and could only partly find solutions. The dire results of this work ended up being published in Olga's thesis [145]. We are going to sketch another solution to implementing a mixed-radix FMA in Chapter 3, Section 3.4.3 of this work.

Our experiments with operations that enhance IEEE754-2008 put us in a constant need of higher precision floating-point formats, in particular in the range between $64$ to $512$ bits of precision. As a matter of course, the MPFR library [1] [86] can be used for multiple precision, even in this middle-end precision range. However, MPFR is, or at least used to be, pretty inefficient in the range up to a couple of hundred of bits. It is designed to obtain best asymptotic complexity values for even higher precisions. In addition, MPFR requires the system it is run on to provided access to dynamic memory allocation. For some applications, this is not acceptable, in particular when the memory subsystem is slow or when the maximum memory consumption needs to be known statically. In [156], we proposed a library intended to address both these issues. Our library has statically known memory consumption, all memory required is held on the stack, and it is particularly optimized for the precision range from $64$ to $512$ bits. It provides formats for precisions from $64$ to $512$ bits in steps of 32 bits, and fully supports all combinations of formats, even when the combination results in heterogeneous input and output precisions. It supports all five basic operations ($+, -, \times, /, \sqrt{}$), FMA, all possible comparisons and the conversions between all supported formats, as well to and from IEEE754-2008 single and double precision. It is fully implemented in a way that enables modern compilers to perform constant propagation and dead code elimination. Overall we provided on over $1300$ functions, partly implementing them through code generation techniques.

With the different previous contributions, we tried to enhance the IEEE754 environment with additional operations that work IEEE754-2008 floating-point inputs –or extended precision formats closely matching IEEE754-2008 formats– but that are not proposed in the Standard as such. All these operations followed the IEEE754 Standard with respect to rounding: they are all supposed to provide their results as if infinite intermediate precision were used. This approach enhances IEEE754 but it might be considered too fine grained: users might not need additional operations working on a small number of arguments. There might rather be a need for high-order operations, and without actually requiring correct rounding. We investigated into this direction and came up with a novel algorithm to compute 2-norms (Euclidian norms) of arbitrary length vectors of floating-point numbers of some IEEE754 format. We presented this work in [101]. Our algorithm provides a faithfully rounded result, is free of spurious underflow or overflow and can be parallelized extremely easily, while maintaining these latter guarantees on its output. As it avoids expensive operations such as floating-point divisions and complex, unpredictable branching, its implementation runs about $4.5$ faster than the previous, reference algorithm. The error analysis we performed for

1. `http://www.mpfr.org/`

the proposed algorithm made exhibit slightly improved bounds for certain cases of doubled-precision compensated addition and, in particular, better bounds on the admissible error on an argument to a floating-point square root such that the square root's result is a faithful rounding of the respective mathematically exact quantity.

We describe the research topics summarized in this Section in more detail in Chapter 3.

We shall add that we also worked on some miscellaneous other floating-point environment topics which we refrain from describing in this thesis. This includes work on rounding mode access [162], recent work on multiple-precision decimal floating-point and interval arithmetic [99, 100] and also work on extending Fused-Multiply-And-Add to correctly rounded 2D scalar products [158].

## 1.2 Floating-Point Code Generation

Our work on floating-point code generation for mathematical functions first required research on some basic algorithmic bricks, in particular concerning the *a posteriori* validation of code implementing the function, once the generation process is finished. We had started this type of basis work just after our Ph.D. defense and we could publish three articles, essential to our further research, with [64] and then [40, 41]. In an implementation of a mathematical function, there are two sources of error: the roundoff error and the truncation error. With [64], we addressed validation from a roundoff error perspective, while with [40,41], we got concerned with the truncation error part.

Based on this ground-laying work, we started working on a mathematical function code generator, called Metalibm. This work was in the sequel integrated in the ANR research project *Metalibm* under the name of Metalibm-Lutetia. We published several articles presenting the general idea and workings of this code generator [32, 148]. In particular with [32], we could show the necessity and feasibility of code generators for mathematical functions in various use cases: we list, for example, adaptation of supported function definition domains and adaptation of accuracy to the user's needs, support of composed functions, such as $e^{x+17} + \sin x$, composed statically at generation time and no longer composed dynamically, support for functions uncommon in classical mathematical libraries (libm), such as for the inverse error function $\mathrm{argerf}$, increase in productivity for libm developers or easier code maintenance. While we presented our fully automatic, open-ended approach from a numerical software design perspective in [148], we worked with colleagues in comparing our approach to their competing, more semi-automatic performance-centric approach in [32]. We shall remark that this later paper received the ARITH2015 Best Paper Award.

We worked in cooperation with several people from industry in an attempt to address certain issues in the GNU mathematical library, in particular with respect to bad worst-case performance and code maintainability. We presented our automatic code generation approach to the open-source community with [146].

Our general publications about our Metalibm code generator were meant to give an overview over this piece of software that has grown pretty large ($> 13000$ lines of code for the core generator). We therefore published certain details in the code generation process in separate papers. In [147], we looked at the case when the function to be implemented does not allow for any algebraic argument reduction but where the function's domain requires the use of several approximation polynomials. Classical domain splitting used a regular splitting grid. As the function's steepness is not constant over the domain, this resulted in polynomials of varying degree for the different subdomains. With our approach, we can equalize the polynomial degree over all subdomains and reduce the number of subdomains.

The regular domain splitting had the advantage that given the function's argument it was easy to deduces the index of the subdomain this argument would fall into and hence which polynomial had to be used to compute the function's image. For more complex domain splittings, a binary tree of comparisons with constants and branches is used in the generated code. While this approach requires only a logarithmic number of branches, it indeed does require branches. Those are not acceptable for certain types of implementations, in particular vectorizable code that applies a mathematical function in parallel elementwise to a vector of arguments. To address this case, we have proposed a novel approach that tries to transpose the regular splitting to the more complex ones: we compute the index of the subdomain by evaluating a certain low-degree polynomial and truncating this polynomial's result to an integer. This work was published in [149].

The generation of code realizing a mathematical function requires itself a means to evaluate the function, for instance in order to run the Remez approximation algorithm to compute approximation polynomials for the function. For uncommon functions, given in input to the generator merely defined e.g. by differential equations, this creates a chicken-and-egg problem. In [163] we proposed a way to address this problem, using software to evaluate functions defined by a certain type of differential equations in order to generate code for these so called D-finite functions. We showed that such a semi-automatic approach allows for efficient code production. We did so at the example of various functions, used for example in high-energy particle physics. We have recently extended this work to the case when the function to be implemented is the functional inverse of such a D-finite function. In regards to this, an article is in preparation.

Even though our work on automatic code generation is still ongoing and is intended to be opened more and more to similar domains for code generation, our Metalibm-Lutetia generator has already delivered valuable results, in particular for applications where the speed in designing code was most important. For example in [157], we described a mathematical library for vector applications, where a function needs to be applied elementwise to vectors. With the help of our code generator, we could design, develop and fine-tune this library, which comprises $9$ functions reaching from exponential over the trigonometric and inverse trigonometric functions to particular functions such as cubic root, in about $2$ weeks of work. This time-frame is extremely low; manual development requires about $1$ man-month per function. Performance-wise, with our vectorized functions, we could obtain speedups of over $250\%$, while we did not use any assembly sequences or intrinsics but only C, coded in such a way to make compiler autovectorization possible. This makes our approach a novel one.

We detail the research topics summarized in this Section in Chapter 4. In that Chapter we also describe the connections between code generation for mathematical functions and for linear time invariant filters we could exhibit.

## 1.3 *A Priori* Error Control in Computer Arithmetic

With the articles we published in function code validation, in particular with [40], we had given ourselves the necessary basic bricks for reliable implementation of mathematical functions. For code generation of reliable implementation of linear time invariant filters, we had no such basic bricks. We strive for reliable implementation of digital filters using fixed-point arithmetic. The basic bricks required for the analysis of the behavior of digital filters at generation time hence need to be reliable, too. As these basic brick algorithms are supposed to be in an automatic tool without human intervention, in addition to the reliability requirement, there is need for the algorithms to be able to guarantee that their error does not exceed a certain *a priori* bound. This bound often is an absolute one. When implemented using floating-point arithmetic, the algorithms therefore need to adapt their internal working precision automatically.

With our work on the so-called Worst Case Peak Gain (WCPG) measure, published in [251], we designed such an algorithm. It reliably computes a certain norm on a linear time invariant filter with an error not exceeding a given *a priori* bound. With this work, a code generator for digital filter no longer has to suffer the impact of truncation and roundoff inherent to floating-point arithmetic, it is up to this enhanced floating-point arithmetic-based algorithm to adapt its internal precisions. The code needs this measure, as it expresses by how much an input signal can get amplified (or attenuated) by a digital filter. Our algorithm for the WCPG measure is a creative mix of different types of arithmetic: the WCPG measure can be expressed as the infinite sum $|\boldsymbol{D}| + \sum_{k=0}^{\infty} \left|\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}\right|$ where $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D}$ are matrices. This sum needs to be properly truncated, in a way compatible with the expected error bound. This first step is performed using an IEEE754 double-precision based eigensolver, as accuracy is not paramount in this step. The approximate result of this eigensolver is then made reliable using interval arithmetic; a reliable truncation order can be derived from this interval arithmetic result. The sum is then evaluated multiple-precision floating-point arithmetic; reusing the already computed eigendecomposition. This floating-point arithmetic is however enhanced with code that adapts precision for the different matrix entries dynamically to satisfy the set roundoff error bounds. Only with such a mix of arithmetics we could realize this algorithm with an *a priori* error bound; plain IEEE754 arithmetic does not seem to suffice.

We recently extended our WCPG measure algorithm to handling a special case when the WCPG measure does not need to capture the amplification of any input signal but only the amplification of signals with a certain frequency envelope [254]. This algorithm is essentially based on our core WCPG algorithm and reliable computation of a digital filter producing that frequency envelope. The new algorithm also supports our *a priori*-error arithmetic concept.

Computing the WCPG measure safely and with easily controllable error is important to make the fixed-point arithmetic used to implement a digital filter reliable, with respect to both absence of overflow and boundedness of the roundoff error – which itself is given *a priori*. An important step in designing a fixed-point algorithm is in determining the point positions, i.e. the implicit scalings applied to the various fixed-point variables, implemented with integer variables. Similarly, it is necessary to check that these integer variables cannot overflow, i.e. flip around in a modular manner. Both targets are antagonistic: by increasing the implicit scale, i.e. by moving the fixed-point variables Most Significant Bit position to the left, overflow becomes less likely and can eventually be avoided. However, at the same time, roundoff error increases as roundoff in fixed-point arithmetic is function of the implicit scale. This task used to be performed manually and in a non-reliable manner through simulations. With [252], we proposed a method to perform this task automatically and reliably, using the WCPG measure. Due to the availability of an *a priori* bound, we can make the Most Significant Bit Positions determined by our algorithm either optimal or over-estimated by at most 1 bit.

We describe the research topics summarized in this Section in more detail in Chapter 5.

We shall add that we have recently published work on reliable verification of digital filter implementations against frequency specifications [253]. This work reuses our WCPG algorithm from [251] and combines it with exact, rational arithmetic to give a reliable test whether the filter satisfies the initial specifications. Even it is an important to filter code generation, we refrain from further describing this algorithm, as it is conceptually different from the arithmetic techniques this thesis focuses on.

## 1.4 Positionning with Respect to Other Research

As already mentionned, our work is organized three main topics: support and extension of the IEEE754 Standard, floating-point code generation and *a priori* error computations. In this Section, we try to position our research, in these three topics, with respect to other research, which is part of state of the art.

**Support and Extension of IEEE754** When published first in 1985, the IEEE754 Standard [116] found its realization in Intel's microprocessor x87, after which it had been modeled [197]. Supporting IEEE754-1985 meant implementing the floating-point operations in hardware, sometimes using software to make the hardware less complex. Work by Figueroa [81] for example shows how IEEE754-1985 operations for the binary32 (single precision) format can be implemented with binary64 (double precision) hardware and mere conversions back and forth between binary32 and binary64. Further work by the same author showed how IEEE754-1985 could be supported in higher level languages [82].

Work by Markstein, Cornea and others [51–53, 185] showed how certain IEEE754 operations, such as division and square root, could be implemented using no hardware but software sequences, leveraging hardware for an fused-multiply-and-add operation and some Newton-Raphson iteration seed operations. This work was showcased for the Itanium family of processors. This fused-multiply-and-add operations, which computes $x \times y + z$ with one single, correct rounding, was not part of IEEE754-1985 but it was later including into IEEE754-2008 [117].

The work initiated and driven by Muller on correctly rounded elementary functions [173] led first to the development of approaches to assess the difficulty of rounding elementary functions, such as $\exp$ to the IEEE754 formats binary32 and, in particular, binary64 [170–174]. In a second step, Hough, Ziv, Defour and others showed that correct rounding of elementary functions was technically and economically feasible [57, 68, 165, 175, 264]. This finally allowed for the inclusion, as a non-mandatory part, of correctly rounded elementary functions into the 2008 version of IEEE754 [117].

The 1985 version of IEEE754 was a binary floating-point Standard only [116]. The work of Cowlishaw showed that the extension of IEEE754 to include decimal floating-point arithmetic was technically feasible and an economically reasonable selling instrument for the Standard [55, 56]. The work of Cornea, Tang and others proposed another, binary significand representation for decimal floating-point formats, allowing for more efficient software implementation [49]. This led to the inclusion of decimal floating-point formats, in two different encodings, in the 2008 version of the IEEE754 Standard [117]. The technical realization of this decimal arithmetic, for example in hardware, was then discussed by Schwarz, Schulte and others [34, 78, 244, 256]. Harrison contributed ways to implement decimal transcendental functions [109].

Algorithms for floating-point summation developed mainly by Rump, Graillat and others and, in particular, rigorously proven correct by Rump [97, 200, 229, 231, 232], showed that so-called errorfree transformations [97, 138, 247, 248], i.e. ways to compute and express the sum or the product of two floating-point numbers as an unevaluated sum of floating-point numbers, are a powerful tool to extend the floating-point environment: with these algorithms, floating-point arithmetic is no longer just an approximation technique with errors everywhere, but can be seen as an exact calculus. Work by Demmel, Riedy and others has led to a proposal to include such errorfree transformations in the upcoming 2019 version of the IEEE754 Standard, most probably as a non-mandatory operation [220]. These approaches compete with the idea put forward by Kulisch to include necessarily hardware-based operations on long accumulators into the IEEE754 Standard [137, 141–144].

Extending the IEEE754-2008 Standard in terms of mixed-radix operations is a domain on which we do not seem to compete with other researchers [28, 125]. Other domains of our work on extending the IEEE754 Standard are more common, such as our work on faithfully rounded 2-norms [101]. Support for binary IEEE754-2008 floating-point arithmetic, including the so-called heterogeneous operations, is available, to our knowledge, only in Intel's proprietary binary floating-point support library. We shall discuss this matter in more detail in Section 3.2.1 of this work.

**Floating-point Code Generation** The idea to generate numerical code instead of designing it manually is not new. Letting the computer take care of the details is an approach longtimes used for VHDL-based hardware description for FPGAs. It can be found in the works of Detrey, Tisserand, de Dinechin and others [62, 67, 72–74, 249]. The members of ANR Evaflo project[2], led by Revol, worked on ways to express the input and output as well as the internal representation of numerical objects to be transformed into code [12].

Code generation has been used as a powerful technique for numerical codes such as Basic Linear Algebra Routines or Fast Fourier Transforms in tools like ATLAS and Spiral by Goto, Püschel and others for quite some time [96, 212, 257]. However, their work concentrates on ways to best express code so to allow optimizing compilers to take full advantage of the parallel nature of modern, superscalar processors. Their work does not analyze the effects on the output accuracy of the generated code in much detail.

At least partially using code generation to design and implement elementary functions has been an idea mentioned by Muller at an informal meeting at École Normale Supérieure de Lyon in 2006 and subsequently followed by several researchers, including in particular Brunie, Revy, Jeannerod and, also, ourselves [30, 33, 126, 127, 165, 218, 219]. The works of Defour already contain basic elements of code generation, as he used to use Maple scripts to compute the parameters of polynomials and tables to be put into code for elementary functions [68].

As we shall discuss in detail in Section 4.3 of this work, code generation for mathematical functions has to overcome a chicken-and-egg problem where code to evaluate the function is needed for the design of code to evaluate the functions. Ways to evaluate differential equations for functions of a certain type, viz. D-finite functions, have been studied by Salvy, Mezzarobba and others for quite some time [15, 190, 191]. We build on their work for open-ended code generation.

2. `http://www.ens-lyon.fr/LIP/Arenaire/EVA-Flo/`

***A priori* Error Computations** Arbitrary and multiprecision arithmetic allows the precision numerical algorithms run in to be chosen freely and, for multiprecision, even dynamically at run-time [86,102]. This concept has been used for quite some time –since the 1970ies– and was popularized by Bailey, Brent, Fousse, Zimmermann, Lefèvre and others [25,86,102,110,128].

However, multiprecision code allows for the adaptation of the precision used in the code only; how this adaptation of precision affects accuracy is not clear. Forward-error analysis allows this this question to be answered [111]. It is mainly performed on particular instances of algorithms, typically for example to ensure the correct rounding of a mathematical function to a dynamically chosen precision [39,86].

For more complex numerical algorithms to solve problems of a less basic nature, such as zero search and refinement with Newton-Raphson iteration or quadrature, what is finally important to be controllable is the output accuracy. Algorithms should adapt their precision to meet this goal. This is the idea behind our *a priori* error computations. The idea has already been applied by Fousse for numerical, multiprecision quadrature [84,85].

Our research presented in Chapter 5 is more inspired by these latter works than by the approach followed by Neumaier, Revol and others where *a priori* error bounds are obtained using interval arithmetic as a means to perform forward error analysis dynamically and to repeat computations with increasing precision as long as the *a priori* accuracy bound is not met [216]. We have however used such dynamic precision adaptation techniques, for example for faithful evaluation of arbitrary functions inside Sollya [42].

## 1.5 Publications

In this Section we give a list of our publications in journals and in the proceedings of peer-reviewed international conferences and peer-reviewed workshops which we have published since our Ph.D. defense in 2008. At the end of this manuscript, on page 229, we shall give a list of all our publications, including those ones that we published before our Ph.D. defense.

**Journal articles**

[100] S. Graillat, C. Jeangoudoux, and C. Lauter. MPDI: A Decimal Multiple-Precision Interval Arithmetic Library. *Reliable Computing Journal*, pages 38–52, 2017.

[28] N. Brisebarre, C. Lauter, M. Mezzarobba, and J.-M. Muller. Comparison between binary and decimal floating-point numbers. *IEEE Transactions on Computers*, 65(7):2032–2044, July 2016.

[101] S. Graillat, C. Lauter, P. T. P. Tang, N. Yamanaka, and S. Oishi. Efficient calculations of faithfully rounded l2-norms of n-vectors. *ACM Trans. Math. Softw.*, 41(4):24:1–24:20, Oct. 2015.

[66] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres. On Ziv's rounding test. *ACM Trans. Math. Softw.*, 39(4):25:1–25:19, July 2013.

[40] S. Chevillard, J. Harrison, M. Joldeş, and C. Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoretical Computer Science*, 412(16):1523 – 1543, 2011. Symbolic and Numerical Algorithms.

[64] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, Feb. 2011.

**Articles in conference proceedings, including peer-reviewed workshops**

[159] C. Lauter. Rigorous polynomial approximation. In *2018 52nd Asilomar Conference on Signals, Systems and Computers*, pages 120–124, 2018.

[125] C. Jeangoudoux and C. Lauter. A correctly rounded mixed-radix fused-multiply-add. In *2018 IEEE 25th Symposium on Computer Arithmetic*, pages 17–24, July 2018.

[130] M. Joldes, C. Lauter, M. Ceberio, O. Kosheleva, and V. Kreinovich. Why Taylor models and modified Taylor models are empirically successful : A symmetry-based explanation. In *Proceedings of the 8th International Workshop on Reliable Engineering Computing REC'2018*, July 2018. Proceedings published online.

[158] C. Lauter. An efficient software implementation of correctly rounded operations extending FMA: $a + b + c$ and $a \times b + c \times d$. In *2017 51st Asilomar Conference on Signals, Systems and Computers*, pages 452–456, 2017.

[253] A. Volkova, T. Hilaire, and C. Lauter. Reliable verification of digital implemented filters against frequency specifications. In *2017 IEEE 24th Symposium on Computer Arithmetic*, pages 180–187, July 2017.

[157] C. Lauter. A new open-source SIMD vector libm fully implemented with high-level scalar C. In *2016 50th Asilomar Conference on Signals, Systems and Computers*, pages 407–411, Nov 2016.

[156] C. Lauter. Easing development of precision-sensitive applications with a beyond-quad-precision library. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 742–746, Nov 2015.

[251] A. Volkova, T. Hilaire, and C. Lauter. Determining fixed-point formats for a digital filter implementation using the worst-case peak gain measure. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 737–741, Nov 2015.

[32] N. Brunie, F. de Dinechin, O. Kupriianova, and C. Lauter. Code generators for mathematical functions. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 66–73, June 2015. Best Paper Award.

[252] A. Volkova, T. Hilaire, and C. Lauter. Reliable evaluation of the worst-case peak gain matrix in multiple precision. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 96–103, June 2015.

[163] C. Lauter and M. Mezzarobba. Semi-automatic floating-point implementation of special functions. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 58–65, June 2015.

[147] O. Kupriianova and C. Lauter. A domain splitting algorithm for the mathematical functions code generator. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1271–1275, Nov 2014.

[148] O. Kupriianova and C. Lauter. Metalibm: A mathematical functions code generator. In H. Hong and C. Yap, editors, *ICMS 2014*, volume 8592 of *LNCS*, pages 713–717. Springer, 2014.

[150] O. Kupriianova, C. Lauter, and J. M. Muller. Radix conversion for IEEE754-2008 mixed radix floating-point arithmetic. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 1134–1138, Nov 2013.

[27] N. Brisebarre, C. Lauter, M. Mezzarobba, and J.-M. Muller. Comparison between binary64 and decimal64 floating-point numbers. In A. Nannarelli, P.-M. Seidel, and P. T. P. Tang, editors, *ARITH 21*, pages 145–152, Austin, Texas, USA, 2013. IEEE Computer Society.

[41] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An Environment for the Development of Numerical Codes. In *Mathematical Software - ICMS 2010*, volume 6327 of *LNCS*, pages 28–31, Sept 2010.

[43] S. Chevillard, M. Joldes, and C. Q. Lauter. Certified and fast computation of supremum norms of approximation errors. In *19th Symposium on Computer Arithmetic*, pages 169–176. IEEE, 2009.

CHAPTER 2

# The IEEE754 Floating-Point Environment

*On résout certains problèmes par approximation, en négligeant de petites quantités.*

Jean le Rond d'Alembert, *French Mathematician*

## 2.1 Introduction

The basic idea behind floating-point arithmetic, i.e. a calculus where numbers are represented by an order of magnitude, expressed as an integer power of some radix, and by a mantissa varying between 1 and that radix, is pretty old [213]. The most early electronic computers also used this system already [6, 221] as it permits both large and small quantities to be represented with the same ease, and, in particular, precision.

However, until the adoption of the IEEE754 Standard on Floating-Point Arithmetic [116], first published in 1985, the precise behavior of the floating-point arithmetic operations of one computer system could heavily differ from the one found on another computer system. Writing portable floating-point programs or giving guarantees on their behavior proved extremely difficult [48].

Since 1985, the IEEE754 Standard on Floating-Point Arithmetic has become adopted by all major hardware vendors, got supported on all systems ranging from smaller processors, as little as some embedded systems, for high end serves, clusters and supercomputers [24]. It has become unthinkable for a hardware vendor to sell Floating-Point Processing Units that are not compatible with the IEEE754 Standard.

The IEEE754 Standard has itself evolved since 1985. A first revision has been published in 2008; this version is the currently binding one [117]. A second revision is under way and expected to be published in 2019. This second revision has minor scope; it is supposed to eradicate certain small flaws in the 2008 version, as well as to correct the definition of one operation. A third revision, supposed to enhance the standard more significantly is scheduled for 2028. An IEEE754 Standard revision is a lengthy process; any proposal for substantial changes to be incorporated in the 2028 version is to be made in the next couple of years. We would like to point out that we actively participate in the current revision process.

The IEEE754 Standard currently defines floating-point numbers in two radices [117], the binary radix $\beta = 2$ and the decimal radix $\beta = 10$. The support for the decimal radix has been a 2008 addition.

Given a radix $\beta$, floating-point numbers are the numbers

$$(-1)^s \cdot \beta^e \cdot t$$

where $s \in \{0, 1\}$ is a sign bit, $e \in \mathbb{Z}$ an exponent and $t$ a significand varying between $1 \leqslant t < 2$ for the binary radix and between $0 < t < 10$ for the decimal radix [117]. That significand, also often called mantissa [197], is written with $k$ digits $t_i \in \{0, \dots, \beta - 1\}$ in the given base $\beta$:

$$t = t_0.t_1t_2t_3 \dots t_{k-1}.$$

In most cases [1], it is required that the significand have a non-zero leading digit $t_0$. Significands that abide by this condition $t_0 \neq 0$ are called normalized. Let us point out that when $\beta = 2$ and the significand is normalized, $t_0$ is necessarily $t_0 = 1$. The leading digit $t_0$ is the only digit of $t$ with a weight $1$ and hence called integer digit. All other digits are called fractional digits [197]. The point separating the integer digit from the fractional digit is fixed but made "floating" through the variable exponent $e$.

The most important parameter for a floating-point format, i.e. a set of floating-point numbers written in the same way, is the parameter $k$ that determines how many digits are stored in a floating-point number's significand. This parameter $k$ is called the precision of the floating-point number, resp. floating-point format. The greater $k$, the more digits are used to represent real numbers with floating-point numbers and the more precise the representation will be. As a matter of course, as a computer's memory is finite, the precision $k$ will always be a finite integer number.

Similarly, as memory is finite, there is a finite set of values the exponent $e$ of the floating-point number representation. The IEEE754 Standard sets precise rules for the cases when the exponent value becomes to small or to large; these rules are explained below. When proving floating-point code, it is common to suppose that the exponent range is unbounded and to provide an additional argument that this assumption is satisfied for the given program, as the exponents actually do stay between the allowable ranges.

Bounding the significand $t$ between $1$ and $\beta$, resp. $0$ and $\beta$ for the decimal format, is a manner of pure convention. It makes the floating-point number representation closest to the so-called scientific notation, which is itself a convention. The IEEE754 Standard also defines floating-point numbers with significands that have these bounds [117]. When handling floating-point numbers in mathematical proofs, it is often easier to manipulate significands that are integer numbers [197]. Converting from one convention to the other is easy, as it suffices to shift the point in $t$ to the right and adjust the exponent $e$ accordingly:

$$(-1)^s \cdot \beta^e \cdot t_0.t_1t_2 \dots t_{k-1} = (-1)^s \cdot \beta^{e-k+1} \cdot t_0t_1t_2 \dots t_{k-1}.$$

Similarly, the fact that IEEE754 decimal floating-point numbers are not always normalized is a particular feature of decimal arithmetic that complexifies proofs but can often be handled in an *ad-hoc* manner.

1. A more detailed description is given just below. The support for a quantum in IEEE754 decimal arithmetic complicates that description.

In this work, we hence define the floating-point numbers ${}^{\beta}\mathbb{F}_k$ of radix $\beta \in \{2, 10\}$ and of precision $k$ with unbounded exponent as the set of numbers

$${}^{\beta}\mathbb{F}_k = \left\{ \beta^E \cdot m,\ E \in \mathbb{Z}, m \in \mathbb{Z}, \beta^{k-1} \leqslant |m| \leqslant \beta^k - 1 \right\} \cup \{0\}\,.$$

In cases when the context leaves no ambiguity about the chosen radix $\beta$, we shall often notate $\mathbb{F}_k$ instead of ${}^{\beta}\mathbb{F}_k$. Similarly, when the precision $k$ is fixed and clear, we shall just notate $\mathbb{F}$.

In certain cases, in particular, when it comes to design an algorithm that is required to work on the full range of possible floating-point inputs or outputs, it is necessary to take the boundedness of the floating-point exponent $E$ into account. This exponent is stored on a certain number of bits; we shall speak of the exponent width $w$. We therefore define the following two sets of floating-point numbers. We shall point out that there is a slight asymmetry between the floating-point numbers in the binary and the decimal radix:

The floating-point numbers in radix $\beta = 2$, precision $k$ and an exponent width $w$ are given by:

$$\begin{array}{rcl} {}^{2}\mathbb{F}_k^w & = & \left\{2^E \cdot m,\ E \in \mathbb{Z}, m \in \mathbb{Z}, 2^{k-1} \leqslant |m| \leqslant 2^k - 1, -2^{w-1} - k + 3 \leqslant E \leqslant 2^{w-1} - k\right\} \\ & \cup & \left\{ 2^{-2^{w-1}-k+3} \cdot m,\ m \in \mathbb{Z}, |m| \leqslant 2^{k-1} - 1 \right\}. \end{array}$$

As shown, the binary floating-point numbers are composed of two subsets; the first subset, where the exponent $E$ varies corresponds to the so-called normal numbers, the second subset, where the exponent is fixed, corresponds to the so-called subnormal numbers [117].

The floating-point numbers in radix $\beta = 10$, precision $k$ and an exponent width $w$ are given by:

$${}^{10}\mathbb{F}_k^w = \left\{ 10^E \cdot m,\ E \in \mathbb{Z}, m \in \mathbb{Z}, |m| \leqslant 10^k - 1, -3 \cdot 2^{w-1} - k + 2 \leqslant E \leqslant 3 \cdot 2^{w-1} - k + 1 \right\}.$$

We shall mention that for the decimal floating-point numbers, there is no distinction between normal and subnormal numbers.

However, the representation of decimal floating-point numbers is not always unique, as their significand $m$ is not necessarily normalized. For example, one tenth may be represented as $10^{-1} \cdot 1$, as $10^{-2} \cdot 10$, or even as $10^{-16} \cdot 1000000000000000$. A decimal floating-point number may have one or more representations. The set of floating-point representations –exponents and significands– that correspond to one decimal floating-point number are called the number's cohort [117].

In addition to floating-point numbers, the IEEE754 Standard foresees additional floating-point data points that can represent non-real, uninitialized or erroneous data or infinities:

— Not-A-Number (NaN) data represent floating-point data that does not correspond to numbers in the set of the extended reals and

— $+\infty$ and $-\infty$ represent floating-point data that correspond to signed infinities [117].

The IEEE754 Standard further distinguishes between zero with a positive sign, i.e. $+0$, and zero with a negative sign, i.e. $-0$. The actual distinction is spelled out for each case in the Standard [117]. The modelization of floating-point numbers does not provide means to express these data and distinctions. Where required, we provide *ad-hoc* proofs that our algorithms correctly work in these cases.

The IEEE754-2008 Standard defines a standard set of floating-point formats, i.e. combinations of a radix $\beta$, a precision $k$ and an exponent width $w$ [117]. We list these standard

| IEEE754-2008 format | IEEE754-1985 name | radix | precision | exponent width |
|---|---|---|---|---|
| | | $\beta$ | $k$ | $w$ |
| binary32 | single precision | 2 | 24 | 8 |
| binary64 | double precision | 2 | 53 | 11 |
| binary128 | quad precision | 2 | 113 | 15 |
| binary16 | - | 2 | 11 | 5 |
| decimal64 | - | 10 | 16 | 8 |
| decimal128 | - | 10 | 34 | 12 |
| decimal32 | - | 10 | 7 | 6 |

Table 2.1 – IEEE754 Standard floating-point formats

formats in Table 2.1, giving the name of the format as defined by the 2008 version of the Standard [117], its name in the 1985 version where appropriate [116], the radix $\beta$, the precision $k$ and the exponent width $w$. We shall mention that the binary32 and binary64 formats are the ones that are the most commonly used.

Up to this point, we have explained what the IEEE754 floating-point numbers are and that we can model them with the set ${}^{\beta}\mathbb{F}_k^w$. Their use for arithmetic is yet to be explained. Before defining operations on floating-point numbers, we have to establish a link between them and the real numbers which they are supposed to represent:

— All floating-point numbers in ${}^{\beta}\mathbb{F}_k^w$ are real numbers. The mapping from the floating-point numbers to the real numbers is hence trivial.

— When a real number $x \in \mathbb{R}$ is a floating-point number in the set ${}^{\beta}\mathbb{F}_k^w$, it can be represented exactly as floating-point number. We say that $x$ is representable in ${}^{\beta}\mathbb{F}_k^w$ [197]. Otherwise, that real number $x$ needs to be mapped to a floating-point number $\widehat{x}$ that approximates it. This mapping can be obtained through the operation called rounding.

The IEEE754 Standard defines several ways a real number $x \in \mathbb{R}$ can be rounded to a floating-point number $\widehat{x} \in {}^{\beta}\mathbb{F}_k^w$: given $x$ it is possible to choose

— the floating-point number $\widehat{x} = \circ(x)$ closest to $x$; in case of tie an additional rule decides how to break the tie,

— the floating-point number $\widehat{x} = \triangledown(x)$ just below $x$,

— the floating-point number $\widehat{x} = \vartriangle(x)$ just above $x$ or

— the floating-point number $\widehat{x} = \bowtie(x)$ whose development into digits has been truncated to fit the precision of the floating-point format.

The rounding $\circ$ is called round-to-nearest, $\triangledown$ is called round-down, $\vartriangle$ is called round-up and $\bowtie$ is called round-toward-zero [117].

As a matter of course, all these roundings $\diamond \in \{\circ, \triangledown, \vartriangle, \bowtie\}$ depend on the floating-point format ${}^{\beta}\mathbb{F}_k^w$ they round the reals to. The most important parameter is precision. This is why we often notate it explicitly as an index of the rounding function: we write $\diamond_k$ instead of $\diamond$. In cases when we manipulate floating-point numbers in two radices at the same time, we shall notate ${}^{\beta}\diamond_k$.

In order to suit our needs for proven floating-point programs, we model these roundings $\diamond_k : \mathbb{R} \to \mathbb{F}_k$ as the functions defined for a non-zero real $x$ by:

$$\begin{aligned}
\circ_k(x) &= \operatorname{ulp}_k(x) \cdot \left\lfloor \frac{x}{\operatorname{ulp}_k(x)} \right\rceil \\
\nabla_k(x) &= \operatorname{ulp}_k(x) \cdot \left\lfloor \frac{x}{\operatorname{ulp}_k(x)} \right\rfloor \\
\Delta_k\,(x) &= \operatorname{ulp}_k(x) \cdot \left\lceil \frac{x}{\operatorname{ulp}_k(x)} \right\rceil \\
\bowtie_k\,(x) &= \operatorname{sgn}(x) \cdot \nabla_k\left(|x|\right),
\end{aligned}$$

where $\operatorname{sgn}$ is the sign function that is $-1$, $0$ or $1$ depending on the sign of its argument and $\operatorname{ulp}_k$ is an auxiliary function we define just below. The function $\lfloor \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ is the function mapping a real number to its closest integer; this function requires a tie-breaking-rule. For any rounding $\diamond_k$, the rounding of zero is zero: $\diamond_k(0) = 0$. We shall remark that the roundings can be defined also in other ways; the definition we use is also the one used by Flocq [22].

The basic idea behind these rounding functions is that when rounding into ${}^{\beta}\mathbb{F}_k^w$, two quantities need to be determined: the integer exponent $E$ that determines the right order of magnitude $\beta^E$ and, in a second step, the integer significand $m$. As the integer significands are separated by units of 1, it is reasonable to express the order of magnitude $\beta^E$ corresponding to a given real $x$ as the so-called unit-in-the-last-place function $\operatorname{ulp}_k$. We define this function for a non-zero real $x$ in radix $\beta$ and precision $k$ as indicated below [195]. Remark that this definition does not account for any limitation of the exponent range in a floating-point system; we need to check the assumption that no underflow occurs separately if we base our proofs on the following definition of the $\operatorname{ulp}_k$ function. We define:

$$\operatorname{ulp}_k(x) = \beta^{\operatorname{EXP}(x)-k+1}.$$

Here, the EXP function yields what we call the natural exponent of a real number [86,195,229]. When rounding to a floating-point format with unbounded exponent range, this function can be easily defined with:

$$\operatorname{EXP}(x) = \left\lfloor \log_\beta |x| \right\rfloor.$$

When rounding to an IEEE754-2008 floating-point format with bounded exponent range is required, this definition can be adjusted to reflect the boundedness of the exponent. The formulas are however rather complicated. We refrain from giving them explicitly for the sake of brevity; the interested reader can consult [117,197] for a more detailed description.

When first published in 1985, the IEEE754 Standard brought a novel concept to floating-point arithmetic. This concept is the cornerstone to the provability of programs running on IEEE754 compliant hardware. As it is basic on a mathematical principle and leaves the (hardware) implementation details to each vendor, it has certainly contributed a lot to the success the IEEE754 Standard has had among hardware vendors. The concept is the concept of *correct rounding* for an operation.

A floating-point realization $F : \mathbb{F} \times \cdots \times \mathbb{F} \to \mathbb{F}$ of a real function $f : \mathbb{R} \times \cdots \times \mathbb{R} \to \mathbb{R}$, mapping a given couple of real arguments to another real number is called correctly rounded iff $F$ yields, for any couple of floating-point arguments in $\mathbb{F}$ the floating-point number in $\mathbb{F}$ that is equal to the one if $f$ were evaluated on these floating-point input arguments exactly with infinite precision and then rounded, with some rounding function $\diamond$ to $\mathbb{F}$. This means that $F$ satisfies

$$\forall x_0, \ldots, x_{n-1} \in \mathbb{F}^n,\; F(x_0, \ldots, x_{n-1}) = \diamond\left(f\left(x_0, \ldots, x_{n-1}\right)\right).$$

The IEEE754 Standard requires the (basic) operations it defines to be correctly rounded [117]. For example the addition operation $\oplus : \mathbb{F} \times \mathbb{F} \rightarrow \mathbb{F}$, which it defines, satisfies: for any floating-point inputs $x, y \in \mathbb{F}$, $x \oplus y = \diamond(x + y)$.

The concept of correct rounding is important for the provability of floating-point software as it allows us to precisely determine, characterize or describe the output of any IEEE754 operation, by decomposing it, formally, into the underlying operation on real numbers and the rounding function. However, this concept of correct rounding is also a challenge to us: even if the Standard requires us to compute in a way as if the underlying real function were evaluated exactly with infinite precision, there is often no infinitely precise way of evaluating it. Infinite precision often means infinite memory, which does not exist.

## 2.2 Novel Operations in IEEE754-2008 and Future Extensions

The concept of correct rounding was first defined in 1985 for the first version of the IEEE754 Standard and the operations it defined. When adding operations to the Standard in 2008, this concept was naturally applied also to these novel operations. We shall describe these additions in the next Section 2.2.1, before highlighting a certain class of novel operations, the so-called Heterogeneous Operations in Section 2.2.2.

### 2.2.1 Overview on IEEE754-2008 Extensions to IEEE754-1985

In this Section, we want to highlight the major differences and enhancements the 2008 version of the IEEE754 Standard brought with respect to the former 1985 version [116, 117]. We want to do this comparison from a three-fold standpoint: first the standpoint of someone who is in charge of providing a complete implementation of the IEEE754-2008 Standard, say through a compliance library such as our libieee754 library we shall describe in Chapter 3, Section 3.2. Second, we want to understand what changes were judged admissible between the 1985 and 2008 version in order to extrapolate what kind of enhancement we might want to propose for later revisions of the Standard. Finally, we also want to describe to a typical user from numerical computing what novelties they can expect from the 2008 version of the Standard.

Designing a compliant implementation of the 1985 version of the IEEE754 Standard [116], be it in hardware or in software, was relatively easy. IEEE754-1985 is concerned solely with binary floating-point arithmetic. It essentially defines two floating-point formats, single and double precision. These two formats were accompanied by a single-extended format, which was essentially never implemented, and a double-extended format which essentially matches the Intel x87 floating-point coprocessor's internal representation [116, 197] but was rarely implemented elsewhere. The standard defined the four rounding modes round-to-nearest-ties-to-even, round-down, round-up and round-towards-zero [116].

IEEE754-1985 defined just a couple of basic operations: addition, subtraction, multiplication, division, division remainder, round to an integer value stored in a floating-point variable, conversions between all formats, conversion to and from machine integers, conversion from and to decimal character sequences and comparisons between floating-point variables [116].

While the Standard required correct rounding for most operations, it did not require correct rounding for conversion from and to character sequences –at least not in all cases [116]. All defined IEEE754-1985 operations immediately depended on a global rounding-direction mode, stored as a global status variable. This included the round-to-integer floating-point operation and the conversions from and to integers [116]. This latter point meant that implementations of languages such as C that require the rounding from a floating-point type to an integer type to be towards zero –a truncation– had to store, change and reestablish the rounding mode for each of these operations [2].

For the 1985 version, all operations were considered homogeneous, i.e. an addition consumed to operands of the same floating-point format and produced a floating-point still in the same format. The IEEE754-1985 even explicitly prohibited operations for which the result would be in a format with less precision that the one of the operands [3] [116]. This means that each of the operations addition, subtraction, multiplication, division and square root require only one implementation per format. This means a compliance library for 1985 just has to support $10$ different functions for these $5$ operations, when supporting the two formats single and double precision.

Certain operations that are commonly required for a complete floating-point environment, such as the copysign, negate, nextafter, scalb or logb operations, as well as the predicates finite, isNaN or unordered, are only recommended by the 1985 version of the IEEE754 Standard [116]. The scalb and logb operations, computing $x \cdot 2^E$ resp. $\lfloor \log_2 x \rfloor$, are not required to work on floating-point numbers that are not normal [116]. This greatly simplifies their implementation but also limits their usage in numerical applications.

For certain aspects, IEEE754-1985 leaves large room for choices made by the implementer of the Standard: for example, when a computation leads to a value smaller in magnitude than the smallest normal floating-point number and the rounded result is unequal to the mathematically exact one, the IEEE754 Standard requires status flags indicating these conditions to be set [4] [116, 117]. Detection of these two conditions can be done, according to IEEE754-1985, in four different ways: underflow can be determined before or after rounding and inexactness can be associated to proper inexactness or to denormalization loss [116]. Similarly, the IEEE754-1985 Standard distinguishes between two different kinds of Not-A-Number (NaN) data: signaling NaNs or quiet NaNs. While it requires signaling NaNs to provoke floating-point exceptions and prohibits quiet NaNs from provoking exceptions [5], the Standard does not describe how these two kinds of NaNs can be distinguished by their memory encoding. This leaves freedom to the implementer of the Standard but is also source of incompatibilities and portability issues the user can be confronted with.

To sum up, for a compliance library designer, the IEEE754-1985 Standard [116] mandates about $70$ functions –operations– to be implemented, when supporting both floating-point formats defined by the Standard. This number includes function for the operations that are just recommended by the Standard, such as copysign or logb.

Implementing the 2008 version of IEEE754, e.g. with a compliance library, is another

2. `https://software.intel.com/en-us/articles/fast-floating-point-to-integer-conversions`
3. The interested reader may consult footnote 4 of the 1985 version of the IEEE754 Standard
4. under default exception handling, see Section 2.3 for details
5. in most circumstances; exceptions to this rule exist [116]

matter. IEEE754-2008 adds a vast number of novelties to the Standard. The first, major difference is that IEEE754-2008 floating-point arithmetic is no longer binary only: IEEE754-2008 defines two radices [117], $\beta = 2$ and $\beta = 10$.

The list of the floating-point formats defined by the Standard is extended: in addition to the binary32, i.e. single precision, and binary64, i.e. double precision, formats, IEEE754-2008 defines [6] the binary16 and binary128 formats for the binary radix and three completely new formats for the decimal radix, viz. decimal32, decimal64 and decimal128 [117].

The 2008 version of the Standard adds a new rounding mode, called roundTiesToAway [117]. In this mode, rounding is performed to the nearest floating-point number, but in contrast to the existing mode that rounds to the nearest floating-point number, roundTiesToEven, ties, i.e. midpoint cases, are broken differently [117].

IEEE754-2008 still supports the basic operations addition, subtraction, multiplication, division and square root but adds a sixth operation, called Fused-Multiply-And-Add (FMA), that computes $a \times b + c$ with one final, correct rounding [117].

A major difference between IEEE754-1985 and IEEE754-2008 with respect to these basic operations –addition through FMA– is that in contrast to the former Standard, which explicitly forbade this feature, IEEE754-2008 requires implementations to support heterogeneous versions of these operations [117]. This means that one operation is supported for all possible combinations of supported formats [7] for the operands and the result. For example, when the two formats binary32 and binary64 are implemented by an environment, the addition operation comes in eight variants, four for the combinations on the operands, times two for the result formats. As it has three operands, the FMA operation already requires $16$ variants for two formats. When three formats like binary32, binary64 and binary128 are implemented, addition already comes in $27$ variants, FMA in $81$.

From a compliance library designer's perspective, supporting all these variants of the heterogeneous operations is not trivial. In particular, when the result's precision is less than the operands' precisions, guaranteeing the correct rounding property, required by IEEE754-2008, is a challenge [117], as traditional hardware will just support homogeneous operations. We devote the next Section 2.2.2 to a detailed analysis of this challenge.

Concerning the other operations that were mandatory already in IEEE754-1985, viz. rounding to an integer floating-point number, conversion from and to machine integers and conversions from and to character sequences, the 2008 version of the IEEE754 Standard is also much broader [117]: the roundings to integer floating-point numbers as well as the conversions from and to machine integers come in various variants, differing in the way their round their result, either depending on the global rounding mode or a rounding direction indicated in their name, and in subtle differences concerning on whether or not they indicate if their result is different from the input by setting an inexact flag. The conversions from and to character sequences support a new radix for the character sequences: the hexadecimal radix. Decimal character sequence input and output is of course still supported, too. The IEEE754-2008 Standard requires correct rounding for character sequence conversion for both input and output in a wide exponent range and recommends these operations to be correctly rounded for all cases [117].

As IEEE754-2008 supports floating-point numbers in two radices, it of course supports

6. The binary16 and decimal32 are so-called interchange formats, intended to store floating-point data, not to compute with them [117].

7. of one radix; multi-radix combinations are not (yet) supported

conversion operations between the formats of both radices. These conversions must be correctly rounded, according to the current rounding mode of the output format's radix [117].

The operations recommended by IEEE754-1985, such as logb, scalb and copysign, become mandated operations in IEEE754-2008 [117]. In the 2008 version, there are no longer any restrictions, such as normal numbers only, for logb and scalb; both work for subnormal numbers and with correct rounding. IEEE754-2008 extends the list of supported predicates and classification operations, which serve the purpose of detecting the nature of a floating-point datum: normal or subnormal number, infinity, Not-A-Number etc. IEEE754-2008 also provides a total order predicate that allows floating-point data to be sorted in an unambiguous manner, even if it includes Not-A-Number data [117].

The IEEE754-2008 itself contains a Clause that defines operations that are just recommended by the Standard, but not mandatory for a compliant implementation [117]. The list of these recommended operations is quite comprehensive: it ranges from so-called reduction operations that compute approximations to sums, dot products and products of vectors of floating-point numbers to correctly rounded mathematical functions, such as exponential, logarithm or trigonometric functions [117]. The implementation of these functions is only recommended by the Standard, not mandatory. When implemented, these functions however must be correctly rounded, too, even though correct rounding of these functions is difficult due to the so-called Table Maker's Dilemma [165, 197].

The 2008 version of the IEEE754 Standard removes certain subtleties from the 1985 version [117]. These subtleties may result in different behavior across platforms implementing the Standard. First of all, underflow and tininess detection could be performed in four different ways in the 1985 version. In the 2008 version, only two different ways are described and permitted [117]. The subsisting two ways do not affect the floating-point results delivered by the environments; they only affect the setting of diagnostic flags when the rounding goes upwards to the least normal floating-point number. Second, IEEE754-2008 clearly determines which bit in the memory encoding of binary IEEE754 floating-point numbers determines whether a Not-A-Number datum is a so-called signaling or quiet Not-A-Number [117].

The IEEE754-2008 however adds a novel cause of portability issues: IEEE754-2008 floating-point numbers in the decimal radix can be encoded in two different encodings, the so-called Densely-Packed-Decimal and the Binary Encoding [117]. The Standard however assumes this double choice and mandates that conversion function be provided by any environment implementing decimal floating-point numbers [117].

To sum up, the 2008 version of the IEEE754 Standard is way more comprehensive than the 1985 version [117]. It provides certain operations in dozens of variants. From a compliance library writer's perspective, these variants add complexity. For a user, they simplify certain tasks however and are hence useful. The novel definition of a second radix supported by the Standard, the decimal radix, effectively doubles the number of floating-point operations required for an implementation. A major increase in operation count stems however from the fact that the basic computational operations, such as addition or Fused-Multiply-And-Add in IEEE754-2008 heterogeneous and must be provided with one single, correct rounding for all combinations of operand and result format combinations [117]. We will discuss this requirement in the next Section 2.2.2. It results in the following numbers: were about 70 operations required for support of the mandatory parts of IEEE754-1985 in the binary32 and binary64 formats, more than 350 operations are required for these formats in IEEE754-2008. When decimal floating-point arithmetic is added with the two formats decimal64 and decimal128, the operation count essentially doubles to about 720 operations. The additional support of the binary128 format increases the operation count again to about the double.

### 2.2.2 Heterogeneous Operations in IEEE754-2008

As we have already stated in Section 2.2.1 above, the IEEE754-2008 Standard defines so-called heterogeneous operations for which the input and output formats differ in precision and exponent width. For example, an addition operation taking two IEEE754 binary64 operands and returning an IEEE754 binary32 result is heterogeneous [117].

While this enhancement of the IEEE754 Standard still increases the closedness of the floating-point environment, as floating-point numbers of all formats can be combined for all operations, it is not completely innocuous: at least in certain rounding modes, typically in the default round-to-nearest mode, the heterogeneous operations cannot be implemented just using existing operations, such as homogeneous operations, followed by format narrowings. Indeed, a format narrowing induces a second rounding and the subsequent rounding may yield to a result different from the one obtained when only performing the IEEE754-mandated single rounding.

For example, take the two IEEE754 binary64 floating-point numbers $a = 2^{23} + 17$ and $b = 2^{-1} - 2^{-31}$. Suppose that a heterogeneous addition with an output as an IEEE754 binary32 is to be performed. The infinite precision addition result clearly is $c = a + b = 2^{23} + 17 + 2^{-1} - 2^{-31}$. The binary32 format having 24 bits of precision this value $c$ is just below the midpoint $2^{23} + 17 + 2^{-1}$ of the two binary32 numbers surrounding $c$, i.e. $2^{23} + 17$ and $2^{23} + 18$. As the value is below the midpoint, the correct heterogeneous rounding to nearest hence is $\widehat{c} = 2^{23} + 17$. Suppose now that a homogeneous addition with rounding-to-nearest to IEEE754 binary64 is performed first. As the binary64 format has 53 bits of precision, the two binary64 numbers surrounding $c = 2^{23} + 17 + 2^{-1} - 2^{-31}$ are $c_D = 2^{23} + 17 + 2^{-1} - 2^{-29}$ and $c_U = 2^{23} + 17 + 2^{-1}$. Their midpoint is $c_M = 2^{23} + 17 + 2^{-1} - 2^{-30}$. The homogeneous addition hence performs a rounding-to-nearest that goes upwards, yielding $c' = c_U = 2^{23} + 17 + 2^{-1}$. This intermediate binary64 result now gets rounded again, to the binary32 format. As $c'$ exactly lies on the midpoint of the two binary32 numbers surrounding it, in round-to-nearest an IEEE754 tie-breaking rule is applied: in default binary IEEE754 floating-point arithmetic [117], the tie is broken by returning the number whose significand ends with an even (zero) bit. For instance, $\widehat{c'} = 2^{23} + 18$ is returned. Clearly, the correct rounding $\widehat{c}$ obtained by rounding once and the one obtained with a double rounding, $\widehat{c'}$, are different.

This double rounding issue is different from double rounding phenomena already described in the literature [81]. For example, it is possible to implement the IEEE754 binary32 homogeneous operations using solely binary64 homogeneous operations and conversions back and forth between binary32 and binary64. The IEEE754 format binary64 provides enough precision so that a double rounding first to binary64 and then to binary32 is innocuous if the binary64 inputs of a binary64 homogeneous operation are actually representable in binary32 [81]. In our case, this last assumption is not satisfied: in a binary64 to binary32 heterogeneous operation, the binary64 inputs do not generally hold on a binary32 number.

It is clear that the issue with the double rounding stems from the rounding in the narrowing operation that induces the second rounding. Format widenings however, for example from the IEEE754 binary32 format to the binary64 format, do not provoke any rounding, as all floating-point numbers of the less precise format are exactly representable in the preciser format. This observation allows for an immense reduction in the number of heterogeneous operations that need to be supported: all heterogeneous operations with operands of different formats can be subsumed into heterogeneous operations where all operands have the same format, as widenings can be used to exactly convert operands of less precise formats to the format that is most precise among all operands. Similarly, heterogeneous operations where the output format's precision is higher than the precision of the operands can be implemented easily with widenings. Some care needs to be taken in order to correctly handle IEEE754 Not-A-Number (NaN) data, which comes with a payload [117]. While the IEEE754 Standard does not precisely define the meaning of a NaN's payload [117], it clearly specifies that an operation must produce the same NaN in output when one of its operands is a NaN, including the payload. For the heterogeneous operations relying on intermediate widenings, this may induce some issues with the NaN payloads when a NaN is first widened to fit the other operands' format and then heterogeneously "rounded" back to its original format, as it traverses a heterogeneous operation. As an example, consider a binary32 NaN with a specific payload on the binary32 operand of an IEEE754 binary32 plus binary64 to binary32 heterogeneous addition that gets implemented with a widening of the first operand to binary64, followed by a heterogeneous binary64 plus binary64 to binary32 addition. In the end of the day, that operation is required to return the original NaN, with its original payload [117].

The introduction of the heterogeneous operations is also a challenge for floating-point environment realization in languages, such as C, and for testing. While the rules for recursive expression evaluation, with infix operations, are settled for most languages, such as C [122], they do no longer fit for the heterogeneous operations. Consider for instance the following C code:

```
double a = ...;
double b = ...;
float  c;

c = a + b;
```

By the classical evaluation rules [122], the compiler is to translate this sequence that does perform a double rounding, by first rounding the sum $a + b$ to IEEE754 binary64 (double precision), then to IEEE754 binary32 (single precision, called `float` in C). By the introduction of heterogeneous operations in IEEE754-2008, languages are challenged with the question whether a change in semantics should occur. Would it be reasonable to translate this sequence into an heterogeneous IEEE754 binary64 to IEEE754 binary32 addition? If yes, would this new evaluation rules endanger existing programs? What about more complex expressions, with intermediate results and, possibly, explicit casts? These questions are not easy to answer. Until a satisfying answer is found, it seems reasonable to express the IEEE754-2008 heterogeneous operations as functions whose input and output types are explicitly given in the function name and that need to be called explicitly where needed. The functions can be provided as inlinable functions or compiler builtins, so they will not suffer any performance overhead due to a real function call. As a matter of course, this explicit fashion of expression heterogeneous operation puts a burden on the user, as they must spell out expression trees with infix operations manually as prefix trees, calling these heterogeneous functions.

However, whatever the solution defined by languages be, as they are among the mandatory operations required by the IEEE754-2008 Standard, solutions to implement the heterogeneous operations are needed. Processor and system vendors, as well as all implementers of the floating-point environment offered by operation systems, have several options:

— System implementers may consider that heterogeneous operations will be used rarely and hence do not affect overall application performance. They may hence choose to implement these operations fully using only plain integer arithmetic, simulating the different steps usually performed by floating-point hardware [183].

— Hardware vendors may also decide to add hardware support for heterogeneous operations. Such hardware support can be realized in different manners: first of all, it is possible to add hardware units that produce their output on formats of less precision than the precision of the operands. Existing designs for adders, multipliers etc. may be modified and even be reused for both tasks. As that datapath width of the operands and the output are not the same, hardware designers may however encounter problems with access to the memory hierarchy in terms of existing datapath widths [237]. Additionally, just adding hardware units or extend existing units for uncommon operations is not the most energy-efficient option.

  Second, hardware designers can adopt approaches for which the existing hardware units need to be modified only minimally: the Itanium Processor family, for example, has supported heterogeneous operations for at least a decade [52, 177]. The floating-point instructions of this processor family take operands and produce results in an extended, $82$ bit wide format [52, 177], with significands of $64$ bit length [52, 177]. However, each instruction is also labeled with a rounding mode and a rounding precision: even though it returns a wide significand all the time, the instruction performs rounding to a lower precision, regardless of the precision of the operands [52]. Implementation of heterogeneous operations is easy with such kind of hardware: it suffices to choose a lower precision for the output. Even though the instructions are well-defined, even in irregular cases. For example, when rounding to a format with lower precision and lower exponent width requires to perform subnormal rounding, the output however is to be delivered in the extended $82$ bit format, the instruction is specified to first perform denormalization for subnormal rounding and then renormalization for the wider, extended format [52]. These cases are not all realized in hardware, though: hardware just handles the most common cases and traps to request software support for corner cases [52, 177]. Additional cost for hardware is hence reduced.

  Third, to provide support for the IEEE754-2008 heterogeneous operations, hardware can include mechanisms to correct the possibly wrong result that may be obtained through a double rounding, when first a rounding to the operands' precision is performed and then a second rounding to the heterogeneous operation's output precision.

  In the literature, two ways to perform such corrections can be found: first a technique, which to our knowledge has been patented but not implemented yet, that saves off three decisive bits of the the infinite precision significand at the first rounding (the round, guard and sticky bits [77, 197]) in order to enable a subsequent, second rounding to be corrected [54]. The description of the approach does not explain clearly where those three correction bits are stored: in a global status register, which would hinder SIMD implementations of the technique, or in special heterogeneous-rounding registers, which might be cumbersome to add to existing processor designs.

  The second approach found in the literature is more precisely specified and has already been realized in certain processors [2]. It is based on the observation that no double rounding issue can be observed in any eventual rounding mode to the lower precision format when the first rounding is done in a non-classical rounding mode, called round-to-reround by certain hardware vendors [2], and also known as round-to-odd in the preexisting literature [21]. The idea is simple: when the difference in precision between the format the operation's result is first rounded to and the final format of the heterogeneous operation is at least 3 bits, all bits required to correctly perform the second rounding can be stored in the intermediate, larger precision format: the round bit, the guard bit and the sticky bit [77, 197]. The round and guard bits are anyway present in the larger precision format. The sticky bit is however the logical or of all other trailing bits in the infinite precision significand [77]. This bit must be computed and materialized as the last significand bit of the intermediate rounding: hence round-to-odd sets the least significant bit to one whenever the rounding is inexact and to whatever value is required when the rounding is exact [21]. The implementation in hardware of this technique is pretty easy and does not involve a large hardware cost in terms of surface. However, the existing instruction sets need to be modified to include ways to set the rounding mode of the first, intermediate rounding to this special round-to-odd mode. Further, compilers and the whole software ecosystem need to support this not very common and hence not very portable rounding mode.

— In order to provide the heterogeneous operations, implementers of an IEEE754-2008 floating-point environment may also choose to find a middle ground between full emulation of the operations using integer instructions and modification of their hardware. The idea is to use standard homogeneous IEEE754-1985 operations together with the double rounding technique but to detect and correct cases when the double rounding does not provide the correct heterogeneous rounding. This approach is most attractive for implementers who need to strive for best portability while still providing best performance. Ideally, all operations required to detect and correct the rounding can be performed in the floating-point unit of a processor, hence avoiding unnecessary data movements between the integer and floating-point datapaths.

A first proposal to implement was made in the literature in [183]. The technique is not fully independent of the integer unit and does require change of the current rounding mode, which may make it sensible to asynchronous exception handling: its basic idea is to identify double rounding issues by checking for certain trailing bit patterns in the higher precision intermediate rounding. This detection is done using integer instructions. When a possibility of a wrong heterogeneous rounding is identified, the first, higher precision operation is performed again, in round-toward-zero mode. When this operation is then found to be inexact, by observation of the so-called inexact flag [8], the significand produced by the higher precision operation is modified using integer operations to force its last bit to one. Essentially, rounding-to-odd [21] is simulated in cases when only this rounding guarantees the second, lower precision rounding to be the correct heterogeneous rounding.

This approach has several shortcomings: it is not fully independent of integer instructions, it requires global state (in the form of the IEEE754 rounding mode) to be changed and it is not easy to make functional in all extreme cases, such as gradual underflow to a subnormal number. For these reasons, we also worked at solving the problem of heterogeneous roundings. Our solution reuses and extends a floating-point operation sequence traditionally used in implementations of elementary functions, the so-called Ziv rounding test, to detect and correct double roundings for the implementation of the heterogeneous operations. We will present this work in Section 3.2.2.

8. see Section 2.3 for a definition of the IEEE754 exceptions and status flags

## 2.3 A Never Ending Story: Non-Standard Exception Handling

As described in Section 3.1, the IEEE754 Standard defines floating-point operations in terms of correct rounding of their mathematical counterpart. This means that a floating-point division operation will behave as if it computed the exact, infinite precision division result and rounded it to the output floating-point format [117].

This correct rounding rule unambiguously determines the floating-point result of operations for arguments on the definition domain of the underlying mathematical functions. However, additional rules are required in the case when an operation is to be evaluated outside its definition domain [117]. For example, a division of zero by zero is mathematically undefined.

Similarly, rounding may provoke exceptional cases: if rounding of a real number makes the exponent grow to large or to small, as it is bounded, the exponent cannot be stored in the IEEE754 format the rounding targets [117]. In this case, an alternative rounded floating-point result needs to be produced, as for example infinity for exponent overflow or zero for complete underflow.

The IEEE754 Standard handles all these exceptional cases through a mechanism [117] that proceeds in two steps: when an operation is performed on floating-point inputs for which exceptional case handling is required –these cases are precisely determined by the Standard– the Standard requires these operations to first *signal* that exception [117]. The signaling of an exception then provokes, in a second step, a certain handling. The default handling consists in replacing the floating-point result by some default floating-point data, in lieu of an answer. This default answer is precisely defined by the IEEE754 Standard and may be zero for underflow cases, infinity for overflow cases or division by zero or Not-A-Number (NaN) for invalid cases such as square root of a negative number [117].

The IEEE754 Standard defines five exceptions that are to be signaled by floating-point operations, when the corresponding condition arises:

— The inexact exception is signaled whenever a floating-point operation produces a result that differs from the mathematically exact one as a consequence of rounding. The inexact exception is sometimes also called the precision exception.

— The underflow exception is signaled whenever a floating-point operation produces a result that is smaller in magnitude than the in magnitude least normal floating-point number of the corresponding format.

— The overflow exception is signaled when a floating-point's operation is larger in magnitude than the largest floating-point number of the corresponding format.

— The divide-by-zero exception is signaled for a division of a non-zero floating-point number by a zero floating-point number. This exception also gets signaled for certain functions, such as logarithm, when they are evaluated on values for which a "division" is morally performed.

— The invalid exception is signaled for all operations when they are executed on arguments for which no meaning can be defined on the extended reals. For example, zero divided by zero provokes signaling this exception. The exception is also executed for certain comparisons, when Not-A-Number data is involved and for Not-A-Number data of a certain kind, called signaling NaNs [117].

When the signaling of an exception is handled by the IEEE754 default exception handler, a floating-point value to be returned in lieu of an actual result is produced. In order to allow the occurring exception to be recorded, the IEEE754 Standard defines five status flags, one for each of the five possible exceptions, that get set [9] during the default exception handling [117]. The flags are sticky, i.e. if they are raised, they stay up until until they are explicitly cleared.

This default exception handling behavior has one important advantage: under default exception handling IEEE754 floating-point arithmetic is closed, i.e. for each operation, a floating-point result, be it just a Not-A-Number, is produced and these default data can also be consumed by subsequent operations. This is desirable for large scale numerical codes, where it is often preferable to handle invalid operations in an *a posteriori* manner instead of interrupting or even stopping the program for each exceptional case [197]. As the floating-point flags are provided by the IEEE754 environment, *a posteriori* handling has even access to some diagnostic information [117].

In certain cases, the default exception handling behavior of the IEEE754 Standard may however yield to misleading results or it might be too cumbersome to clear the IEEE754 flags for some portion of code, run the code, check the flags and perform *a posteriori* handling based on these flags. For example, under default exception handling and round-down as the rounding mode, the expression $\frac{1}{\sqrt{x-y}}$ evaluates to $-\infty$ for $x = y$ [117]. It will hence be negative for just this case, whereas it stays positive everywhere else.

Even if the default floating-point values to be returned by default exception handling have been chosen with care by the designers of the IEEE754 Standard, default exception handling cannot foresee any combination of floating-point states, intended mathematical behavior and corresponding programs.

The IEEE754 Standard therefore defines alternate ways to handle the signaling of exceptions, instead of default exception handling. In its 1985 version, the Standard describes this alternate way in terms of so-called traps [116]. These traps correspond to interrupt vectors into which addresses of program subroutines to handle the corresponding exception is to be loaded. In order to have the processor branch to these subroutines for a certain exception, the exception must be unmasked, by unsetting a certain bit in a so-called exception mask register [116]. When describing this mechanism, the Standard does not address any issues arising to multi-process environments, privilege separation between kernel and user code or issues due to exception occurring in parallel, e.g. when executing SIMD instructions.

When the IEEE754 Standard was revised in 2008, it was felt that this 1985 description of alternate exception handling was too hardware-centric and suffered from certain issue, such as the ones cited above. The 2008 version of the IEEE754 Standard therefore describes alternate exception handling in terms of an informal API which is supposed to allow the user to register subroutines as handlers for IEEE754 floating-point exceptions [117]. The Standard leaves considerations of the exact semantics of such an API to the “languages”, i.e. to the implementers of the Standard in certain programming environments and languages [117].

In both the 1985 and the 2008 version, the subroutine that can be registered for a certain IEEE754 exception is supposed to receive the arguments of the floating-point operation that signaled the exception as well as an indication which type of operation (addition, subtraction, multiplication etc.) was executed. The subroutine can then raise the IEEE754 flags it judges necessary to raise and it can provide a floating-point datum that will be used as a result of the exceptional operation [116, 117].

The 2019 revision of the IEEE754 Standard is not supposed to change anything to the

9. Some exceptions do not provoke setting of a flag in certain cases. The interested reader may consult Clause 7.5 of the IEEE754 Standard [117]

ways IEEE754-2008 implements alternate exception handling. Since 1985, hence for about 33 years, the IEEE754 Standard has contained this mechanism for alternate exception handling, which is valuable in certain circumstances, such as expression evaluation or debugging. When designing hardware, processor vendors spent considerable resources on conceiving the hardware behind floating-point exception traps, as well as on validating these hardware implementations. The logic required to implement exception traps in hardware costs area on chips and power to drive it.

However, for now 33 years, software support for IEEE754 alternate exception handling has been essentially nonexistent. There are certain highly non-portable sequences, partly in assembly, that activate IEEE754 traps on certain systems and try to register trap handling subroutines [10]. The GNU C library provides functions to activate generation of a Unix/Posix signal when an IEEE754 exception is signaled. That Unix signal can be acted upon, however, there is no support to return alternate floating-point results or to easily inspect which operations provoked the signal [118].

We find this situation unsatisfactory. If nobody needs support for IEEE754 alternate exception handling –and we doubt that actually nobody does [132]– support for alternate exception handling should be removed from IEEE754 in future revisions. This would allow hardware vendors to stop supporting these exceptions in hardware, which might save energy. The details of these savings still need to be investigated; see Chapter 6, Section 6.2.1 for our proposal for future research on this subject.

However, if there are actual use cases for alternate exception handling –and we find that the debugging argument is extremely compelling– portable user libraries, built on appropriate kernel support, should be provided on all major systems.

In 2014, we proposed a cooperative research project at the LIP6 laboratory level. The project was accepted and got financed. Together with researchers in a team working on Operating Systems, we instructed a Master student to investigate the feasibility of such a library for IEEE754 alternate exception handling. Unfortunately, the project did not lead to any results, mainly due to health problems of the student working on the subject. We keep on working on this matter.

## 2.4 Other Floating-Point Arithmetic besides IEEE754

IEEE754 is sometimes considered to be the cornerstone of floating-point arithmetic. This is certainly true up to a certain extent. Other means to perform floating-point arithmetic however do exist. We shall address them quickly in this Section.

The floating-point environment defined by the IEEE754 Standard [117] and implemented by most hardware vendors [24] provides satisfactory support for a wide range of applications, ranging from Computer Games [258] over smaller numerical programs [197] to super-computing applications [106].

However some applications currently need support for floating-point arithmetic that goes beyond IEEE754, mainly in terms of available precision [11,79,152,153,259]. Hardware support for IEEE754-2008 arithmetic typically ends with the binary64 (double precision) format, which provides 53 bits in its significand [117]. The IEEE754-2008 binary128 format, which would

10. `http://www.msg.ucsf.edu/local/programs/IBM_Compilers/Fortran/html/pgs/ug39.htm`

provide 113 bits, is mostly implemented in software or software-defined hardware units [176]. No IEEE754 format with more bits in its significand is available on current systems.

As a matter of course, all numerical software that is used to design programs for the IEEE754 world, such as mathematical libraries, needs to be run at a precision larger than the ones targeted in the IEEE754 world. For example, it seems hard to imagine how one could compute a polynomial that would approximate a mathematical function, like exponential, with an accuracy at or below IEEE754 double precision, using an algorithm, such as the Remez algorithm, running itself at the same or even lower precision [38].

In order to answer this need for precision exceeding the one provided by the IEEE754 formats [117], several approaches have been proposed, some of which have got into common usage. First, we may mention the so-called compensation techniques: these approaches run numerical code using the highest available IEEE754 precision and compensate the round-off error of each operation by maintaining a correction term for each quantity occurring in the computation [182]. The error correction term can be computed for the round-off of the basic operations addition, subtraction, multiplication, division and square root using so-called Errorfree Transformations [97,138,247,248]. A global compensation term is deduced of the elementary error term, by integrating it into the global term. This integration step can be performed after each operation, in which case compensation techniques become classical doubled-precision arithmetic [110,138,236,247,248] or only after a certain number of computation steps [97]. Compensation can be done once, or, if precision is still not sufficient, several times, i.e. by compensating the error correction term by one more term. This leads to tripled or quadrupled precision [110,131,164,165,236].

While these compensation techniques, based on error-free transformations, are able to extend the precision range of IEEE754, they fail to extend the dynamic of the floating-point environment, i.e. the available exponent range. Typically, the error-free transformations for addition work as long as no spurious overflow occurs [18,138]. They also work in the underflow domain, as floating-point addition becomes naturally exact there [197]. Error-free transformations for multiplication however suffer severely from underflow or overflow. They are clearly not "errorfree" in general if underflow occurs. These limitations make the use of errorfree transformations sometimes cumbersome, in particular when the whole exponent range must be covered for input and output. In Section 3.6, we shall use errorfree transformations ourselves but as we shall see, most of the difficulty of our algorithm will stem from properly avoiding spurious underflow and overflow.

A second approach to increasing precision is using so-called multiple precision floating-point libraries. The most prominent one among such libraries is the GNU MPFR library [11] [86]. These multiple precision libraries emulate high precision floating-point arithmetic by representing floating-point numbers as data structures containing an integer exponent field and a long, high-precision integer significand. This significand is itself a long integer represented as an array of machine integers that form the digits of the long integer in a high radix, corresponding to the machine word length, such as $2^{32}$. Asymptotically optimal long integer algorithms are used on these significands to implement the basic operations: Karatsuba and Toom-Cook multiplication, FFT-based multiplication etc. [86]

In this work, we try to propose yet another approach to increasing the precision, besides compensation techniques based on IEEE754, which have the disadvantage of not widening the exponent range, and asymptotically optimal multi-precision. In Section 3.5, we shall

11. `http://www.mpfr.org/`

present `libwidefloat`, which is supposed to address a slightly larger precision range than what IEEE754 offers, typically up to $512$ bits, as well as an extended exponent range. In contrast to MPFR, our `libwidefloat`, as we shall see, does not offer dynamically adaptable precision, which has the advantage of leveraging compiler optimization for static memory access.

In terms of internal representation of floating-point numbers, these multiple-precision libraries are incompatible with the IEEE754 Standard. As a matter of fact, the IEEE754 does not define arbitrary, wide-precision floating-point numbers. The multiple-precision libraries however often follow the IEEE754 Standard in terms of special values, such as signed infinities, signed zeros, Not-A-Number data, as well as in terms of rounding-modes. Even floating-point flags, raised as a result of default handling of exceptions, are supported [86]. The MPFR library also honors the correct rounding principle: all operations behave as if the mathematically exact result were computed first and then rounded, even if the operation involves computing a transcendental function [86].

However, the multiple-precision libraries, in particular MPFR, allow for a richer floating-point arithmetic than plain IEEE754. All MPFR floating-point variables have dynamically adjustable precision, both at variable initialization time and at any later step in an algorithm. Variables with different precisions can freely be mixed when used as operands or results of operations [86]. It becomes hence possible to write algorithms that adapt their precision in order to always guarantee a certain output accuracy. In other words, multiple-precision enables floating-point arithmetic with *a priori* error bounds. We shall heavily use this property for our work presented in Chapter 5 of this thesis.

This additional freedom provided by dynamic multiple-precision comes at a price, though. While coding with MPFR is almost like coding in assembly –a function must be called per elementary floating-point operation– analyzing the global error of a multiple-precision algorithm, in terms of the dynamically adjusted precision, is challenging.

As execution performance is anyway affected by the fact that multiple-precision is essentially realized in software, it is reasonable for most applications to switch from plain multiple-precision floating-point arithmetic to multiple-precision interval arithmetic. In interval arithmetic, quantities, as they are stored in variables, are represented not just by single approximated floating-point values but with intervals in which that mathematically exact quantity is guaranteed to lie [5]. Basic interval arithmetic can be implemented by representing both endpoints of an interval by two floating-point numbers; these floating-point numbers can be multiple-precision floating-point variables. All operations are then performed with rounding modes chosen in such a manner that roundoff error might enlarge the width of the intervals but never induce any wrong result with respect to the guarantee that the mathematically exact value stays contained in the respective interval [5, 217].

The MPFI library [12] [217] implements such a multiple-precision interval library, where all intervals are represented with multiple-precision floating-point numbers as endpoints. MPFI allows the dynamic roundoff-error issue to be solved by having the computer performing the error analysis dynamically as the algorithm runs. When the final interval result is too inaccurate, the algorithm can be re-run with increased precision in an attempt to increase the output accuracy [217].

However, coding with MPFI still is a tedious process: again, a function call must be made per operation. This means that expression trees, such as $a + \sin b + 17 \times e^c$, must be

12. `https://gforge.inria.fr/projects/mpfi/`

traversed manually and operations coded one by one. In particular for use-cases where numerically programs are short-lived as they are written just to design or analyze other floating-point programs, this is too tedious. An enhanced, rigorous, dynamic and multiple-precision numerical tool, with automatic precision adaptation and an easy-to-use interface, is required.

Sollya[13], which we develop, is intended to be such a tool [41]. Sollya provides the user with an interface that allows for definition of composed functions as expression trees which can be evaluated to variable output accuracy. The tool automatically adapts compute precision, using a mix of MPFR and MPFI to realize this goal. Sollya is rigorous through interval arithmetic and warns the user in cases when functional expressions cannot be evaluated accurately at particular points, essentially where they are not well defined. Sollya is tightly coupled to IEEE754 floating-point arithmetic as it allows roundings to IEEE754 formats, such as single or double precision, to be simulated where required. Sollya comes with a small programming language but can also be exploited through a C language interface, for which bindings to higher level languages such as Python exist[14]. As we consider Sollya to be just a tool that we give ourselves –and the community– for our research but not necessarily the focus of our research, and, in particular, as we see in Sollya a mere translation, into code, of some of our algorithms, we refrain from discussing Sollya in detail in this work.

13. `http://www.sollya.org/`
14. `https://gitlab.com/metalibm-dev/pythonsollya/`

CHAPTER 3

# Software to Support an Enhanced Floating-Point Environment

*J. Palmer had hoped that the [x87] coprocessor would have everything on one chip. It would have [...] the transcendental function library, the math library [...] so, when you plugged it in, everybody would get the same [...] results as everybody else, regardless of the programming language they fancied.*

William Kahan, *interviewed by Th. Haigh*

## 3.1 Introduction

The IEEE754-2008 Standard states it pretty clearly: *An implementation of a floating-point system [...] may be realized entirely in software, entirely in hardware, or in any combination of software and hardware.* [117]. It is the software aspect behind floating-point arithmetic that often gets forgotten or unduly neglected. This is what we also want to present to the reader in this Chapter: what is needed to support at least the mandatory part of IEEE754-2008 from a software perspective, even if we intend to base this software on existing floating-point hardware? Are there means to extend the scope of the IEEE754-2008 Standard, to make it more complete, even if some of these novel operations end up being useful but too anecdotal to actually invest into adding hardware? The IEEE754-2008 Standard is written in terms of “operations”. Where does an operation stop being an operation and becomes an algorithm? When is the opposite true: can’t we propose basic algorithms such as some basic linear algebra operations that are so common that subsequent revisions of IEEE754-2008 could include them as novel “operations”?

This Chapter is organized as follows: in Section 3.2, largely extending our publications in [66, 161], we describe our work on the software side of the existing IEEE754-2008 Standard. In Sections 3.3 and 3.4, we try to make our case for extending the Standard in a way that would allow binary and decimal floating-point arithmetic to be freely mixed. These Sections are based on [28,125]. In Section 3.5, based on [156], we then extend the scope of floating-point arithmetic with work on extended precision arithmetic. In Section 3.6, we report our findings on higher-level operations, as we proposed them in [101]. We will give a conclusion for this Chapter in Section 3.7.

## 3.2 Supporting the Mandatory Part of IEEE754-2008

Before we set out to proposing extensions to future editions of the IEEE754 Standard, we will look on what we need and are able to do to support the current revision, IEEE754-2008 [117]. In Section 3.2.1, we will shortly describe the `libieee754` support library we have started to develop; this Section is based on [161]. In Section 3.2.2, based on [66], we will look into a way to "recycle" Ziv's rounding test, which was developed in a different context, for the implementation of the heterogeneous operations in IEEE754-2008. Finally, in Section 3.2.3, we will briefly discuss ways to implement IEEE754-2008 correctly rounded conversion of arbitrary length decimal character sequences to binary floating-point without any dynamic memory allocation. The algorithm we propose has been described by one of the Ph.D. students we advised in her Ph.D. manuscript [145]; we shall present it slightly differently in that Section.

In order to set a correct framing, let us precisely state what we understand by supporting IEEE754-2008. The IEEE754 Standard leaves a large amount of freedom to implementer's of the Standard in terms of what is called "mandatory" to support the Standard. An implementation can be achieved in hardware, software or a combination of both [117]. In order to claim compliance, an implementation needs to support at least one IEEE754-2008 format in at least one of the two possible radices $2$ and $10$. Hence at least support for the binary32 format is required [117]. For this given format, all mandatory operations need to be implemented, which sum up to $83$ operations [117]. These mandatory operations include addition, subtraction, multiplication, division etc. but also operations like conversion from and to character sequences. They do not include support for mathematical functions, like $\exp$. Those operations are just recommended by the Standard for implementation [117].

If only one format is implemented, there is no trouble with heterogeneous operations, as those operations mix two different formats in input and output. However, as per the expectations already set by IEEE754-1985, it is common to consider that at least the binary32 and the binary64 format need to be implemented for reasonable support of IEEE754. We shall hence also consider that at least this level of support is required. In this case, however, we need to support heterogeneous operations. The overall number of mandatory operations we need to support hence increases to $354$ operations [117].

Out of these $354$ operations that are mandatory for support of binary32 and binary64 in IEEE754-2008, the vast majority directly maps either to hardware instructions offered by most of the current systems, may they come from Intel, ARM or other vendors, or, by conversion of arguments for example, to other operations that are also required [117]. At the end of the day, only $8$ operations are non-trivial to implement:

— correctly rounded heterogeneous operations for $+, -, \times, /, \sqrt{}$ and $\mathrm{FMA}$ with all arguments being in binary64 format and the output being in binary32 format,

— conversions from decimal character sequences to binary32 and binary64.

Someone might argue that the FMA operation is not always supported in hardware and that support for it may require using software emulation. This is certainly true. However, we consider this software emulation to be easy to design as it can use state-of-the-art principles, such as the ones we shall describe in Section 3.5 for extended precision arithmetic.

Similarly, we might argue that the mandatory part of IEEE754-2008 includes operations like conversions from binary floating-point to decimal character sequences or conversions from hexadecimal character sequences to binary floating-point, none of which is supported in hardware and none of which is completely trivial to implement. Once again, these operations can however tackled with state-of-the-art approaches. For example, for conversion from binary floating-point to decimal character sequences, the approach described in [49] is helpful. We shall reuse that approach for other means, viz. to perform exact mixed-radix comparisons, in Section 3.3.

A support library for IEEE754-2008 hence has to go the tedious route of implementing $354$ functions. However, the interesting part is in $8$ operations, which we shall discuss in Sections 3.2.2 and 3.2.3.

### 3.2.1 A Support Library for IEEE754-2008: `libieee754`

The level of support for IEEE754-2008 varies, in particular depending on what one might call "support". The IEEE754-2008 formats binary32 and binary64 are of course bound by most compilers (like gcc, icc or clang) to the existing C-language `float` and `double` datatypes and most IEEE754-2008 operations are available through infix operators like `+`, `*` or (builtin-) functions like `sqrt`, declared in `math.h`. However, to our knowledge, there is no comprehensive library supporting in on single place and with a clear, common calling interface all mandatory binary floating-point IEEE754-2008 operations but Intel's proprietary `libbfp754` library [1]. Intel ships this library as a compiled binary with its compilers.

In order to provide an open-source variant of that Intel proprietary library and, more importantly, to give us a test-bed for the design of algorithms for heterogeneous operations, like we will discuss then in Section 3.2.2, or our character sequence comparisons described in Section 3.2.3, we started work on a library which we call `libieee754`. Access to the library is granted at

`https://gitlab.com/cquirin/libieee754.`

We have published on that library with [161].

The library is supposed to implement all the $354$ operations IEEE754-2008 mandates for binary floating-point arithmetic in both binary32 and binary64 formats. While the library should be reasonably fast, speed should not be the main purpose but 100% standard compliance. All operations support all rounding modes and set all flags as required by IEEE754-2008. The library is reentrant as there is no global state other than the global state foreseen by the standard.

The library is not fully complete, in particular, the testing functionality is under-developed. The maintenance cost of the library is relatively high, for a project that is supposed to be contributed to mostly by graduate students.

However, as already stated, the library contains code implementing our ideas for the heterogeneous operations, which we are going to discuss in the next Section.

### 3.2.2 Ziv's Rounding Test Reused To Implement Heterogeneous Operations

Several approaches have been proposed for the implementation of the heterogeneous operations in IEEE754-2008. Again, these operations take arguments in one format, say binary64, and return their result in a lower precision format, such as binary32. As per IEEE754-2008, there must not be more than one rounding and this rounding must be performed as if the operation were executed with infinite precision and then rounded once [117].

1. `https://software.intel.com/en-us/node/524476`

A naive approach would be to nevertheless perform two roundings and to hope for the best. For a binary64 $\times$ binary64 addition yielding a binary32 format, the corresponding algorithm would read as follows. Let be $x, y \in \mathbb{F}_{53}^{11}$ the operands and assume that we want to compute $w = \diamond_{24}(x+y)$, where $w \in \mathbb{F}_{24}^{8}$. Inhere, $\diamond_k \in \{\circ_k, \triangledown_k, \vartriangle_k, \bowtie_k\}$ is any of the IEEE754-2008 roundings to a precision of $k$ bits (and the corresponding, implicitly assumed IEEE754-2008 exponent range). The naive code reads:

$$\begin{array}{rcl} z & \leftarrow & \diamond_{53}(x+y) \\ \zeta & \leftarrow & \diamond_{24}(z) \\ w & \leftarrow & \zeta \end{array}$$

As a matter of course, as we have already seen in Section 2.2.2, this naive is not correct, as even though $w \in \mathbb{F}_{24}^{8}$ holds, it is not the correct rounding of $x+y$ to binary32 for rounding-to-nearest, $\diamond = \circ$, on which a double rounding issue occurs.

**Problem statement**

However, we may want to use this incorrect, naive algorithm for the heterogeneous operations as a basis to develop a correct algorithm. This approach is sensible: while double rounding issues may occur, they occur only seldom for well-distributed inputs $x$ and $y$. In particular, a double rounding issue occurring requires the first binary64 rounding that yields $z$ to be to a binary32 midpoint. These midpoints are of course rare.

We shall therefore set out to design an algorithm based on the naive one that

— starts by detecting whether a double rounding issue may occur on a given input $x, y$ and that

— is able to correct the rounding so that $w$ becomes equal to the correct rounding $\diamond_{24}(x+y)$.

For short, we want to replace the trivial copy $w \leftarrow \zeta$ of the last line of the algorithm by a detection and correction sequence.

Let us now observe that a similar problem arises in the implementation of correctly rounded elementary functions [165,196]. In order to achieve a correct rounding for an elementary function, which is transcendental and which we can do nothing about but approximate it, we need to have a way of checking whether an approximation to that function is already accurate enough to allow for correct rounding. Our heterogeneous operation problem looks different but is quite similar: in the naive algorithm, $z$ is a pretty accurate approximation to the exact result. We know that for round-to-nearest, $\diamond = \circ$ there exists[2] $\varepsilon \in \mathbb{R}$, bounded by $|\varepsilon| \leqslant 2^{-53}$ such that

$$z = (x+y) \cdot (1+\varepsilon) \, .$$

Rounding such an approximation to the exact, infinitely precise value $x+y$ should be possible in most cases: the approximation is accurate up to a relative error of $2^{-53}$ and we want to round to a binary32, which has 24 bits of precision. In other words, we have 29 "guard bits".

2. Subnormal rounding where such relative-error $\varepsilon$ would of course not exist is not an issue here: if the first, binary64 rounding was to a subnormal number, the second binary32 rounding would be completely underflowed as the binary32 exponent range is way smaller than the binary64 one.

### Heterogeneous Operations Seen as A "Main+Correcting" Term Problem

The step towards designing a correctly rounded algorithm is to separate the bits in the candidate rounding $\zeta$ from the guard bits in $z$. This is actually pretty easy: as a matter of course, $\zeta \in \mathbb{F}_{24}^{8}$ is exactly representable as a binary64 number, too. The rounding of $z$ to $\zeta$ being towards a binary32, unless it gradually or completely underflows or overflows, will not affect $z$ by more than $2^{-24}$ in relative error. If gradual binary32 underflow occurs, the error will always be less or equal to half the subnormal that this binary32 rounding yields. Hence we have in all cases:

$$\frac{1}{2}\,|z| \leqslant |\zeta| \leqslant 2\,|z|\,.$$

The condition of Sterbenz' lemma [238] is hence satisfied and we can modify our algorithm as follows:

$$\begin{array}{rcl} z & \leftarrow & \diamond_{53}(x+y) \\ \zeta & \leftarrow & \diamond_{53}\left(\diamond_{24}(z)\right) \\ \delta & \leftarrow & \diamond_{53}(z-\zeta) \\ w & \leftarrow & \diamond_{24}(\zeta). \end{array}$$

For this algorithm, we can state the following properties:

$$\begin{array}{rcl} z & = & (x+y)\cdot(1+\varepsilon) \text{ with } |\varepsilon| \leqslant 2^{-53} \\ \zeta+\delta & = & z \\ \diamond_{24}(\zeta) & = & \diamond_{24}(z). \end{array}$$

This means we have an instance of a problem where some approximation $z$ of an exact quantity, for instance $x+y$, is available and this approximation is split onto two floating-point variables $\zeta$ and $\delta$. Out of these two, $\zeta$ is a main term for which we want to determine whether it is equal to the correct rounding of the exact quantity. The other term $\delta$, is a correction term, of this candidate rounding $\zeta$, added in order to obtain the accurate approximation $z$. In some sense, $\delta$ holds the guard bits of the accurate approximation of the exact result $x+y$.

### Ziv's Rounding Test

Ziv's rounding test is the state-of-the-art solution to the problem we are confronted with: an exact quantity can only be approximated, it is store as an unevaluated sum of a main and a correction term and we need to decide whether the main term corresponds to the correct rounding of the exact quantity. The test has been invented by Ziv [264]; we have published on some improvements on how to determine a certain constant that is required for the implementation of the test [66].

The basic idea of Ziv's rounding test goes like this: we assume that the rounding mode is round-to-nearest. The main term, in our case $\zeta$, is at a floating-point number that is a candidate rounding of some exact quantity, for instance $x+y$. From that main term a correcting term $\delta$ points towards the approximation $z=\zeta+\delta$ of the exact quantity $x+y$. The candidate rounding is the correct one if the approximation $\zeta+\delta=z=\diamond_{53}(x+y)$ is not around the midpoint of the two floating-point numbers straddling the exact result; it is at the midpoint that the rounding changes. The main term is one of these two floating-point

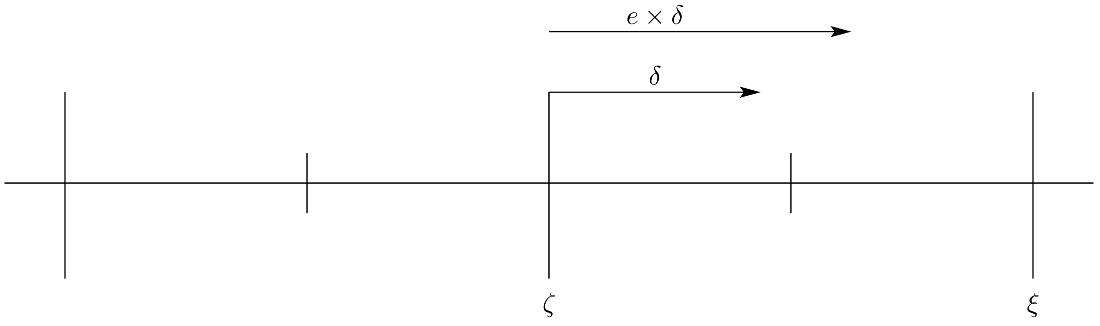

Figure 3.1 – Ziv's rounding test

numbers, the other one is the floating-point number strictly below or above, depending on the direction $\delta$ points to.

Now, as Figure 3.1 illustrates, if $\delta$ makes the approximation $z$ being close to this midpoint, a small relative increase $e > 1$ of $\delta$ will make $\zeta + e\,\delta$ go beyond the midpoint and make the rounding $\xi = \circ_{24}(\zeta + e \times \delta)$ change with respect to $\zeta$. In other words, if $z$ is close to the midpoint, $\delta$ will be, in magnitude, just below or equal to $1/2\,\mathrm{ulp}_{24}(z)$. Increasing $\delta$ by multiplying it by $e > 1$ will make $\delta$ larger in magnitude than $1/2\,\mathrm{ulp}_{24}(z)$, and the rounding of $\zeta + e \times \delta$, namely $\xi = \circ_{24}(\zeta + e \times \delta)$ will be different from $\zeta$.

On the other hand, if the constant $e$ is correctly chosen with respect to the accuracy of $z$, whenever $\xi = \zeta$, the approximation $z$ and hence the exact value $x + y$ is far enough from the midpoint and $\zeta$ is already the correct rounding. Figure 3.1 illustrates this.

```
Input: $x, y \in \mathbb{F}_{53}^{11}$ two binary64 inputs, $e > 1$ an appropriate Ziv's rounding test constant
Output: $\circ_{24}(x + y)$ the binary32 correct rounding of $x + y$
$z \leftarrow \circ_{53}(x + y)$ ;
$\zeta \leftarrow \circ_{53}(\circ_{24}(z))$ ;
$\delta \leftarrow \circ_{53}(z - \zeta)$ ;
$\sigma \leftarrow \circ_{53}(e \times \delta)$ ;
$\tau \leftarrow \circ_{53}(\zeta + \sigma)$ ;
$\xi \leftarrow \circ_{53}(\circ_{24}(\tau))$ ;
if $\zeta = \xi$ then
    $w \leftarrow \circ_{24}(\zeta)$;
    return $w$;
end
/* Correct rounding not possible, compute correct rounding */
return ...;
```

**Algorithm 1:** Candidate algorithm for heterogeneous addition

Unless an FMA operation is available, the quantity $\circ_{24}(\zeta + e \times \delta)$ cannot be computed with one rounding $\circ_{24}$ as $e \times \delta$ needs evaluating. An additional error here is however not an issue if it is properly accounted for when determining the constant $e$. The details on how to determine $e$ properly can be found in [66]. Similarly, an additional error due to double rounding when determining the quantity $\xi$ to compare $\zeta$ to is not an issue as long as the error gets accounted for in $e$. Assuming $e$ as correctly determined, we can hence modify our algorithm again so that it becomes Algorithm 1.

Algorithm 1 is the first algorithm that returns the correct rounding for an heterogeneous addition operation - when it does return something. The algorithm is still incomplete. In cases when $\zeta$ cannot be shown to be the correct rounding, no answer has been determined, yet.

**Heterogeneous Operations: An Algorithm**

```
Input: $x, y \in \mathbb{F}_{53}^{11}$ two binary64 inputs, $e > 1$ an appropriate Ziv's rounding test constant
Output: $\diamond_{24}(x + y)$ the binary32 correct rounding of $x + y$
$z \leftarrow \diamond_{53}(x + y)$ ;
$\zeta \leftarrow \diamond_{53}(\diamond_{24}(z))$ ;
$\delta \leftarrow \diamond_{53}(z - \zeta)$ ;
$\sigma \leftarrow \diamond_{53}(e \times \delta)$ ;
$\tau \leftarrow \diamond_{53}(\zeta + \sigma)$ ;
$\xi \leftarrow \diamond_{53}(\diamond_{24}(\tau))$ ;
if $\zeta = \xi$ then
    $w \leftarrow \diamond_{24}(\zeta)$;
    return $w$;
end
$\mu \leftarrow \diamond_{53}(1/2 \times \diamond_{53}(\zeta + \xi))$;
if $z \neq \mu$ then return $\diamond_{24}(\zeta)$;
$\alpha_1 \leftarrow \diamond_{53}(z - x)$;
$\alpha_2 \leftarrow \diamond_{53}(z - \alpha_1)$;
$\delta_1 \leftarrow \diamond_{53}(y - \alpha_1)$; $\delta_2 \leftarrow \diamond_{53}(x - \alpha_2)$;
$\varepsilon \leftarrow \diamond_{53}(\delta_1 + \delta_2)$ ;
if $((\varepsilon = 0) \vee ((\varepsilon > 0)$ xor $(\delta > 0)))$ then return $\diamond_{24}(\zeta)$;
return $\diamond_{24}(\xi)$;
```

**Algorithm 2:** Elaborated candidate algorithm for heterogeneous addition

In order to complete the algorithm, we first need to find a way to refine the answer provided by Ziv's rounding test. That test of course does not yield false positives in the sense that it will not deem an instance to be correctly rounded when it is not. However, it does classify some instances as *might not be the correct rounding* while $\zeta$ is the correct rounding. For this refinement, we need to express the decisive quantity, the midpoint $\mu$ of the two floating-point numbers straddling the exact result $x + y$ in the algorithm. We can then compare $z$ to this midpoint.

Let us first observe that the midpoint $\mu$ of the two floating-point numbers *in the less precise format* we eventually round to can be expressed exactly *in the more precise format* we use for the first rounding. As a matter of fact, a midpoint in a floating-point format of $k$ bits of precision holds on $k + 1$ bits. The IEEE754-2008 formats are such that the next more precise format is always at least 1 bit more precise than the preceding one. For binary32 and binary64, the difference in precision is even 29 bits [117].

It is then important to show that if they are unequal, the two values $\zeta$ and $\xi$ as they get computed by Algorithm 1 are consecutive floating-point numbers in the less precise floating-point format, for instance binary32 with a precision $k = 24$. It is easy to see that they hold on $k = 24$ bits. The fact that they are consecutive if unequal can be established with an elaboration of the following argument: the error $\delta$ between $z$ and $\zeta$ is bounded in magnitude by half an $\mathrm{ulp}_{24}$ of $\zeta$ [3]. If Ziv's rounding constant $e > 1 > 0$ is then bounded by $3/2$, which is always the case and easy to establish [66], $\sigma$ is bounded in magnitude by $3/4\,\mathrm{ulp}_{24}(\zeta)$. As it is very easy to show that $1/2\,\mathrm{ulp}_{53}(\tau) \leqslant 2^{-29}\,\mathrm{ulp}_{24}(\tau)$, the difference between $\tau$ and $\zeta$ is hence strict bounded in magnitude by $\mathrm{ulp}_{24}(\zeta)$ [4], the rounding $\circ_{24}(\tau)$ will never be beyond the 24 bit floating-point number consecutive to $\zeta$ in the direction of $\delta$. Hence $\zeta$ and $\xi$ are either equal or consecutive floating-point numbers in the less precise format.

Assuming now that $\zeta$ and $\xi$ are unequal, it is hence possible to express the midpoint $\mu$ between $\zeta$ and $\xi$ as $\mu = 1/2\ (\zeta + \xi)$. As $\mu$ is representable in the more precise format, so is $2\,\mu$ [5]. Hence the rounding $\circ_{53}(\zeta + \xi)$ is errorfree. We can therefore actually compute $\mu$ as $\mu = \circ_{53}\left(1/2 \times \circ_{53}\left(\zeta + \xi\right)\right)$ without running into trouble as we would for regular floating-point averages [18].

It is pretty easy to show that a double-rounding issue can come up in the implementation of heterogeneous operations only if the first rounding in the wider format, for instance the rounding $z = \circ_{53}(x + y)$, exactly goes to a midpoint in the less precise format. In all other cases, as this first rounding is a correct one, the bits following the bit marking the midpoint between the two floating-point numbers straddling the two less precise format floating-point numbers around $x + y$ serve as sticky bits [77]. We can hence refine Ziv's rounding test with an additional test that returns the correct result $\circ_{24}(\zeta)$ in all cases when $z$ is different from $\mu$, the midpoint.

If, in contrast, the first rounding that yielded $z$ was to the midpoint $\mu$, a double-rounding occurred in the second rounding that yielded $\zeta$ iff the signs of the errors $\delta$ and the error $\varepsilon = (x+y) - z$ are the same and the first rounding yielding $z$ was inexact. A double-rounding occurring implies that both roundings $x + y$ yielding $z$ and $z$ yielding $\zeta$ go into the same direction, which is equivalent to the signs of $\delta = z - \zeta$ and $\varepsilon = (x + y) - z = (x + y) - \zeta - \delta$ having the same sign. Fortunately, the error $\varepsilon$ can be computed exactly in floating-point arithmetic using the TwoSum sequence [71, 97, 192]. We can hence give a new candidate algorithm for an heterogeneous addition with Algorithm 2.

Unfortunately, Algorithm 2 still does not cover all possible cases of IEEE754-2008 heterogeneous addition. Until now, we have assumed that the exponent ranges for both the wider format, binary64, and the less precise format, binary32, were unbounded. This is clearly not the case. We must make sure that overflow and binary32 underflow are correctly handled by our algorithm. Further IEEE754-2008 also defines the behavior of operations on non-finite inputs, i.e. on infinities, and for NaN inputs. More importantly, IEEE754-2008 not only defines the floating-point results of operations but also their side-effects in terms of IEEE754-2008 exceptions, resp. the raising of IEEE754-2008 flags [117]. We must make sure that our algorithm works in all these cases.

Overflow handling requires us to address one double-rounding case in particular: when $|z|$ is equal to the largest finite binary32 number plus $1/2\,\mathrm{ulp}_{24}$ of that largest binary32 number, i.e. if it is equal to the 24 bit floating-point "midpoint" that we would observe if the exponent range were unbounded, our current way of computing $\zeta = \circ_{53}(\circ_{24}(x + y))$ yields to infinity as $\circ_{24}(x + y)$ overflows as per IEEE754-2008 [117] whereas for the correct working of our

3. The actual proof, which is too lengthy to be included inhere, needs to distinguish between the case when $\zeta$ is an integer power of 2 and $\delta$ is of opposite sign of $\zeta$ –in which case the bound is $1/4\,\mathrm{ulp}_{24}(\zeta)$– and the other, regular case.

4. resp. $1/2\,\mathrm{ulp}_{24}(\zeta)$ for the integer-power-of-2 case

5. Assuming unbounded exponent, which we can as will be shown below.

algorithm the 24 bit floating-point number just above $x + y$ ought to be stored in $\zeta$. We can address this issue with an additional test and some scaling. A second test then can filter all cases when $|z|$ is strictly greater than the “midpoint” following the largest finite binary32 number, as no double-rounding issue can affect the rounding towards infinity anymore.

The handling of NaN inputs is relatively easy. The same applies to infinities. The IEEE754-2008 rules for NaN and infinity handling are the same for all formats. We just need to make sure to return $\zeta$ in such a case. This is easy to handle as we can add tests in the existing branching statements to make this happen.

```
**Input:** $x, y \in \mathbb{F}_{53}^{11}$ two binary64 inputs, $e > 1$ an appropriate Ziv's rounding test constant
**Output:** $\diamond_{24}(x + y)$ the binary32 correct rounding of $x + y$ and all flags set correctly
$z \leftarrow \diamond_{53}(x + y)$ ;
$\bar{z} \leftarrow |z|$ ;
**if** $\bar{z} = 2^{128} \cdot (1 - 2^{-25})$ **then** $\zeta \leftarrow \diamond_{53}\left(2 \times \diamond_{53}(\diamond_{24}(\diamond_{53}(1/2 \times z)))\right)$; **else** $\zeta \leftarrow \diamond_{53}\left(\diamond_{24}(z)\right)$;
$\bar{\zeta} \leftarrow |\zeta|$ ;
**if** $((\bar{\zeta} \neq \bar{\zeta}) \vee (\bar{z} > 2^{128} \cdot (1 - 2^{-25}))$ **then return** $\diamond_{24}(\zeta)$;
$\delta \leftarrow \diamond_{53}(z - \zeta)$ ;
$\sigma \leftarrow \diamond_{53}(e \times \delta)$ ;
$\tau \leftarrow \diamond_{53}(\zeta + \sigma)$ ;
$\xi \leftarrow \diamond_{53}\left(\diamond_{24}(\tau)\right)$ ;
**if** $\neg(\zeta \neq \xi)$ **then**
    **if** $((\delta = 0) \wedge (\bar{\zeta} \leqslant 2^{-126}))$ **then**
        **if** $(|x| \geqslant |y|)$ **then** $a \leftarrow x$; $b \leftarrow y$; **else** $a \leftarrow y$; $b \leftarrow x$;
        $t \leftarrow \diamond_{53}(z - a)$ ;
        **if** $t \neq b$ **then**
            $d_1 \leftarrow \diamond_{53}(z \times \left(1 + 2^{-28}\right))$ ;
            $d_2 \leftarrow \diamond_{24}(d_1)$;
        **end**
    **end**
    $w \leftarrow \diamond_{24}(\zeta)$;
    **return** $w$;
**end**
$\mu \leftarrow \diamond_{53}\left(1/2 \times \diamond_{53}\left(\zeta + \xi\right)\right)$;
**if** $z \neq \mu$ **then return** $\diamond_{24}(\zeta)$;
$\alpha_1 \leftarrow \diamond_{53}(z - x)$;
$\alpha_2 \leftarrow \diamond_{53}(z - \alpha_1)$;
$\delta_1 \leftarrow \diamond_{53}(y - \alpha_1)$; $\delta_2 \leftarrow \diamond_{53}(x - \alpha_2)$;
$\varepsilon \leftarrow \diamond_{53}(\delta_1 + \delta_2)$ ;
**if** $((\varepsilon = 0) \vee ((\varepsilon > 0) \,\mathtt{xor}\, (\delta > 0)))$ **then return** $\diamond_{24}(\zeta)$;
**return** $\diamond_{24}(\xi)$;
```

**Algorithm 3:** Algorithm for IEEE754-2008 heterogeneous addition

The handling of underflow is on the one hand easy but requires care on the other hand. It is easy as Algorithm 2 already works for underflow cases. Underflow cases of the more precise format, binary64, are so way beyond the complete underflow threshold for the less precise format, binary32, that the do not require additional tests. Ziv's rounding test will always make $\zeta$ be returned. Underflow cases in the less precise format, binary32, are no different from all other normal cases, as Ziv's rounding test and all logic to compute the midpoints are executed in binary64 floating-point arithmetic. Underflowed binary32 cases are just normal numbers in binary64. On the other hand, a phenomenon previously unknown occurs for IEEE754-2008 heterogeneous addition: unlike homogeneous addition results [197], underflowed heterogeneous addition results can be inexact. As we want to get the IEEE754-2008 flags get set correctly, we need to address this point. As it stands, Algorithm 2 will return correctly rounded results for underflowed cases but it will not set the underflow nor inexact flag for cases when the second rounding producing the binary32 number $\zeta$ out of the binary64 candidate $z$ is exact. IEEE754-2008 specifies that the underflow flag be raised for an operation that produces a result below the underflow threshold only if that operation was inexact at the same time. The inexact flag is raised, too [117]. In cases an underflowed IEEE754-2008 operation is exact, for default exception handling, the underflow flag is not set. For our heterogeneous algorithm, we hence need to add special handling to detect this case and to set the underflow flag, typically by executing a dummy IEEE754-2008 operation that produces an inexact underflowed result, which we throw away. Such a test is readily added: the case can only occur for $\delta = 0$, $|\zeta|$ below the underflow threshold and the rounding $z = \circ_{53}(x + y)$ being inexact.

Algorithm 3 includes all the changes required for the correct handling of overflow, underflow with flag setting, infinities and NaN inputs. Besides correctly setting the flags for underflowed cases, it sets all the required flags and only the required flags correctly. The proof that this part of its behavior is correct would go beyond the purpose of this document. The proof essentially requires considering all cases of inputs that IEEE754-2008 requires to raise a certain flag and to exhibit an operation in the algorithm that raises the flag as well as going over all operations in the algorithm, considering the cases when they do raise flags and ruling out the possibility that spurious flags get erroneously raised.

#### Directed rounding modes

IEEE754-2008 defines several rounding modes, such as round-to-nearest or the directed rounding modes like round-towards-infinity [117]. As a matter of course, the IEEE754-2008 heterogeneous operations need to be supported in all of these rounding modes. In the discussion that precedes, we have silently assumed the rounding mode to be round-to-nearest. This is also the only rounding mode in which Ziv's rounding test classically works [66]. An algorithm to support for the heterogeneous operations in all rounding modes might hence proceed as follows: first, test the rounding mode, if the rounding mode is round-to-nearest, execute Algorithm 3, otherwise execute another algorithm for the directed rounding modes.

However, this scheme would be slow: access to the prevailing rounding mode is extremely slow on most architectures [162, 230]. The fall-back technique that consists in executing a couple of dummy operations and determining the prevailing rounding mode from their results is contradictory to our requirement that the final algorithm should only raise the flags required by IEEE754-2008: the inexact flag would be spuriously raised by the dummy operations. Further, that other algorithm to be executed for the directed rounding would need to be implemented, even if it is pretty small. As a matter of fact, no double-rounding issue may occur for the directed rounding modes: in the directed roundings modes, all rounding boundaries are at the floating-point numbers itself and all lesser precision floating-point numbers are included in the wider format's floating-point number set.

This latter point also means that an algorithm for the directed rounding modes heterogeneous operations is included in the first lines of Algorithm 3. In the directed rounding modes,

$w = \circ_{24}(\zeta)$ always is the correct heterogeneous rounding. If we could make Ziv's rounding mode always decide that rounding is possible in the directed rounding modes so that $w$ gets returned, our algorithm would work in any rounding mode.

As it turns out, Algorithm 3 is already agnostic to the rounding mode; no further modification is required. As a matter of course, Ziv's rounding test does work in the sense that it would detect double-roundings when it is executed in a directed rounding mode. But as in the very particular case we are considering there are no double-roundings, when Ziv's rounding test classifies an instance as "correct", this rounding-test-answer is not sensible, but as $w$ is the correct rounding nevertheless, the algorithm stays correct.

Turned differently, Algorithm 3 is correct also for the directed rounding modes, unless the line where $\circ_{24}(\xi)$ is returned instead of $w = \circ_{24}(\zeta)$. It is easy to see that this is not possible for the directed rounding modes: the signs of $\delta$, the absolute error of a rounding, and $\varepsilon$, the absolute error of another rounding, are always the same if the rounding mode is directed and both rounding operations are executed with the same directed rounding mode.

The reader might at this point raise the issue that making the last statement, we have considered that $\varepsilon$ is computed correctly – with a TwoSum sequence that, classically, does not work for the directed roundings [197]. The reader is of course right; however in the very particular case the TwoSum sequence is used in Algorithm 3 –only when $z$ is equal to the midpoint $\mu$– at least the sign of $\varepsilon$ is correct even though the TwoSum sequence is executed in an improper rounding mode. And the sign of $\varepsilon$ is the only piece of information that needs to be correct. A formal proof of this last statement would go far beyond the purpose of this document; we shall note that a recent result by Boldo et al. also implies the correctness of Algorithm 3 for the directed rounding modes [20].

### Subtraction, Multiplication, Division and Square Root

Similarly to the support of the directed rounding modes that IEEE754-2008 requires but that we did not originally consider with Algorithm 3, IEEE754-2008 requires the implementation of heterogeneous operations also for subtraction, multiplication, division, square root and fused-multiply-add. We shall exclude fused-multiply-add from the discussion at first, as it is by itself an operation that was added to the IEEE754 Standard with the 2008 revision.

For subtraction, Algorithm 3 needs to be modified only slightly. This modification is trivial. For multiplication, the modification is more substantial: the TwoSum sequence in the end needs to be replaced by a TwoMul sequence [197]. Care needs to be taken with respect to subnormal numbers occurring in that TwoMul sequence. Proper scaling allows for the correct computation of the sign of $\varepsilon$ notwithstanding an underflowed $z$. The modified FastTwoMul sequence used for the correct setting of flags in the the underflow case needs to be replaced by a test that establishes if $z \neq x \times y$ by other means. The use of a TwoMul sequence is likely to be the easiest solution for this purpose.

For division and square root, no direct errorfree transformations exist as they do for addition and multiplication with the TwoAdd resp. TwoMul sequences. As Algorithm 3 never relies on the actual value of the correcting term but only the sign of the error $\varepsilon$ (resp. of $t - b$), it is possible to modify the algorithm for support of division and square root. The sign of the error of $z = \circ_{53}(x/y)$ resp. $z = \circ_{53}(\sqrt{x})$ can be expressed in terms of $x - z \times y$ with respect to the sign of $x$ resp. in terms of $x - z \times z$ for square root – where $x$ is necessarily positive or zero. These quantities $x - z \times y$ and $x - z \times z$ can be exactly computed as floating-point numbers, utilizing the TwoMul sequence and Sterbenz' lemma [197,238].

For fused-multiply-add, the modifications required to adapt Algorithm 3 are still more heavy but nevertheless possible. An exact result $x \times y + u$ can be computed using the TwoMul and TwoSum sequences, yielding a result on three binary64 bit numbers [165]. Similarly, the error of a homogeneous fused-multiply-add can be computed exactly on two floating-point numbers [23]. As the three elements of this floating-point expansion holding $x \times y + u$ are ordered by magnitude, deducing the sign of the error $\varepsilon$ is possible. Extreme care to avoid spurious underflow or underflow and undue flag settings is required, though. However, on platforms where the fused-multiply-add operation is not implemented in hardware but requires emulation in software anyway, the use of our algorithm is not sensible: one would rather start from the software emulation of an heterogeneous operation and modify the last emulated rounding step in order to round to a narrower format.

#### Conclusion

In this Section, we have shown how Ziv's rounding test, a classical floating-point building block for the implementation of elementary functions, can be "recycled" to address the issue of implementation of the IEEE754-2008 heterogeneous operations, which take operands in a floating-point format larger than the floating-point format they return their results in.

With Algorithm 3 we have given an algorithm for the heterogeneous addition that is agnostic to the prevailing rounding mode while still working in all rounding modes. It is carefully designed to raise all flags required by IEEE754-2008 for heterogeneous addition and raise no other flags than the ones required. We have briefly outlined how Algorithm 3 can be modified to support subtraction, multiplication, division, square-root and fused-multiply-add, too.

We shall draw the reader's attention to the fact that Algorithm 3 also allows to emulate a homogeneous binary32 fused-multiply-add operation on systems that do not implement this operation in hardware but support binary64. The three inputs are readily upconverted to binary64 before multiplying the multiplication operands exactly with a binary64 homogeneous multiplication. The rest of the work corresponds to adding two binary64 floating-point numbers, performing one correct rounding to binary32, which is what our algorithm does.

From a more theoretical standpoint, let us add that the fact that we could propose an algorithm to perform the IEEE754-2008 heterogeneous operations with nothing but IEEE754-1985 homogeneous operations [116] means that the addition of these operations in the 2008 revision did not extend what can be computed within the set of the IEEE754 Standard's operations.

### 3.2.3 Correct Rounding of Arbitrary Length Decimal-To-Binary Conversions

For quite some time, the world of floating-point arithmetic has been separated into two parts: the user facing part, where data was displayed and parsed in decimal, and the more hidden part of the actual computation that was carried out in binary floating-point arithmetic. IEEE754-2008 did partially alleviate this schism: it added support for decimal floating-point arithmetic to the Standard, allowing data to stay in decimal all the way through the computation from input, through processing until the output [117].

A huge amount of software, written using binary IEEE754 floating-point arithmetic, continues to exist and serve its purpose. While support for binary floating-point arithmetic starts to enter even the market of the smallest devices [3,184], support for decimal arithmetic, in particular in hardware, stays limited [36].

We hence continue to experience the need to convert decimal user input, i.e. character sequences written with the decimal digits $0$ through $9$ and some special characters to mark the fractional part of a number, like the point, or to mark the beginning of the decimal exponent, to binary IEEE754 floating-point numbers. To this input conversion adds the respective output conversion from binary floating-point to decimal character sequences. The latter conversion can essentially follow the classical approaches for binary-to-decimal conversion, as shown in [49] and as we shall reuse them in Section 3.3.

The input conversion from decimal character sequences to binary floating-point is challenging when we wish to make it correctly rounded: all classical techniques for correct rounding are based on the observation that for a function $f$ with some discrete input $x \in \mathbb{F}$ that must be rounded to a discrete output $y \in \mathbb{F}$, there exists a certain worst-case accuracy that is required to distinguish between cases when $f(x)$ is exactly representable and when it needs rounding to the discrete output [173,197]. For decimal character sequences this is no longer the case: the decimal input $x$ can be of arbitrary length, i.e. it can be arbitrarily close to the preimage $f^{-1}(y)$ of a binary floating-point number $y \in \mathbb{F}$. We can hence not precompute worst-case accuracies for these decimal character sequence to binary floating-point conversions.

For example, take $\check{x} = 2^{24} + 1 = 16777217$. This midpoint number is exactly halfway between the two IEEE754 binary32 floating-point numbers $y_1 = 2^{24} \in \mathbb{F}_{24}$ and $y_2 = 2^{24}+2 \in \mathbb{F}_{24}$. Now imagine the character sequence $x =$ `16777217.0000000000000000000000000001`. The conversion software can decide the rounding to nearest only once it has read in –and converted– the whole bunch of zeros followed by that trailing one. As a matter of course, the amount of zeros is unbounded and the conversion routine needs to keep going as long as the input goes.

**State of the art**

Libraries, such as glibc [179], that do offer correct rounding –at least the default rounding mode round-to-nearest– typically perform the conversion as sketched above: they represent the whole input as an arbitrary length integer $p \in \mathbb{N}$ that represents the input digits with the decimal point dropped, and another arbitrary length integer $q \in \mathbb{N}$ that represents the power of $10$ that is appropriate for the position of the decimal point and the specified exponent. These two integers $p$ and $q$ form a rational number $p/q$. The libraries continue by determining an appropriate binary output exponent $E$, typically by determining the respective bit-lengths of $p$ and $q$. They then form another rational number $p'/q' = 2^{-E}\,p/q$ that eventually shall become the integer mantissa $m$ closest to the input number, before $2^E \cdot m$ gets output as an IEEE754-2008 binary floating-point number. Determining $m$ out of $p'/q'$ can be performed essentially using a long division algorithm [179].

For the example of the sequence $x =$ `16777217.0000000000000000000000000001`, existing conversion software hence forms the two integers

$$p = 167772170000000000000000000000000001$$

and

$$q = 10000000000000000000000000000.$$

It then determines that $M = \lceil \log_2(p) \rceil = 118$ and $N = \lceil \log_2(q) \rceil = 94$. This means that $E = M - N - k + 1 = 118 - 94 + 24 + 1 = 1$ is an appropriate exponent for a binary32 floating-point number with precision $k = 24$. The number $p'/q'$ is hence formed as $p' = 1677721700000000000000000000000001$ and $q' = 20000000000000000000000000000$. Long division yields $m$ and $r$ such that $m'\,q' + r = p'$, where $m' = 8388608$ and $r = 10000000000000000000000000001$. As $r$ is just above $1/2\,q' = 10000000000000000000000000000$, round-up needs to be performed, correcting $m'$ to $m = 8388609$. The closest binary32 number hence is $2^E \cdot m = 2^1 \cdot 8388609$.

As a matter of course, the algorithm presented above has the disadvantage of requiring dynamic memory allocation for the long integers $p$ and $q$. The maximum memory use of a process using decimal character sequence scan routine such as `scanf` from a library as glibc hence cannot be predicted. The memory usage is typically $\mathcal{O}(t)$ where $t$ is the size of the input. Further, the complexity of the long division is at least $\mathcal{O}(t^\star)$; for real-life implementations one might even find $\mathcal{O}(t^2)$ [255].

### Arbitrary Length Inputs: No Need for Dynamic Memory Allocation

Our goal is hence to design an algorithm for the conversion of decimal character sequences to binary floating-point numbers in some IEEE754-2008 format, such as binary32 or binary64, that respect the following constraints:

— The algorithm is correctly rounded. This means that if only the last digit in some long character sequence decides the rounding, the algorithm will read the whole input.

— Its time complexity is hence $\mathcal{O}(t)$, where $t$ is the length of the input.

— The algorithm nevertheless uses static memory allocation only, meaning that its space complexity is $\mathcal{O}(1)$.

— The algorithm does not use long division for the sake of simplicity.

— Correct rounding is achieved in any of the IEEE754-2008 binary rounding modes. The algorithm is agnostic of the rounding mode; IEEE754-2008 operations are executed in such a way that the returned floating-point value is the correct rounding in the prevailing rounding mode.

The basic observation that will allow us to design a conversion algorithm to fit these constraints, is the following one: even though decimal to binary conversion is not always exact –$0.2$ being an easy example, as no integer power of 2 is divisible by $5$– the inverse conversion from binary floating-point to decimal floating-point can be made exact for an *a priori* known decimal precision $l \in \mathbb{N}, l \geqslant 2$ – under the condition that the exponent range of the binary floating-point format be bounded, which is the case for the IEEE754-2008 formats binary32 and binary64.

Typically, we have a number $2^E \cdot m$ in input with $m \in \mathbb{N}$ s.t. $2^{k-1} \leqslant m \leqslant 2^k - 1$ and $E \in \mathbb{N}$ s.t. $E_{\text{min}} \leqslant E \leqslant E_{\text{max}}$. For IEEE754-2008 binary64 for example, $k = 53$, $E_{\text{min}} = -1128$ and $E_{\text{max}} = 971$, as can easily be shown adapting the values given in Section 3.1 to a normalized integer mantissa. For the corresponding decimal number $10^F \cdot n$ in output similar bounds can be stated: $n \in \mathbb{N}$ with $10^{l-1} \leqslant n \leqslant 10^l -$ and $F \in \mathbb{N}$ with $F_{\text{min}} \leqslant F \leqslant F_{\text{max}}$. The fact that $F$ is bounded comes from the fact that $E$ is bounded: the binary input's dynamic is preserved by a conversion, the decimal output can only vary in orders of magnitude as much as the binary input does. Formally we have:

$$\begin{aligned} 2^E \cdot m &= 10^F \cdot n \\ 10^F &= \frac{2^E \cdot m}{n} \\ F &= \log_{10} \frac{2^E \cdot m}{n} \end{aligned}$$

and therefore

$$-l + \log_{10}\left(2^{E_{\text{min}}+k-1}\right) < F < -l + 1 + \log_{10}\left(2^{E_{\text{max}}+k}\right). \tag{3.1}$$

Given the fact that $F$ is integer, this reduces to

$$F_{\text{min}} \triangleq -l + \left\lceil \log_{10}\left(2^{E_{\text{min}}+k-1}\right)\right\rceil \leqslant F \leqslant -l + 1 + \left\lfloor \log_{10}\left(2^{E_{\text{max}}+k}\right)\right\rfloor \triangleq F_{\text{max}}. \tag{3.2}$$

Now, if the decimal precision $l \in \mathbb{N}$ is chosen such that

$$l \geqslant \left\lfloor \log_{10}\left(2^{E_{\text{max}}+k}\right)\right\rfloor + 1 - E_{\text{min}}, \tag{3.3}$$

we will have $F$ upper-bounded by $F \leqslant F_{\text{max}} \leqslant E_{\text{min}}$. Assuming that $E_{\text{min}} \leqslant 0$, which is the case for all binary IEEE754-2008 formats [117], this implies that $n$, expressed as follows, is indeed an integer for all choices of $E$, $m$ and $F$:

$$\begin{aligned} n &= 10^{-F} \cdot 2^E \cdot m \\ &= 10^{-F} \cdot 2^{E_{\text{min}}} \cdot 2^{E-E_{\text{min}}} \cdot m \\ &= 10^{-F+E_{\text{min}}} \cdot 5^{-E_{\text{min}}} \cdot 2^{E-E_{\text{min}}} \cdot m. \end{aligned} \tag{3.4}$$

In the last Eq. (3.4), we have $-F + E_{\text{min}} \geqslant 0$, hence $10^{-F+E_{\text{min}}} \in \mathbb{N}$, $-E_{\text{min}} \geqslant 0$, hence $5^{-E_{\text{min}}} \in \mathbb{N}$, $E - E_{\text{min}} \geqslant 0$, hence $2^{E-E_{\text{min}}} \in \mathbb{N}$, and, of course $m \in \mathbb{N}$. Therefore $n \in \mathbb{N}$ and the binary to decimal conversion, provided that $m$ and $E$ are bounded as indicated and the decimal precision $l$ is at least as high as given by Eq. (3.3), is always exact.

We may now wonder how an exact binary floating-point to decimal floating-point conversion can help us with our decimal character sequence to binary floating-point conversion. We build our answer, the algorithm, on the following observations:

— The rounding boundaries, i.e. the points where a rounding abruptly changes from one floating-point value to the next one, for binary floating-point numbers of precision $k$, are the floating-point numbers of precision $k' = k + 1$ [165, 166].

— The exact binary to decimal conversion sketched above hence works not only for the floating-point numbers of IEEE754-2008 formats but also for their rounding boundaries. The values $k$, $E_{\text{min}}$ and $E_{\text{max}}$ just need adapting; they do not change but slightly.

— The exact binary to decimal conversion hence allows us to exactly find the decimal counterpart to a binary rounding boundary, which represents the decisive point where the rounding we are about to perform for a decimal to binary conversion changes.

**A Layered Approach**

This enables the following layered approach to decimal character sequence to binary floating-point conversion:

1. We read in a certain number $\widehat{l} \in \mathbb{N}$ of significant decimal digits as well as the decimal exponent $\widehat{F}$, adjusted to the position of the decimal point and such that we have a integer $\widehat{n} \in \mathbb{N}$ such that $10^{\widehat{l}-1} \leqslant \widehat{n} \leqslant 10^{\widehat{l}} - 1$ representing, together with $\widehat{F}$, as $10^{\widehat{F}} \cdot \widehat{n}$, an approximation with a relative error of up to $1/2 \cdot 10^{-\widehat{l}+1}$ the exact input. For this step, we determine $\widehat{l}$ such that $1/2 \cdot 10^{-\widehat{l}+1}$ is less than the machine error of the binary format we intend to round to, plus some extra guard bits $g \geqslant 2$:
$$1/2 \cdot 10^{-\widehat{l}+1} \leqslant 2^{-k-g}.$$
This first step has of course a $\mathcal{O}(1)$ memory consumption and can be performed in $\mathcal{O}(t)$ time [6].
2. We convert the decimal approximation $10^{\widehat{F}} \cdot \widehat{n}$ to a binary approximation $2^{\widehat{E}} \cdot \widehat{m}$, $2^{\widehat{k}-1} \leqslant \widehat{m} \leqslant 2^{\widehat{k}} - 1$ with some precision $\widehat{k} \in \mathbb{N}$ chosen such that it represents the exact input –through the stack of the two approximations– with a relative error bounded by $2^{-k-g+2}$, where $k$ is still the precision of the format we intend to round to. This conversion step can be performed in a classical manner, determining first a candidate exponent $\widehat{E}$ through an approximation of $F \cdot \log_2(10)$, a table lookup of an approximation of $10^{-F+E}$, a small, fixed-size multiplication to get a candidate $\widehat{m}$ and some adjustments to get $2^{\widehat{k}-1} \leqslant \widehat{m} \leqslant 2^{\widehat{k}} - 1$. The time and space complexities of this step are $\mathcal{O}(1)$.
3. We perform a rounding test [66,197,264]: if we find no binary rounding boundary in the small interval $\left[2^{\widehat{E}} \cdot \widehat{m} \cdot \frac{1}{1+2^{-k-g+2}}; 2^{\widehat{E}} \cdot \widehat{m} \cdot \frac{1}{1+2^{-k-g+2}}\right]$ around $2^{\widehat{E}} \cdot \widehat{m}$, we are sure that we can round $2^{\widehat{E}} \cdot \widehat{m}$ to the same binary floating-point value $2^{E} \cdot m$ as if we rounded the exact input, given by the character sequence. In this case we return the correctly rounded value and stop the algorithm. If we do find a rounding boundary, call it $2^{\check{E}} \cdot \check{m}$, with $2^{k} \leqslant \check{m} \leqslant 2^{k+1} - 1$, we proceed to the next step. As a matter of course, this rounding test can be performed in $\mathcal{O}(1)$ time and space.
4. We exactly convert the rounding boundary $2^{\check{E}} \cdot \check{m}$, which satisfies $2^{k} \leqslant \check{m} \leqslant 2^{k+1} - 1$ and $\check{E}_{\text{min}} \leqslant \check{E} \leqslant \check{E}_{\text{max}}$ to a decimal equivalent $10^{\check{F}} \cdot \check{n} = 2^{\check{E}} \cdot \check{m}$. According to the Section above, with Eq. (3.3), we can predetermine a precision $l$ such that this conversion is exact. Hence, even though $l$ is pretty large –a couple of hundreds of digits for IEEE754-2008 binary64 [145]– the memory consumption, needed to store $\check{n}$ and all the intermediate values to determine it, is statically known. This means we obtain a $\mathcal{O}(1)$ time and space complexity also for this step [7].
5. We re-read [8] the decimal character sequence up to the $l$-th significant digit, forming a long integer $\widetilde{n}$ such that $10^{l-1} \leqslant \widetilde{n} \leqslant 10^{l} - 1$ and a corresponding decimal exponent

6. The exponent field being in the end of the decimal character string we need to keep going until we find that exponent field, which translates to a $\mathcal{O}(t)$ complexity instead of $\mathcal{O}(1)$. Interestingly, even if we use a naive, quadratic algorithm for the conversion of the $\widehat{l}$ characters to $\widehat{n}$, as $\widehat{l}$ is fixed, the time complexity stays $\mathcal{O}(t)$.

7. It's just that the constant hidden in the $\mathcal{O}$-notation is quite high.

8. The algorithm admits a variant for which the input is read only once, as is required for certain applications, e.g. when the input comes from a UNIX pipe file descriptor that can be read only once.

$\widetilde{F}$, adjusted for the position of the decimal point and the precision $l$. After the $l$-th significant digit, we continue reading the digits but only to compute a sticky bit $\sigma \in \{0, 1\}$: if all the digits after the $l$-th significant digit are zeros, we set $\sigma$ to $0$, if we encounter a non-zero digit, we set it to $1$[9]. In the end of this process we have an "approximation" $10^{\widetilde{F}} \cdot \widetilde{n}$ of the input, together with an indication if this approximation happens to be exact or if it is inexact. This step can be performed in $\mathcal{O}(t)$ time[10], where $t$ is the size of the input, and $\mathcal{O}(1)$ space, as the space required for $\widetilde{n}$ is bounded with $\widetilde{n} \leqslant 10^l - 1$.

6. We compare the reconverted rounding boundary $10^{\check{F}} \cdot \check{n}$ with the $l$-digit accurate representation of the input $10^{\widetilde{F}} \cdot \widetilde{n}$. This comparison takes $\mathcal{O}(1)$ time and space, as the size of $\check{n}$ and $\widetilde{n}$ is statically known. In the case when $10^{\check{F}} \cdot \check{n}$ is equal to $10^{\widetilde{F}} \cdot \widetilde{n}$, we replace the comparison test by what is indicated by the sticky bit $\sigma$. As a result of the comparison we exactly know whether the input is less than, equal to or greater than the rounding boundary $2^{\check{E}} \cdot \check{m}$. This allows us to exhibit a value $2^{\check{E}'} \cdot \check{m}'$, $2^{k+2} \leqslant \check{m}' \leqslant 2^{k+3} - 1$, that will round to the same IEEE754-2008 binary floating point number $2^E \cdot m$ as if the complete decimal input were converted with infinite precision and then rounded. The latter statement holds in any rounding mode; it is hence possible to make the algorithm rounding-mode agnostic.

The preceding algorithmic scheme can be transformed into an effective algorithm that performs decimal character sequence to binary floating-point conversion and that respects the design constraints we have given ourselves. Unfortunately, the precise –and quite gory– details of the different approximation and conversion steps, together with a presentation of the state machine that analyzes in decimal character sequence and of the checks for the corner conditions like complete underflow, gradual underflow, overflow or NaN handling, would go exhaust the space available in this document. The interested reader may consult [145] for an attempt to present them.

**Conclusion**

In this Section, we have presented an algorithmic approach to convert arbitrary length decimal character sequences to binary IEEE754-2008 floating-point numbers, performing correct rounding in any of the IEEE754-2008 binary rounding modes. The novelty of our algorithm is that has statically known memory consumption, hence requires no dynamic memory allocation, which alleviates any problems due to memory exhaustion in critical environments. The algorithm has a time complexity of $\mathcal{O}(t)$, where $t$ is the size of the decimal input.

The actual implementation of the algorithm is technically challenging and extremely tedious as an large amount of details need checking. Even though we have engaged in implementing the algorithm already, we still list a (formally) proven implementation in our list of things to do as future work. We would like to work hand in hand with specialists in formal proofs to come up with an implementation that is both water-tight correct and competitive with existing implementations of decimal character sequence conversions, which do result to dynamic memory allocation.

9. and we can stop reading

10. Once again, interestingly, even if we use a quadratic algorithm to accumulate the digits into $\widetilde{n}$, as $l$ is fixed, the time complexity stays $\mathcal{O}(t)$.

### 3.2.4 Support for IEEE754-2008: an attempt of a conclusion

In this Section we have seen that supporting the entirety of the binary part of IEEE754-2008 is perfectly feasible but tedious work: for the two formats binary32 and binary64 alone, IEEE754-2008 mandates the support for $354$ operations. Most of these operations are due to the combinatorial explosion that can be observed for the so-called heterogeneous operations, taking inputs larger than their outputs, rounding only once.

We have presented a way to implement these heterogeneous operations with no more than already existing IEEE754-1985 homogeneous operations. Our algorithm was based on Ziv's rounding test, a floating-point operation sequence initially developed for the implementation of elementary functions. The novelty of our algorithm for heterogeneous operations is that it is rounding-mode agnostic, i.e. it produces a correct rounding in any IEEE754-2008 binary rounding mode, without actually inquiring that rounding mode. While we have proven our heterogeneous operations algorithm on paper, we would like to work in the future with specialists in formal proof in order to develop an implementation that is fully proven formally.

We have also presented an algorithm to convert an arbitrarily long decimal character sequence to a correctly rounded binary floating-point number. The novelty of this algorithm is that it has statically known memory consumption and that it hence does not need dynamic memory allocation. Its execution time is $\mathcal{O}(t)$ and its space requirements are $\mathcal{O}(1)$, where $t$ is the size of the input. It is based on the observation that even if decimal to binary conversion is not generally exact, binary to decimal conversion can be made exact given a bounded binary exponent range and a sufficiently precise decimal format. The algorithm converts the decimal input to a preliminary binary representation and checks if correct rounding is possible. If not, it reconverts the binary rounding boundary back to decimal and performs a decimal conversion.

So has the last word about support for IEEE754-2008 been spoken? We think no. We would ideally engage in and follow up on the development of a support library for IEEE754-2008, similar to our `libieee754` library, but with the following enhancements:

— A support library for IEEE754-2008 needs to come in several flavors: a version for high performance computing, leveraging existing hardware at a maximum level, giving the use just that couple of operations they can directly address with a programming language like C, a version that does not at all depend on floating-point hardware and that works in every environment, even though its performance is not so great, a version that is formally proven, with respect to some formalization of the IEEE754 Standard, and that can serve a reference for the Standard, and perhaps a version that even supports non-default exception handling. These flavors should be selectable at compile time.

— Users will not make huge use of support libraries for standards like IEEE754-2008 if these libraries are hard to use or awkward to adapt to existing code. We would like to come up with a compiler environment that would allow users to compile existing code, enhanced with pragmas to express IEEE754 behavior that is hard to express in standard C, in a way that the floating-point operations are all mapped to a support library, which can then come in one of the flavors cited above.

What a challenge! We shall try to get funding and help from other people for the realization of such a library in the next years, though.

## 3.3 Enhancing IEEE754-2008: Mixed-Radix Comparisons

As we have seen in Sections 2.2 and 3.2, the 2008 revision of the IEEE754 Standard has enhanced the scope and, hence, number of features of the floating-point Standard in many ways: decimal arithmetic was added, the heterogeneous operations increased the "closedness" of a radix' floating-point formats and numbers representable with them. On the other hand, some of the additions to IEEE754-2008 may seem a little inconsequential: why do implementers of the Standard, who want (or are required) to support three floating-point formats, such as binary32, binary64 and binary128, have to support all $3^4 = 81$ possible combinations of operand and result formats of Fused-Multiply-And-Add, while the Standard specifies no way of even correctly comparing floating-point numbers in the binary radix with floating-point numbers in the decimal radix [117]? In order to propose solutions to possibly remove these inconsistencies in future revisions of the Standard, we worked at solving these issues.

In this Section, we hence propose our work on such mixed-radix comparisons. This work has been published in [28], which is itself an extended version of [27]. In the subsequent Section 3.4, we will go even one step further and start looking at mixed-radix computational operations.

### 3.3.1 Uniting the Disjoint: Linking the IEEE754 Radices with Comparisons

The IEEE 754-2008 Standard for Floating-Point Arithmetic [117] specifies binary (radix-2) and decimal (radix-10) floating-point number formats for a variety of precisions. Presented already in Section 3.1, the so-called "basic interchange formats" are described with more detail in Table 3.1.

The Standard neither requires nor forbids comparisons between floating-point numbers of different radices (it states that *floating-point data represented in different formats shall be comparable as long as the operands' formats have the same radix*). However, such "mixed-radix" comparisons may offer several advantages. It is not infrequent to read decimal data from a database and to have to compare it to some binary floating-point number. The comparison may be inaccurate if the decimal number is preliminarily converted to binary, or, respectively, if the binary number is first converted to decimal.

Consider for instance the following C code:

```
double x = ...;
_Decimal64 y = ...;
if (x <= y)
    ...
```

The standardization of decimal floating-point arithmetic in C [124] is still at draft stage, and compilers supporting decimal floating-point arithmetic handle code sequences such as the previous one at their discretion and often in an unsatisfactory way. As it occurs, Intel's icc 12.1.3 translates this sequence into a conversion from binary to decimal followed by a decimal comparison. The compiler emits no warning that the boolean result might not be the expected one because of rounding.

This kind of strategy may lead to inconsistencies. Consider such a "naive" approach built as follows: when comparing a binary floating-point number $x_2$ of format $\mathcal{F}_2$, and a decimal floating-point number $x_{10}$ of format $\mathcal{F}_{10}$, we first convert $x_{10}$ to the binary format $\mathcal{F}_2$ (that is,

| | binary32 | binary64 | binary128 |
|---|---|---|---|
| precision (bits) | 24 | 53 | 113 |
| $e_{\min}$ | $-126$ | $-1022$ | $-16382$ |
| $e_{\max}$ | $+127$ | $+1023$ | $+16383$ |

| | decimal64 | decimal128 |
|---|---|---|
| precision (digits) | 16 | 34 |
| $e_{\min}$ | $-383$ | $-6143$ |
| $e_{\max}$ | $+384$ | $+6144$ |

Table 3.1 – The basic binary and decimal interchange formats specified by the IEEE 754-2008 Standard.

we replace it by the $\mathcal{F}_2$ number nearest $x_{10}$), and then we perform the comparison in binary. Denote the comparison operators so defined as $\olessthan$, $\oleq$, $\ogreaterthan$, and $\ogeq$. Consider the following variables (all exactly represented in their respective formats):

— $x = 3602879701896397/2^{55}$, declared as a binary64 number;

— $y = 13421773/2^{27}$, declared as a binary32 number;

— $z = 1/10$, declared as a decimal64 number.

Then it holds that $x \olessthan y$, but also $y \oleq z$ and $z \oleq x$. Such an inconsistent result might for instance suffice to prevent a sorting program from terminating.

Remark that, in principle, $x_2$ and $x_{10}$ could both be converted to some longer format in such a way that the naive method yields a correct answer. However, that method requires a correctly rounded radix conversion, which is an intrinsically more expensive operation than the comparison itself.

An experienced programmer is likely to explicitly convert all variables to a larger format. However, we believe that ultimately the task of making sure that the comparisons are consistent and correctly performed should be left to the compiler, and that the only fully consistent way of comparing two numeric variables $x$ and $y$ of a different type is to effectively compare $x$ and $y$, and not to compare an approximation to $x$ to an approximation to $y$.

In what follows below, we describe algorithms to perform such "exact" comparisons between binary numbers in any of the basic binary interchange formats and decimal numbers in any of the basic decimal interchange formats. Our algorithms were developed with a software implementation in mind. They end up being pretty generic, but we make no claim as to their applicability to hardware implementations.

It is natural to require that mixed-radix comparisons not signal any floating-point exception, except in situations that parallel exceptional conditions specified in the Standard [117, Section 5.11] for regular comparisons. For this reason, we avoid floating-point operations that might signal exceptions, and mostly use integer arithmetic to implement our comparisons.

Our method is based on two steps:

— Algorithm 4 tries to compare numbers just by examining their floating-point exponents;

— when Algorithm 4 is inconclusive, Algorithm 5 provides the answer.

Algorithm 5 admits many variants and depends on a number of parameters. Tables 3.5 and 3.6 suggest suitable parameters for most typical use cases.

Implementing the comparison for a given pair of formats just requires the implementation of Algorithms 4 and 5. The analysis of the algorithms as a function of all parameters, as presented in Sections 3.3.4 to 3.3.6, is somewhat technical.

### 3.3.2 Mixed Radix Comparisons: Setting and Algorithmic Outline

We consider a binary format of precision $p_2$, minimum exponent $e_2^{\min}$ and maximum exponent $e_2^{\max}$, and a decimal format of precision $p_{10}$, minimum exponent $e_{10}^{\min}$ and maximum exponent $e_{10}^{\max}$. We want to compare a binary floating-point number $x_2$ and a decimal floating-point number $x_{10}$.

Without loss of generality we assume $x_2 > 0$ and $x_{10} > 0$ (when $x_2$ and $x_{10}$ are negative, the problem reduces to comparing $-x_2$ and $-x_{10}$, and, when they have different signs, the comparison is straightforward). The floating-point representations of $x_2$ and $x_{10}$ are

$$\begin{aligned} x_2 &= M_2 \cdot 2^{e_2-p_2+1}, \\ x_{10} &= M_{10} \cdot 10^{e_{10}-p_{10}+1}, \end{aligned}$$

where $M_2$, $M_{10}$, $e_2$ and $e_{10}$ are integers that satisfy:

$$\begin{aligned} e_2^{\min} - p_2 + 1 \leqslant e_2 &\leqslant e_2^{\max}, \\ e_{10}^{\min} \leqslant e_{10} &\leqslant e_{10}^{\max}, \\ 2^{p_2-1} \leqslant M_2 &\leqslant 2^{p_2} - 1, \\ 1 \leqslant M_{10} &\leqslant 10^{p_{10}} - 1 \end{aligned} \tag{3.5}$$

with $p_2, p_{10} \geqslant 1$. (The choice of lower bound for $e_2$ makes the condition $2^{p_2-1} \leqslant M_2$ hold even for subnormal binary numbers.)

We assume that the so-called *binary encoding* (BID) [117,197] of IEEE 754-2008 is used for the decimal format [11], so that the integer $M_{10}$ is easily accessible in binary. Denote by

$$p'_{10} = \lceil p_{10} \log_2 10 \rceil ,$$

the number of bits that are necessary for representing the decimal significands in binary.

When $x_2$ and $x_{10}$ have significantly different orders of magnitude, examining their exponents will suffice to compare them. Hence, we first perform a simple exponent-based test (Section 3.3.3). When this does not suffice, a second step compares the significands multiplied by suitable powers of 2 and 5. We compute "worst cases" that determine how accurate the second step needs to be (Section 3.3.4), and show how to implement it efficiently (Section 3.3.5). Section 3.3.6 is an aside describing a simpler algorithm that can be used if we only want to decide whether $x_2$ and $x_{10}$ are equal. Finally, Section 3.3.7 discusses our implementation of the comparison algorithms and presents experimental results.

11. This encoding is typically used in software implementations of decimal floating-point arithmetic, even if BID-based hardware designs have been proposed [244].

| | b32/d64 | b32/d128 | b64/d64 | b64/d128 | b128/d64 | b128/d128 |
|---|---|---|---|---|---|---|
| $p_2$ | 24 | 24 | 53 | 53 | 113 | 113 |
| $p_{10}$ | 16 | 34 | 16 | 34 | 16 | 34 |
| $p'_{10}$ | 54 | 113 | 54 | 113 | 54 | 113 |
| $w$ | 29 | 88 | 0 | 59 | −60 | −1 |
| $h^{(1)}_{\min}$ | −570 | −6371 | −1495 | −7296 | −16915 | −22716 |
| $h^{(1)}_{\max}$ | 526 | 6304 | 1422 | 7200 | 16782 | 22560 |
| $s_{\min}$ | 18 | 23 | 19 | 23 | 27 | 27 |
| datatype | `int32` | `int64` | `int32` | `int64` | `int64` | `int64` |

Table 3.2 – The various parameters involved in Step 1 of the comparison algorithm.

### 3.3.3 First step: eliminating the “simple cases” by examining the exponents

#### Normalization

As we have

$$1 \leqslant M_{10} \leqslant 10^{p_{10}} - 1,$$

there exists a unique $\nu \in \{0, 1, 2, \dots, p'_{10} - 1\}$ such that

$$2^{p'_{10}-1} \leqslant 2^{\nu} M_{10} \leqslant 2^{p'_{10}} - 1.$$

Our initial problem of comparing $x_2$ and $x_{10}$ reduces to comparing $M_2 \cdot 2^{e_2-p_2+1+\nu}$ and $(2^{\nu} M_{10}) \cdot 10^{e_{10}-p_{10}+1}$.

The fact that we “normalize” the decimal significand $M_{10}$ by a binary shift between two consecutive powers of two is of course questionable; $M_{10}$ could also be normalized into the range $10^{p_{10}-1} \leqslant 10^t \cdot M_{10} \leqslant 10^{p_{10}} - 1$. However, hardware support for this operation [244] is not widespread. A decimal normalization would thus require a loop, provoking pipeline stalls, whereas the proposed binary normalization can exploit an existing hardware leading-zero counter with straight-line code.

#### Comparing the Exponents

Define

$$\begin{cases} m = M_2, \\ h = e_2 - e_{10} + \nu + p_{10} - p'_{10} + 1, \\ n = M_{10} \cdot 2^{\nu}, \\ g = e_{10} - p_{10} + 1, \\ w = p'_{10} - p_2 - 1 \end{cases} \tag{3.6}$$

so that

$$\begin{cases} 2^{\nu} x_2 &= m \cdot 2^{h+g+w}, \\ 2^{\nu} x_{10} &= n \cdot 10^{g}. \end{cases}$$

Our comparison problem becomes:

*Compare $m \cdot 2^{h+w}$ with $n \cdot 5^g$.*

We have

$$\begin{aligned} m_{\min} = 2^{p_2-1} &\leqslant m \leqslant m_{\max} = 2^{p_2} - 1, \\ n_{\min} = 2^{p'_{10}-1} &\leqslant n \ \leqslant n_{\max} = 2^{p'_{10}} - 1. \end{aligned} \tag{3.7}$$

It is clear that $m_{\min} \cdot 2^{h+w} > n_{\max} \cdot 5^g$ implies $x_2 > x_{10}$, while $m_{\max} \cdot 2^{h+w} < n_{\min} \cdot 5^g$ implies $x_2 < x_{10}$. This gives

$$\begin{aligned} (1 - 2^{-p'_{10}}) \cdot 5^g < 2^{h-2} \quad &\Rightarrow \quad x_{10} < x_2, \\ (1 - 2^{-p_2}) \cdot 2^h < 5^g \quad &\Rightarrow \quad x_2 < x_{10}. \end{aligned} \tag{3.8}$$

In order to compare $x_2$ and $x_{10}$ based on these implications, define

$$\varphi(h) = \lfloor h \cdot \log_5 2 \rfloor.$$

**Proposition 3.3.1.** *We have*

$$\begin{aligned} g < \varphi(h) \Rightarrow x_2 > x_{10}, \\ g > \varphi(h) \Rightarrow x_2 < x_{10}. \end{aligned}$$

*Proof.* If $g < \varphi(h)$ then $g \leqslant \varphi(h) - 1$, hence $g \leqslant h \log_5 2 - 1$. This implies that $5^g \leqslant (1/5) \cdot 2^h < 2^h/(4 \cdot (1 - 2^{-p'_{10}}))$, therefore, from (3.8), $x_{10} < x_2$. If $g > \varphi(h)$ then $g \geqslant \varphi(h) + 1$, hence $g > h \log_5 2$, so that $5^g > 2^h > (1 - 2^{-p_2}) \cdot 2^h$. This implies, from (3.8), $x_2 < x_{10}$. □

Now consider the range of $h$: by (3.6), $h$ lies between

$$h^{(1)}_{\min} = (e_2^{\min} - p_2 + 1) - e_{10}^{\max} + p_{10} - p'_{10} + 1$$

and

$$h^{(1)}_{\max} = e_2^{\max} - e_{10}^{\min} + p_{10}.$$

That range is given in Table 3.2 for the basic IEEE formats. Knowing the range, it is easy to implement $\varphi$ as follows.

**Proposition 3.3.2.** *Denote by $\lfloor \cdot \rceil$ the nearest integer function. For large enough $s \in \mathbb{N}$, the function defined by*

$$\widehat{\varphi}(h) = \lfloor L \cdot h \cdot 2^{-s} \rfloor, \qquad \textit{with } L = \lfloor 2^s \log_5 2 \rceil,$$

*satisfies $\varphi(h) = \widehat{\varphi}(h)$ for all $h$ in the range $[h^{(1)}_{\min}, h^{(1)}_{\max}]$.*

Proposition 3.3.2 is an immediate consequence of the irrationality of $\log_5 2$. For known, moderate values of $h^{(1)}_{\min}$ and $h^{(1)}_{\max}$, the optimal choice $s_{\min}$ of $s$ is small and easy to find. For instance, if the binary format is binary64 and the decimal format is decimal64, then $s_{\min} = 19$.

Table 3.2 gives the value of $s_{\min}$ for the basic IEEE formats. The product $L \cdot h$, for $h$ in the indicated range and $s = s_{\min}$, can be computed exactly in (signed or unsigned) integer arithmetic, with the indicated data type. Computing $\lfloor \xi \cdot 2^{-\beta} \rfloor$ of course reduces to a right-shift by $\beta$ bits.

Propositions 3.3.1 and 3.3.2 yield to the following algorithm.

```
1 Compute $h = e_2 - e_{10} + \nu + p_{10} - p'_{10} + 1$ and $g = e_{10} - p_{10} + 1$;
2 With the appropriate value of $s$, compute $\varphi(h) = \lfloor L \cdot h \cdot 2^{-s} \rfloor$ using integers;
3 if $g < \varphi(h)$ then return "$x_2 > x_{10}$";
4 else if $g > \varphi(h)$ then return "$x_2 < x_{10}$";
5 else first step is inconclusive (perform the second step);
```

**Algorithm 4:** First, exponent-based step

Note that, when $x_{10}$ admits multiple distinct representations in the precision-$p_{10}$ decimal format (i.e., when its cohort is non-trivial [117]), the success of the first step may depend on the specific representation passed as input. For instance, assume that the binary and decimal formats are binary64 and decimal64, respectively. Both $A = \{M_{10} = 10^{15}, e_{10} = 0\}$ and $B = \{M_{10} = 1, e_{10} = 15\}$ are valid representations of the integer 1. Assume we are trying to compare $x_{10} = 1$ to $x_2 = 2$. Using representation $A$, we have $\nu = 4$, $h = -32$, and $\varphi(h) = -14 > g = -15$, hence the test from Algorithm 4 shows that $x_{10} < x_2$. In contrast, if $x_{10}$ is given in the form $B$, we get $\nu = 53$, $\varphi(h) = \varphi(2) = 0 = g$, and the test is inconclusive.

### How Often is the First Step Enough?

We may quantify the quality of this first filter as follows. We say that Algorithm 4 *succeeds* if it answers "$x_2 > x_{10}$" or "$x_2 < x_{10}$" without proceeding to the second step, and *fails* otherwise. Let $X_2$ and $X_{10}$ denote the sets of *representations* of positive, finite numbers in the binary, resp. decimal formats of interest. (In the case of $X_2$, each number has a single representation.) Assuming zeros, infinities, and NaNs have been handled before, the input of Algorithm 4 may be thought of as a pair $(\xi_2, \xi_{10}) \in X_2 \times X_{10}$.

**Proposition 3.3.3.** *The proportion of input pairs* $(\xi_2, \xi_{10}) \in X_2 \times X_{10}$ *for which Algorithm 4 fails is bounded by*

$$\min\left\{ \frac{1}{e_{10}^{\max} - e_{10}^{\min} + 1}, \frac{4}{e_2^{\max} - e_2^{\min} + 1} \right\},$$

*assuming* $p_2 \geqslant 3$.

*Proof.* The first step fails if and only if $\varphi(h) = g$. Write $h = e_2 + t$, that is, $t = p_{10} - p'_{10} + 1 - e_{10} + \nu$. Then $\varphi(h) = g$ rewrites as

$$\lfloor (e_2 + t) \log_5 2 \rfloor = g,$$

which implies

$$\begin{aligned} -t + g \log_2 5 \leqslant e_2 &< -t + (g+1) \log_2 5 \\ &< -t + g \log_2 5 + 2.4. \end{aligned}$$

The value of $\nu$ is determined by $M_{10}$, so that $t$ and $g = e_{10} - p_{10} + 1$ depend on $\xi_{10}$ only. Thus, for any given $\xi_{10} \in X_{10}$, there can be at most 3 values of $e_2$ for which $\varphi(h) = g$.

A similar argument shows that for given $e_2$ and $M_{10}$, there exist at most one value of $e_{10}$ such that $\varphi(h) = g$.

Let $X_2^{\mathrm{norm}}$ and $X_2^{\mathrm{sub}}$ be the subsets of $X_2$ consisting of normal and subnormal numbers respectively. Let $r_i = e_i^{\max} - e_i^{\min} + 1$. The number $N_{\mathrm{norm}}$ of pairs $(\xi_2, \xi_{10}) \in X_2^{\mathrm{norm}} \times X_{10}$ such that $\varphi(h) = g$ satisfies

$$\begin{aligned} N_{\mathrm{norm}} &\leqslant \#\{M_{10} : \xi_{10} \in X_{10}\} \cdot \#\{M_2 : \xi_2 \in X_2^{\mathrm{norm}}\} \\ &\quad \cdot \#\{(e_2, e_{10}) : \xi_2 \in X_2^{\mathrm{norm}} \wedge \varphi(h) = g\} \\ &\leqslant (10^{p_{10}} - 1) \cdot 2^{p_2 - 1} \cdot \min\{r_2, 3r_{10}\}. \end{aligned}$$

For subnormal $x_2$, we have the bound

$$\begin{aligned} N_{\mathrm{sub}} &:= \#\{(\xi_2, \xi_{10}) \in X_2^{\mathrm{sub}} \times X_{10} : \varphi(h) = g\} \\ &\leqslant \#\{M_{10}\} \cdot \#X_2^{\mathrm{sub}} \\ &= (10^{p_{10}} - 1) \cdot (2^{p_2 - 1} - 1). \end{aligned}$$

The total number of elements of $X_2 \times X_{10}$ for which the first step fails is bounded by $N_{\mathrm{norm}} + N_{\mathrm{sub}}$. This is to be compared with

$$\begin{aligned} \#X_2 &= r_2 \cdot 2^{p_2 - 1} + (p_2 - 1) \cdot (2^{p_2 - 1} - 1) \\ &\geqslant (r_2 + 1) \cdot 2^{p_2 - 1}, \\ \#X_{10} &= r_{10} \cdot (10^{p_{10}} - 1). \end{aligned}$$

We obtain

$$\frac{N_{\mathrm{norm}} + N_{\mathrm{sub}}}{\#(X_2 \times X_{10})} \leqslant \min\left\{\frac{1}{r_{10}}, \frac{4}{r_2}\right\}. \qquad \square$$

These are rough estimates. One way to get tighter bounds for specific formats is simply to count, for each value of $\nu$, the pairs $(e_2, e_{10})$ such that $\varphi(h) = g$. For instance, in the case of comparison between binary64 and decimal64 floating-point numbers, one can check that the failure rate in the sense of Proposition 3.3.3 is less than $0.1\%$.

As a matter of course, pairs $(\xi_2, \xi_{10})$ will almost never be equidistributed in practice. Hence the previous estimate should not be interpreted as a *probability* of success of Step 1. It seems more realistic to assume that a well-written numerical algorithm will mostly perform comparisons between numbers which are suspected to be close to each other. For instance, in an iterative algorithm where comparisons are used as part of a convergence test, it is to be expected that most comparisons need to proceed to the second step. Conversely, there are scenarios, e.g., checking for out-of-range data, where the first step should be enough.

### 3.3.4 Second step: a closer look at the significands

#### Problem Statement

In the following we assume that $g = \varphi(h)$, i.e.,

$$e_{10} - p_{10} + 1 = \left\lfloor (e_2 - e_{10} + \nu + p_{10} - p'_{10} + 1) \log_5(2) \right\rfloor.$$

(Otherwise, the first step already allowed us to compare $x_2$ and $x_{10}$.)

Define a function

$$f(h) = \frac{5^{\varphi(h)}}{2^{h+w}}.$$

We have

$$\begin{cases} f(h) \cdot n > m \Rightarrow x_{10} > x_2, \\ f(h) \cdot n < m \Rightarrow x_{10} < x_2, \\ f(h) \cdot n = m \Rightarrow x_{10} = x_2. \end{cases} \tag{3.9}$$

The second test consists in performing this comparison, with $f(h) \cdot n$ replaced by an accurate enough approximation.

In order to ensure that an approximate test is indeed equivalent to (3.9), we need a lower bound $\eta$ on the minimum nonzero value of

$$d_h(m, n) = \left| \frac{5^{\varphi(h)}}{2^{h+w}} - \frac{m}{n} \right| \tag{3.10}$$

that may appear at this stage. We want $\eta$ to be as tight as possible in order to avoid unduly costly computations when approximating $f(h) \cdot n$. The search space is constrained by the following observations.

**Proposition 3.3.4.** *Let $g$ and $h$ be defined by Equation (3.6). The equality $g = \varphi(h)$ implies*

$$\frac{e_2^{\min} - p_2 - p'_{10} + 3}{1 + \log_5 2} \leqslant h < \frac{e_2^{\max} + 2}{1 + \log_5 2}. \tag{3.11}$$

*Additionally, $n$ satisfies the following properties:*

1. *if $n \geqslant 10^{p_{10}}$, then $n$ is even;*
2. *if $\nu' = h + \varphi(h) - e_2^{\max} + p'_{10} - 2$ is nonnegative (which holds for large enough $h$), then $2^{\nu'}$ divides $n$.*

*Proof.* From (3.6), we get

$$e_2 = h + g + p'_{10} - \nu - 2.$$

Since $e_2^{\min} - p_2 + 1 \leqslant e_2 \leqslant e_2^{\max}$ and as we assumed $g = \varphi(h)$, it follows that

$$e_2^{\min} - p_2 + 1 \leqslant h + \varphi(h) - \nu + p'_{10} - 2 \leqslant e_2^{\max}.$$

Therefore we have

$$e_2^{\min} - p_2 + \nu - p'_{10} + 3 \leqslant (1 + \log_5 2)h < e_2^{\max} + \nu - p'_{10} + 3,$$

and the bounds (3.11) follow because $0 \leqslant \nu \leqslant p'_{10} - 1$.

The binary normalization of $M_{10}$, yielding $n$, implies that $2^\nu \mid n$. If $n \geqslant 10^{p_{10}}$, then, by (3.5) and (3.6), it follows that $\nu \geqslant 1$ and $n$ is even. As $h + \varphi(h) - \nu \leqslant e_2^{\max} - p'_{10} + 2$, we also have $\nu \geqslant \nu'$. Since $\varphi$ is increasing, there exists $h_0$ such that $\nu' \geqslant 0$ for $h \geqslant h_0$. □

Table 3.3 gives the range (3.11) for $h$ with respect to the comparisons between basic IEEE formats.

| | b32/d64 | b32/d128 | b64/d64 | b64/d128 | b128/d64 | b128/d128 |
|---|---|---|---|---|---|---|
| $h^{(2)}_{\min}, h^{(2)}_{\max}$ | $-140, 90$ | $-181, 90$ | $-787, 716$ | $-828, 716$ | $-11565, 11452$ | $-11606, 11452$ |
| $g^{(2)}_{\min}, g^{(2)}_{\max}$ | $-61, 38$ | $-78, 38$ | $-339, 308$ | $-357, 308$ | $-4981, 4932$ | $-4999, 4932$ |
| $\#g$ | 100 | 117 | 648 | 666 | 9914 | 9932 |
| $h_0$ | 54 | 12 | 680 | 639 | 11416 | 11375 |

Table 3.3 – Range of $h$ for which we may have $g = \varphi(h)$, and size of the table used in the "direct method" of Section 3.3.5.

#### Required Worst-Case Accuracy

Let us now deal with the problem of computing $\eta$, considering $d_h(m, n)$ under the constraints given by Equation (3.7) and Proposition 3.3.4. A similar problem was considered by Cornea et al. [50] in the context of correctly rounded binary to decimal conversions. Their techniques yield worst and bad cases for the approximation problem we consider. We will take advantage of them in Section 3.3.7 to test our algorithms on many cases when $x_2$ and $x_{10}$ are very close. Here, we favor a different approach that is less computationally intensive and mathematically simpler, making the results easier to check either manually or with a proof-assistant.

**Problem 1.** *Find the smallest nonzero value of*

$$d_h(m, n) = \left| \frac{5^{\varphi(h)}}{2^{h+w}} - \frac{m}{n} \right|$$

*subject to the constraints*

$$\begin{cases} 2^{p_2-1} \leqslant m \leqslant 2^{p_2} - 1, \\ 2^{p'_{10}-1} \leqslant n \leqslant 2^{p'_{10}} - 1, \\ h^{(2)}_{\min} \leqslant h \leqslant h^{(2)}_{\max}, \\ n \textit{ is even if } n \geqslant 10^{p_{10}}, \\ \textit{if } h \geqslant h_0, \textit{ then } 2^{\nu'} \mid n \end{cases}$$

*where*

$$\begin{aligned} h^{(2)}_{\min} &= \left\lceil \frac{e_2^{\min} - p_2 - p'_{10} + 3}{1 + \log_5 2} \right\rceil, \\ h^{(2)}_{\max} &= \left\lfloor \frac{e_2^{\max} + 2}{1 + \log_5 2} \right\rfloor, \\ h_0 &= \left\lceil \frac{e_2^{\max} - p'_{10} + 3}{1 + \log_5 2} \right\rceil, \\ \nu' &= h + \varphi(h) - e_2^{\max} + p'_{10} - 2. \end{aligned}$$

We recall how such a problem can be solved using the classical theory of continued fractions [108, 135, 208].

Given $\alpha \in \mathbb{Q}$, build two finite sequences $(a_i)_{0 \leqslant i \leqslant n}$ and $(r_i)_{0 \leqslant i \leqslant n}$ by setting $r_0 = \alpha$ and

$$\begin{cases} a_i = \lfloor r_i \rfloor, \\ r_{i+1} = 1/(r_i - a_i) \text{ if } a_i \neq r_i. \end{cases}$$

For all $0 \leqslant i \leqslant n$, the rational number

$$\frac{p_i}{q_i} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{\ddots + \cfrac{1}{a_i}}}$$

is called the $i$th *convergent* of the *continued fraction expansion* of $\alpha$. Observe that the process defining the $a_i$ essentially is the Euclidean algorithm. If we assume $p_{-1} = 1$, $q_{-1} = 0$, $p_0 = a_0$, $q_0 = 1$, we have $p_{i+1} = a_{i+1}p_i + p_{i-1}$, $q_{i+1} = a_{i+1}q_i + q_{i-1}$, and $\alpha = p_n/q_n$.

It is a classical result [135, Theorem 19] that any rational number $m/n$ such that

$$\left| \alpha - \frac{m}{n} \right| < \frac{1}{2n^2}$$

is a convergent of the continued fraction expansion of $\alpha$. For basic IEEE formats, it turns out that this classical result is enough to solve Problem 1, as cases when it is not precise enough can be handled in an *ad-hoc* way.

First consider the case $\max(0, -w) \leqslant h \leqslant p_2$. We can then write

$$d_h(m, n) = \frac{|5^{\varphi(h)}n - 2^{h+w}m|}{2^{h+w}n}$$

where the numerator and denominator are both integers. If $d_h(m, n) \neq 0$, we have $|5^{\varphi(h)}n - m2^{h+w}| \geqslant 1$, hence $d_h(m, n) \geqslant 1/(2^{h+w}n) > 2^{p_2 - h - 2p'_{10} + 1} \geqslant 2^{-2p'_{10}+1}$. For the range of $h$ where $d_h(m, n)$ might be zero, the *non-zero* minimum is thus no less than $2^{-2p'_{10}+1}$.

If now $p_2 + 1 \leqslant h \leqslant h^{(2)}_{\max}$, then $h + w \geqslant p'_{10}$, so $5^{\varphi(h)}$ and $2^{h+w}$ are integers, and $m2^{h+w}$ is divisible by $2^{p'_{10}}$. As $n \leqslant 2^{p'_{10}} - 1$, we have $d_h(m, n) \neq 0$. To start with, assume that $d_h(m, n) \leqslant 2^{-2p'_{10}-1}$. Then, the integers $m$ and $n \leqslant 2^{p'_{10}} - 1$ satisfy

$$\left| \frac{5^{\varphi(h)}}{2^{h+w}} - \frac{m}{n} \right| \leqslant 2^{-2p'_{10}-1} < \frac{1}{2n^2},$$

and hence $m/n$ is a convergent of the continued fraction expansion of $\alpha = 5^{\varphi(h)}/2^{h+w}$.

We compute the convergents with denominators less than $2^{p'_{10}}$ and take the minimum of the $|\alpha - p_i/q_i|$ for which there exists $k \in \mathbb{N}$ such that $m = kp_i$ and $n = kq_i$ satisfy the constraints of Problem 1. Notice that this strategy only yields the actual minimum nonzero of $d_h(m, n)$ if we eventually find, for some $h$, a convergent with $|\alpha - p_i/q_i| < 2^{-2p'_{10}-1}$. Otherwise, taking $\eta = 2^{-2p'_{10}-1}$ yields a valid (yet not tight) lower bound.

The case $h^{(2)}_{\min} \leqslant h \leqslant \min(0, -w)$ is similar. A numerator of $d_h(m, n)$ then is $|2^{-h-w}n - 5^{-\varphi(h)}m|$ and a denominator is $5^{-\varphi(h)}n$. We notice that $5^{-\varphi(h)} \geqslant 2^{p'_{10}}$ if and only if $h \leqslant -p'_{10}$. If $-p'_{10} + 1 \leqslant h \leqslant \min(0, -w)$ and $d_h(m, n) \neq 0$, we have $|2^{-h}n - 5^{-\varphi(h)}m| \geqslant 1$, hence $d_h(m, n) \geqslant 1/(5^{-\varphi(h)}n) \geqslant 2^{-2p'_{10}}$. For $h^{(2)}_{\min} \leqslant h \leqslant -p'_{10}$, we use the same continued fraction tools as above.

| | $h$ | $m$ | $n$ | $-\log_2 \eta$ |
|---|---|---|---|---|
| b32/d64 | 50 | 10888194 | 13802425659501406 | 111.40 |
| b32/d128 | −159 | 11386091 | 8169119658476861812680212016502305 | 229.57 |
| b64/d64 | −275 | 4988915232824583 | 12364820988483254 | 113.68 |
| b64/d128 | −818 | 5148744585188163 | 9254355313724266263661769079234135 | 233.58 |
| b128/d64 | 2546 | 7116022508838657793249305056613439 | 13857400902051554 | 126.77 |
| b128/d128 | 10378 | 7977485665655127446147737154136553 | 9844227914381600512882010261817769 | 237.14 |

Table 3.4 – Worst cases for $d_h(m,n)$ and corresponding lower bounds $\eta$.

There remains the case $\min(0,-w) \leqslant h \leqslant \max(0,-w)$. If $0 \leqslant h \leqslant -w$, a numerator of $d_h(m,n)$ is $|5^{\varphi(h)}2^{-h-w}n - m|$ and a denominator is $n$. Hence, if $d_h(m,n) \neq 0$, we have $d_h(m,n) \geqslant 1/n > 2^{p'_{10}}$. If $-w \leqslant h \leqslant 0$, a numerator of $d_h(m,n)$ is $|n - 5^{-\varphi(h)}2^{h+w}m|$ and a denominator is $5^{-\varphi(h)}2^{h+w}n \leqslant 5 \cdot 2^{-h}2^{h+w}n < 5 \cdot 2^{2p'_{10}-p_2-1}$. It follows that if $d_h(m,n) \neq 0$, we have $d_h(m,n) \geqslant 1/5 \cdot 2^{-2p'_{10}+p_2+1}$.

**Proposition 3.3.5.** *The minimum nonzero values of $d_h(m,n)$ under the constraints from Problem 1 for the basic IEEE formats are attained for the parameters given in Table 3.4.*

*Proof.* This follows from applying the algorithm outlined above to the parameters associated with each basic IEEE format. Scripts to verify the results are provided along with our example implementation of the comparison algorithm (see Section 3.3.7). □

### 3.3.5 Inequality testing

As already mentioned, the first step of the full comparison algorithm is straightforwardly implementable in 32-bit or 64-bit integer arithmetic. The second step reduces to comparing $m$ and $f(h) \cdot n$, and we have seen in Section 3.3.4 that it is enough to know $f(h)$ with a relative accuracy given by the last column of Table 3.4. We now discuss several ways to perform that comparison. The main difficulty is to efficiently evaluate $f(h) \cdot n$ with just enough accuracy.

#### Direct Method

A direct implementation of Equation (3.9) just replaces $f(h)$ with a sufficiently accurate approximation, read in a table indexed with $h$. The actual accuracy requirements can be stated as follows. Recall that $\eta$ denotes a (tight) lower bound on (3.10).

**Proposition 3.3.6.** *Assume that $\mu$ approximates $f(h) \cdot n$ with relative accuracy $\chi < \eta/2^{-w+2}$ or better, that is,*

$$|f(h) \cdot n - \mu| \leqslant \chi f(h) \cdot n < \frac{\eta}{2^{-w+2}} f(h) \cdot n. \tag{3.12}$$

*The following implications hold:*

$$\begin{cases} \mu > m + \chi \cdot 2^{p_2+1} \implies x_{10} > x_2, \\ \mu < m - \chi \cdot 2^{p_2+1} \implies x_{10} < x_2, \\ |m - \mu| \leqslant \chi \cdot 2^{p_2+1} \implies x_{10} = x_2. \end{cases} \tag{3.13}$$

*Proof.* First notice that $f(h) \leqslant 2^{-w} = 2^{p_2 - p'_{10} + 1}$ for all $h$. Since $n < 2^{p'_{10}}$, condition (3.12) implies $|\mu - f(h) \cdot n| < 2^{p_2+1}\chi$, and hence $|f(h) \cdot n - m| > |\mu - m| - 2^{p_2+1}\chi$.

Now consider each of the possible cases that appear in (3.13). If $|\mu - m| > 2^{p_2+1}\chi$, then $f(h) \cdot n - m$ and $\mu - m$ have the same sign. The first two implications from (3.9) then translate into the corresponding cases from (3.13).

If finally $|\mu - m| \leqslant 2^{p_2+1}\chi$, then the triangle inequality yields $|f(h) \cdot n - m| \leqslant 2^{p_2+2}\chi$, which implies $|f(h) - m/n| < 2^{p_2+2}\chi/n < 2^{p_2+w}\eta/n \leqslant \eta$. By definition of $\eta$, this cannot happen unless $m/n = f(h)$. This accounts for the last case. □

For instance, in the case of binary64–decimal64 comparisons, it is more than enough to compute the product $f(h) \cdot n$ in 128-bit arithmetic.

Proposition 3.3.4 gives the number of values of $h$ to consider in the table of $f(h)$, which grows quite large (23.5 kB for b64/d64 comparisons, 721 kB in the b128/d128 case, assuming a memory alignment on 8-byte boundaries). However, it could possibly be shared with other algorithms such as binary-decimal conversions. Additionally, $h$ is available early in the algorithm flow, so that the memory latency for the table access can be hidden behind the first step.

The number of elements in the table can be reduced at the price of a few additional operations. Several consecutive $h$ map to a same value $g = \varphi(h)$. The corresponding values of $f(h) = 2^{\varphi(h) \cdot \log_2 5 - h}$ are binary shifts of each other. Hence, we may store the most significant bits of $f(h)$ as a function of $g$, and shift the value read off the table by an appropriate amount to recover $f(h)$. The table size goes down by a factor of about $\log_2(5) \approx 2.3$.

More precisely, define $F(g) = 5^g 2^{-\psi(g)}$, where

$$\psi(g) = \lfloor g \cdot \log_2 5 \rfloor. \tag{3.14}$$

We then have $f(h) = F(\varphi(h)) \cdot 2^{-\rho(h)}$ with $\rho(h) = h - \psi(\varphi(h))$. The computation of $\psi$ and $\rho$ is similar to that of $\varphi$ in Step 1. In addition, we may check that

$$1 \leqslant F(\varphi(h)) < 2 \quad \text{and} \quad 0 \leqslant \rho(h) \leqslant 3$$

for all $h$. Since $\rho(h)$ is nonnegative, multiplication by $2^{\rho(h)}$ can be implemented with a bitshift, not requiring branches.

We did not implement this size reduction: instead, we concentrated on another size reduction opportunity that we shall describe now.

### Bipartite Table

That second table size reduction method uses a bipartite table. Additionally, it takes advantage of the fact that, for small nonnegative $g$, the exact value of $5^g$ takes less space than an accurate enough approximation of $f(h)$ (for instance, $5^g$ fits on 64 bits for $g \leqslant 27$). In the case of a typical implementation of comparison between binary64 and decimal64 numbers, the bipartite table uses only 800 bytes of storage.

Most of the overhead in arithmetic operations for the bipartite table method can be hidden on current processors through increased instruction-level parallelism. The reduction in table size also helps decreasing the probability of cache misses.

Recall that we suppose $g = \varphi(h)$, so that $h$ lies between bounds $h^{(2)}_{\min}, h^{(2)}_{\max}$ deduced from Proposition 3.3.4. Corresponding bounds $g^{(2)}_{\min}, g^{(2)}_{\max}$ on $g$ follow since $\varphi$ is nondecreasing.

For some integer "splitting factor" $\gamma > 0$ and $\varepsilon = \pm 1$, write

$$g = \varphi(h) = \varepsilon(\gamma q - r), \qquad q = \left\lceil \frac{\varepsilon g}{\gamma} \right\rceil, \tag{3.15}$$

so that

$$f(h) = \left( \frac{5^{\gamma q}}{5^r} \right)^{\varepsilon} \cdot 2^{-h-w}.$$

Typically, $\gamma$ will be chosen as a power of 2, but other values can occasionally be useful. With these definitions, we have $0 \leqslant r \leqslant \gamma - 1$, and $q$ lies between

$$\left\lceil \varepsilon \frac{g^{(2)}_{\min}}{\gamma} \right\rceil \quad \text{and} \quad \left\lceil \varepsilon \frac{g^{(2)}_{\max}}{\gamma} \right\rceil$$

(the order of the bounds depends on $\varepsilon$). The integer $5^r$ fits on $\psi(r) + 1$ bits, where $\psi$ is the function defined by (3.14).

Instead of tabulating $f(h)$ directly, we will use two tables: one containing the most significant bits of $5^{\gamma q}$ roughly to the precision dictated by the worst cases computed in Section 3.3.4, and the other containing the exact value $5^r$ for $0 \leqslant r \leqslant \gamma - 1$. We denote by $\lambda_1$ and $\lambda_2$ the respective precisions (entry sizes in bits) of these tables. Based on these ranges and precisions, we assume

$$\lambda_1 > \log_2(\eta^{-1}) - w + 3, \tag{3.16}$$

$$\lambda_2 \geqslant \psi(\gamma - 1) + 1. \tag{3.17}$$

In addition, it is natural to suppose $\lambda_1$ larger than or close to $\lambda_2$: otherwise, an algorithm using two approximate tables will likely be more efficient.

In practice, $\lambda_2$ will typically be one of the available machine word widths, while, for reasons that should become clear later, $\lambda_1$ will be chosen slightly smaller than a word size. For each pair of basic IEEE formats, Tables 3.5 and 3.6 provide values of $\varepsilon, \gamma, \lambda_1, \lambda_2$ (and other parameters whose definition follows later) suitable for typical processors. The suggested values were chosen by optimizing either the table size or the number of elementary multiplications in the algorithm, with or without the constraint that $\gamma$ be a power of two.

It will prove convenient to store the table entries left-aligned, in a fashion similar to floating-point significands. Thus, we set

$$\theta_1(q) = 5^{\gamma q} \cdot 2^{\lambda_1 - 1 - \psi(\gamma q)},$$

$$\theta_2(r) = 5^{r} \ \cdot 2^{\lambda_2 - 1 - \psi(r)},$$

where the power-of-two factors provide for the desired alignment. One can check that

$$2^{\lambda_1 - 1} \leqslant \theta_1(q) < 2^{\lambda_1}, \quad 2^{\lambda_2 - 1} \leqslant \theta_2(r) < 2^{\lambda_2} \tag{3.18}$$

for all $h$.

The value $f(h)$ now decomposes as

$$f(h) = \frac{5^g}{2^{h+w}} = \left( \frac{\theta_1(q)}{\theta_2(r)} \right)^{\varepsilon} 2^{\varepsilon \cdot (\lambda_2 - \lambda_1) - \sigma(h) - w} \tag{3.19}$$

where

$$\sigma(h) = h - \varepsilon \cdot \big(\psi(\gamma q) - \psi(r)\big). \tag{3.20}$$

Equations (3.14) and (3.20) imply that

$$h - g \log_2 5 - 1 < \sigma(h) < h - g \log_2 5 + 1.$$

As we also have $h \log_5 2 - 1 \leqslant g = \varphi(h) < h \log_5 2$ by definition of $\varphi$, it follows that

$$0 \leqslant \sigma(h) \leqslant 3. \tag{3.21}$$

Now let $\tau \geqslant -1$ be an integer serving as an adjustment parameter to be chosen later. In practice, we will usually[12] choose $\tau = 0$ and sometimes $\tau = -1$ (see Tables 3.5 and 3.6). For most purposes the reader may simply assume $\tau = 0$. Define

$$\begin{cases} \Delta = \theta_1(q) \cdot n \cdot 2^{\tau - p'_{10}} \\ \qquad - \theta_2(r) \cdot m \cdot 2^{\tau + \lambda_1 - \lambda_2 + \sigma(h) - p_2 - 1}, & \varepsilon = +1, \\ \Delta = \theta_1(q) \cdot m \cdot 2^{\tau - p_2 - 1} \\ \qquad - \theta_2(r) \cdot n \cdot 2^{\tau + \lambda_1 - \lambda_2 - \sigma(h) - p'_{10}}, & \varepsilon = -1. \end{cases}$$

The relations $|x_2| > |x_{10}|$, $|x_2| = |x_{10}|$, and $|x_2| < |x_{10}|$ hold respectively when $\varepsilon\Delta < 0$, $\Delta = 0$, and $\varepsilon\Delta > 0$.

**Proposition 3.3.7.** *Unless $x_2 = x_{10}$, we have $|\Delta| > 2^{\tau+1}$.*

*Proof.* Assume $x_2 \neq x_{10}$. For $\varepsilon = +1$, and using (3.19), the definition of $\Delta$ rewrites as

$$\Delta = \theta_2(r)\, n\, 2^{\tau + \lambda_1 - \lambda_2 + \sigma(h) - p_2 - 1} \left( f(h) - \frac{m}{n} \right).$$

Since $f(h) = m/n$ is equivalent to $x_2 = x_{10}$, we know by Proposition 3.3.5 that $|f(h) - m/n| \geqslant \eta$. Together with the bounds (3.7), (3.16), (3.18), and (3.21), this implies

$$\begin{aligned} \log_2 |\Delta| &\geqslant (\lambda_2 - 1) + (p'_{10} - 1) + \tau + \lambda_1 - \lambda_2 - p_2 - 1 \\ &\quad + \log_2 \eta \\ &= (\lambda_1 + \log_2 \eta + w - 3) + \tau + 1 > \tau + 1. \end{aligned}$$

Similarly, for $\varepsilon = -1$, we have

$$\Delta = -\theta_1(q)\, n\, 2^{\tau - p_2 - 1} \left( f(h) - \frac{m}{n} \right),$$

so that

$$\begin{aligned} \log_2 |\Delta| &\geqslant (\lambda_1 - 1) + (p'_{10} - 1) + \tau - p_2 - 1 + \log_2 \eta \\ &> \tau + 1 \end{aligned}$$

when $\Delta \neq 0$. □

12. See the comments following Equations (3.23) to (3.25). Yet allowing for positive values of $\tau$ enlarges the possible choices of parameters to satisfy (3.25) (resp. (3.25′)) for highly unusual floating-point formats and implementation environments. For instance, it makes it possible to store the table for $\theta_1$ in a more compact fashion, using a different word size for the table than for the computation.

As already explained, the values of $\theta_2(r)$ are integers and can be tabulated exactly. In contrast, only an approximation of $\theta_1(q)$ can be stored. We represent these values as $\lceil\theta_1(q)\rceil$, hence replacing $\Delta$ by the easy-to-compute

$$\begin{cases} \widetilde{\Delta} = \big\lfloor \lceil\theta_1(q)\rceil \cdot n \cdot 2^{\tau - p_{10}'} \big\rfloor \\ \qquad - \big\lfloor \theta_2(r) \cdot m \cdot 2^{\tau+\lambda_1-\lambda_2+\sigma(h)-p_2-1} \big\rfloor, & \varepsilon = +1, \\ \widetilde{\Delta} = \big\lfloor \lceil\theta_1(q)\rceil \cdot m \cdot 2^{\tau - p_2 - 1} \big\rfloor \\ \qquad - \big\lfloor \theta_2(r) \cdot n \cdot 2^{\tau+\lambda_1-\lambda_2-\sigma(h)-p_{10}'} \big\rfloor, & \varepsilon = -1. \end{cases}$$

Proposition 3.3.7 is in principle enough to compare $x_2$ with $x_{10}$, using a criterion similar to that from Proposition 3.3.6. Yet, additional properties hold that allow for a more efficient final decision step.

**Proposition 3.3.8.** *Assume $g = \varphi(h)$, and let $\widetilde{\Delta}$ be the signed integer defined above. For $\tau \geqslant 0$, the following equivalences hold:*

$$\begin{aligned} \Delta < 0 &\iff & \widetilde{\Delta} &\leqslant -2^{\tau}, \\ \Delta = 0 &\iff & 0 \leqslant \widetilde{\Delta} &\leqslant \ \ 2^{\tau}, \\ \Delta > 0 &\iff & \widetilde{\Delta} &\geqslant \ \ 2^{\tau+1}. \end{aligned}$$

*Furthermore, if $\tau \in \{-1, 0\}$ and*

$$\lambda_1 - \lambda_2 + \tau \geqslant \begin{cases} p_2 + 1, & \varepsilon = +1, \\ p_{10}' + 3, & \varepsilon = -1, \end{cases} \tag{3.22}$$

*then $\Delta$ and $\widetilde{\Delta}$ have the same sign (in particular, $\widetilde{\Delta} = 0$ if and only if $\Delta = 0$).*

*Proof.* Write the definition of $\widetilde{\Delta}$ as

$$\widetilde{\Delta} = \big\lfloor \lceil\theta_1(q)\rceil X \big\rfloor - \big\lfloor \theta_2(r) X' \big\rfloor$$

where the expressions of $X$ and $X'$ depend on $\varepsilon$, and define error terms $\delta_{\text{tbl}}, \delta_{\text{rnd}}, \delta_{\text{rnd}}'$ by

$$\delta_{\text{tbl}} = \lceil\theta_1(q)\rceil - \theta_1(q), \qquad \begin{aligned} \delta_{\text{rnd}} &= \lfloor \lceil\theta_1(q)\rceil X \rfloor - \lceil\theta_1(q)\rceil X, \\ \delta_{\text{rnd}}' &= \lfloor \theta_2(r) X' \rfloor - \theta_2(r) X'. \end{aligned}$$

The $\delta$'s then satisfy

$$0 \leqslant \delta_{\text{tbl}} < 1, \qquad -1 < \delta_{\text{rnd}}, \delta_{\text{rnd}}' \leqslant 0.$$

For both possible values of $\varepsilon$, we have [13] $0 \leqslant X \leqslant 2^{\tau}$ and hence

$$-1 < \widetilde{\Delta} - \Delta = X\delta_{\text{tbl}} + \delta_{\text{rnd}} - \delta_{\text{rnd}}' < 2^{\tau} + 1.$$

If $\Delta < 0$, Proposition 3.3.7 implies that

$$\widetilde{\Delta} < -2^{\tau+1} + 2^{\tau} + 1 = -2^{\tau} + 1.$$

13. Actually $0 \leqslant X \leqslant 2^{\tau-1}$ for $\varepsilon = -1$: this dissymmetry is related to our choice of always expressing the lower bound on $\Delta$ in terms of $\eta$ rather than using a lower bound on $f(h)^{-1} - n/m$ when $\varepsilon = -1$.

It follows that $\widetilde{\Delta} \leqslant -\lfloor 2^{\tau} \rfloor$ since $\widetilde{\Delta}$ is an integer. By the same reasoning, $\Delta > 0$ implies $\widetilde{\Delta} \geqslant \lfloor 2^{\tau+1} \rfloor$, and $\Delta = 0$ implies $0 \leqslant \widetilde{\Delta} \leqslant \lceil 2^{\tau} \rceil$. This proves the first claim.

Now assume that (3.22) holds. In this case, we have $\delta'_{\mathrm{rnd}} = 0$, so that $-1 < \widetilde{\Delta} - \Delta < 2^{\tau}$. The upper bounds on $\widetilde{\Delta}$ become $\widetilde{\Delta} \leqslant -\lfloor 2^{\tau} \rfloor - 1$ for $\Delta < 0$ and $\widetilde{\Delta} \leqslant \lceil 2^{\tau} \rceil - 1$ for $\Delta = 0$. When $-1 \leqslant \tau \leqslant 0$, these estimates respectively imply $\widetilde{\Delta} < 0$ and $\widetilde{\Delta} = 0$, while $\widetilde{\Delta} \geqslant \lfloor 2^{\tau+1} \rfloor$ implies $\Delta > 0$. The second claim follows. □

The conclusion $\widetilde{\Delta} = 0 = \Delta$ when $x_2 = x_{10}$ for certain parameter choices means that there is no error in the approximate computation of $\Delta$ in these cases. Specifically, the tabulation error $\delta_{\mathrm{tbl}}$ and the rounding error $\delta_{\mathrm{rnd}}$ cancel out, thanks to our choice to tabulate the ceiling of $\theta_1(q)$ (and not, say, the floor).

We now describe in some detail the algorithm resulting from Proposition 3.3.8, in the case $\varepsilon = +1$. In particular, we will determine integer datatypes on which all steps can be performed without overflow, and arrange the powers of two in the expression of $\widetilde{\Delta}$ in such a way that computing the floor functions comes down to dropping the least significant words of multiple-word integers.

Let $W : \mathbb{N} \to \mathbb{N}$ denote a function satisfying $W(k) \geqslant k$ for all $k$. In practice, $W(k)$ will typically be the width of the smallest "machine word" that holds at least $k$ bits. We further require that (for $\varepsilon = +1$):

$$W(p'_{10}) - p'_{10} \geqslant 1, \tag{3.23}$$
$$W(\lambda_1) - \lambda_1 \geqslant W(\lambda_2) - \lambda_2 + \tau + 3, \tag{3.24}$$
$$W(p_2) - p_2 + W(\lambda_2) - \lambda_2 \geqslant W(\lambda_1) - \lambda_1 - \tau + 1. \tag{3.25}$$

These assumptions are all very mild in view of the IEEE-754-2008 Standard. Indeed, for all IEEE interchange formats (including binary16 and formats wider than 64 bits), the space taken up by the exponent and sign leaves us with at least 5 bits of headroom if we choose for $W(p_2)$, resp. $W(p'_{10})$ the word width of the format. Even if we force $W(\lambda_2) = \lambda_2$ and $\tau = 0$, this always allows for a choice of $\lambda_1$, $\lambda_2$ and $W(\lambda_2)$ satisfying (3.16), (3.17) and (3.23)–(3.25).

**Proposition 3.3.9.** *When either $\tau \geqslant 0$ or $\tau = -1$ and (3.22) holds, Algorithm 5 correctly decides the ordering between $x_2$ and $x_{10}$.*

*Proof.* Hypotheses (3.23) and (3.25), combined with the inequalities $\sigma(h) \geqslant 0$ and $\tau \geqslant -1$, imply that the shifts in Steps 4 and 5 are left shifts. They can be performed without overflow on unsigned words of respective widths $W(\lambda_1)$, $W(p'_{10})$ and $W(p_2)$ since $W(\lambda_1) - \lambda_1 \geqslant \tau + 1$ and $W(p_2) - p_2 \geqslant \omega + 3$, both by (3.24). The same inequality implies that the (positive) integers $A$ and $B$ both fit on $W(\lambda_1) - 1$ bits, so that their difference can be computed by a subtraction on $W(\lambda_1)$ bits. The quantity $A - B$ computed in Step 6 agrees with the previous definition of $\widetilde{\Delta}$, and $\Delta'$ has the same sign as $\widetilde{\Delta}$, hence Proposition 3.3.8 implies that the relation returned in Step 8 is correct. □

Denote $x^+ = \max(x, 0)$ and $x^- = \max(-x, 0)$, so that $x = x^+ - x^-$.

```
1 Compute $q$ and $r$ as defined in (3.15), with $\varepsilon = +1$;
2 Compute $\sigma(h)$ using (3.20) and Proposition 3.3.2;
3 Read $\lceil \theta_1(q) \rceil$ from the table, storing it on a $W(\lambda_1)$-bit (multiple-)word;
4 Compute
      $$A = \left\lfloor (\lceil \theta_1(q) \rceil \cdot 2^{\tau^+}) \cdot (n 2^{W(p'_{10}) - p'_{10} - \tau^-}) \cdot 2^{-W(p'_{10})} \right\rfloor$$
  by (constant) left shifts followed by a $W(\lambda_1)$-by-$W(p'_{10})$-bit multiplication, dropping
  the least significant word(s) of the result that correspond to $W(p'_{10})$ bits;
5 Compute
      $$B = \left\lfloor \theta_2(r) \cdot (m 2^{\sigma(h)+\omega}) \cdot 2^{W(\lambda_1) - W(\lambda_2) - W(p_2)} \right\rfloor$$
  where the constant $\omega$ is given by
      $$\omega = W(p_2) - p_2 + W(\lambda_2) - \lambda_2 - W(\lambda_1) + \lambda_1 + \tau - 1$$
  in a similar way, using a $W(\lambda_2)$-by-$W(p_2)$-bit multiplication;
6 Compute the difference $\widetilde{\Delta} = A - B$ as a $W(\lambda_1)$-bit signed integer;
7 Compute $\Delta' = \lfloor \widetilde{\Delta} 2^{-\tau-1} \rfloor 2^{\tau+1}$ by masking the $\tau + 1$ least significant bits of $\widetilde{\Delta}$;
8 **return**
      $$\begin{cases} \text{“}x_{10} < x_2\text{”} & \textit{if } \Delta' < 0, \\ \text{“}x_{10} = x_2\text{”} & \textit{if } \Delta' = 0, \\ \text{“}x_{10} > x_2\text{”} & \textit{if } \Delta' > 0. \end{cases}$$
```

**Algorithm 5:** Step 2, second method, $\varepsilon = +1$

When $\tau \in \{-1, 0\}$ and (3.22) holds, no low-order bits are dropped in Step 5, and Step 7 can be omitted (that is, replaced by $\Delta' := \widetilde{\Delta}$). Observe additionally that a version of Step 8 returning $-1$, 0 or 1 according to the comparison result can be implemented without branching, using bitwise operations on the individual words which make up the representation of $\Delta'$ in two's complement encoding.

The algorithm for $\varepsilon = -1$ is completely similar, except that (3.23)–(3.25) become

$$W(p_2) - p_2 \geqslant 2, \tag{3.23′}$$

$$W(\lambda_1) - \lambda_1 \geqslant W(\lambda_2) - \lambda_2 + \tau + 1, \tag{3.24′}$$

$$W(p'_{10}) - p'_{10} + W(\lambda_2) - \lambda_2 \geqslant W(\lambda_1) - \lambda_1 - \tau + 3, \tag{3.25′}$$

and $A$ and $B$ are computed as

$$A = \left\lfloor (\lceil \theta_1(q) \rceil \cdot 2^{\tau^+}) \cdot (m 2^{W(p_2) - p_2 - \tau^- - 1}) \cdot 2^{-W(p_2)} \right\rfloor$$
$$B = \left\lfloor \theta_2(r) \cdot (n 2^{\omega - \sigma(h)}) \cdot 2^{W(\lambda_1) - W(\lambda_2) - W(p'_{10})} \right\rfloor,$$

with

$$\omega = W(p'_{10}) - p'_{10} + W(\lambda_2) - \lambda_2 - W(\lambda_1) + \lambda_1 + \tau,$$

respectively using a $W(\lambda_1)$-by-$W(p_2)$-bit and a $W(\lambda_2)$-by-$W(p'_{10})$ multiplication.

Tables 3.5 and 3.6 suggest parameter choices for comparisons in basic IEEE formats. (We only claim that these are reasonable choices that satisfy all conditions, not that they are optimal for any well-defined metric.) Observe that, though it has the same overall structure, the algorithm presented in [27] is not strictly speaking a special case of Algorithm 5, as the definition of the bipartite table (cf. (3.15)) is slightly different.

### 3.3.6 An Equality Test

Besides deciding the two possible inequalities, the direct and bipartite methods described in the previous sections are able to determine cases when $x_2$ is *equal* to $x_{10}$. However, when it comes to solely determine such equality cases additional properties lead to a simpler and faster algorithm.

The condition $x_2 = x_{10}$ is equivalent to

$$m \cdot 2^{h+w} = n \cdot 5^g, \tag{3.26}$$

and thus implies certain divisibility relations between $m$, $n$, $5^{\pm g}$ and $2^{\pm h \pm w}$. We now describe an algorithm that takes advantage of these relations to decide (3.26) using only binary shifts and multiplications on no more than $\max(p'_{10} + 2, p_2 + 1)$ bits.

**Proposition 3.3.10.** *Assume $x_2 = x_{10}$. Then, we have*

$$g_{\min}^{\text{eq}} \leqslant g \leqslant g_{\max}^{\text{eq}}, \qquad -p'_{10} + 1 \leqslant h \leqslant p_2.$$

*where*

$$g_{\min}^{\text{eq}} = -\lfloor p_{10}(1 + \log_5 2) \rfloor, \qquad g_{\max}^{\text{eq}} = \lfloor p_2 \log_5 2 \rfloor.$$

*Additionally,*

— *if $g \geqslant 0$, then $5^g$ divides $m$, otherwise $5^{-g}$ divides $M_{10}$ (and $n$);*

— *if $h \geqslant -w$, then $2^{h+w}$ divides $n$, otherwise $2^{-(h+w)}$ divides $m$.*

*Proof.* First, observe that $g$ must be equal to $\varphi(h)$ by Proposition 3.3.1, so $g \geqslant 0$ if and only if $h \geqslant 0$. The divisibility properties follow immediately from the coprimality of 2 and 5.

When $g \geqslant 0$, they imply $5^g \leqslant m < 2^{p_2} - 1$, whence $g < p_2 \log_5 2$. Additionally, $2^{-h-w} n$ is an integer. Since $h \geqslant 0$, it follows that $2^h \leqslant 2^{-w} n < 2^{p'_{10} - w}$ and hence

$$h \leqslant p'_{10} - w - 1 = p_2.$$

If now $g < 0$, then $5^{-g}$ divides $m = 2^\nu M_{10}$ and hence divides $M_{10}$, while $2^{-h}$ divides $2^w m$. Since $M_{10} < 10^{p_{10}}$ and $2^w m < 2^{p_2 + w}$, we have $-g < p_{10} \log_5 10$ and $-h < p_{10} \log_2 10 < p'_{10}$. □

These properties translate into Algorithm 6 below. Recall that $x^+ = \max(x, 0)$ and $x^- = x^+ - x$. (Branchless algorithms exist to compute $x^+$ and $x^-$ [10].) Let $p = \max(p'_{10} + 2, p_2 + 1)$.

As a matter of course, the algorithm can return false as soon as any of the conditions from Step 8 is known unsatisfied.

**Proposition 3.3.11.** *Algorithm 6 correctly decides whether $x_2 = x_{10}$.*

```
1 if $\neg(g^{\text{eq}}_{\min} \leqslant g \leqslant g^{\text{eq}}_{\max})$ then return "$x_2 \neq x_{10}$";
2 Set $u = (w+h)^-$ and $v = (w+h)^+$;
3 Using right shifts, compute
      $m_1 = \lfloor 2^{-u} m \rfloor, \qquad n_1 = \lfloor 2^{-v} n \rfloor;$
4 Using left shifts, compute $m_2 = 2^u m_1$ and $n_2 = 2^v n_1$;
5 Read $5^{g^+}$ and $5^{g^-}$ from a table (or compute them);
6 Compute $m_3 = 5^{g^-} m_1$ with a $p'_{10} \times p_2 \to p$ bit multiplication ;
7 Compute $n_3 = 5^{g^+} n_1$ with a $p_2 \times p'_{10} \to p$ bit multiplication;
8 if $(m_2 = m) \wedge (n_2 = n) \wedge (g = \varphi(h)) \wedge (n_3 = m_3)$ then return "$x_2 == x_{10}$" ;
9 return "$x_2 \neq x_{10}$";
```

**Algorithm 6:** Equality test.

*Proof.* First observe that, if any of the conditions $g = \varphi(h)$, $g^{\text{eq}}_{\min} \leqslant g \leqslant g^{\text{eq}}_{\max}$, $m_2 = m$ and $n_2 = n$ is violated, then $x_2 \neq x_{10}$ by Propositions 3.3.1 and 3.3.10. Indeed, $x_2 = x_{10}$ implies $m_2 = m$ and $n_2 = n$ due to the divisibility conditions of Proposition 3.3.10. In all these cases, the algorithm correctly returns false. In particular, no table access occurs unless $g$ lies between $g^{\text{eq}}_{\min}$ and $g^{\text{eq}}_{\max}$.

Now assume that these four conditions hold. In particular, we have $m_1 = 2^{-u}m$ as $m_2 = m$, and similarly $n_1 = 2^{-v}n$. The bounds on $g$ imply that $5^{g^+}$ fits on $p_2$ bits and $5^{g^-}$ fits on $p'_{10}$ bits. If $g$ and $w+h$ are both nonnegative, then $m_3 = m$, and we have

$$5^{g^+} n_1 = 5^{\varphi(h)} 2^{-(w+h)} n < 2^{-w+p'_{10}} = 2^{p_2+1} \leqslant 2^p.$$

By the same reasoning, if $g, w+h \leqslant 0$, then $n_3 = n$ and

$$5^{-g} 2^{w+h} m < 5 \cdot 2^{p'_{10}-1} < 2^p.$$

If now $g \geqslant 0$ with $h \leqslant -w$, then $m_3 = m_1$ and

$$5^g n < 2^{h+p'_{10}} \leqslant 2^{-w+p'_{10}} = 2^{p_2+1} \leqslant 2^p.$$

Similarly, for $g \leqslant 0$, $-h \leqslant w$, we have

$$5^{-g} m < 5 \cdot 2^{-h+p_2} \leqslant 5 \cdot 2^{w+p_2} \leqslant 2^p.$$

In all four cases,

$$m_3 = 5^{g^-} 2^{-u} m \quad \text{and} \quad n_3 = 5^{g^+} 2^{-v} n$$

are computed without overflow. Therefore

$$\frac{m_3}{n_3} = \frac{m \cdot 2^{(h+w)^+ - (h+w)^-}}{n \cdot 5^{g^+ - g^-}} = \frac{m \cdot 2^{h+w}}{n \cdot 5^g}$$

and $x_2 = x_{10}$ if and only if $m_3 = n_3$, by (3.26). □

| | b32/d64 | b32/d128 | b64/d64 | b64/d128 | b128/d64 | b128/d128 |
|---|---|---|---|---|---|---|
| $W(p_2)$ | 32 | 32 | 64 | 64 | 128 | 128 |
| $W(p'_{10})$ | 64 | 128 | 64 | 128 | 64 | 128 |
| $\gamma = 2^k$, optimizing multiplication count | | | | | | |
| $\varepsilon$ | $-1$ | $-1$ | $+1$ | $-1$ | $+1$ | $+1$ |
| $\gamma$ | 8 | 8 | 8 | 8 | 8 | 8 |
| range of $q$ | $-4, 8$ | $-4, 10$ | $-42, 39$ | $-38, 45$ | $-622, 617$ | $-624, 617$ |
| min $\lambda_1$ satisfying (3.16) | 86 | 145 | 117 | 178 | 190 | 242 |
| min $\lambda_2$ satisfying (3.17) | 17 | 17 | 17 | 17 | 17 | 17 |
| chosen $\lambda_1, W(\lambda_1)$ | $95, 96$ | $159, 160$ | $125, 128$ | $191, 192$ | $190, 192$ | $253, 256$ |
| chosen $\lambda_2, W(\lambda_2)$ | $32, 32$ | $32, 32$ | $32, 32$ | $32, 32$ | $32, 32$ | $32, 32$ |
| $\tau$ | 0 | 0 | 0 | 0 | $-1$ | 0 |
| Step 7 of Algo. 5 can be omitted | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| total table size in bytes | 188 | 332 | 1344 | 2048 | 29792 | 39776 |
| 32-bit muls | **5** | **9** | **10** | **16** | **16** | **36** |
| $\gamma = 2^k$, optimizing table size | | | | | | |
| $\varepsilon$ | $-1$ | $-1$ | $+1$ | $-1$ | $-1$ | $+1$ |
| $\gamma$ | 8 | 16 | 32 | 32 | 64 | 64 |
| range of $q$ | $-4, 8$ | $-2, 5$ | $-10, 10$ | $-9, 12$ | $-77, 78$ | $-78, 78$ |
| min $\lambda_1$ satisfying (3.16) | 86 | 145 | 117 | 178 | 190 | 242 |
| min $\lambda_2$ satisfying (3.17) | 17 | 35 | 72 | 72 | 147 | 147 |
| chosen $\lambda_1, W(\lambda_1)$ | $95, 96$ | $159, 160$ | $125, 128$ | $191, 192$ | $191, 192$ | $253, 256$ |
| chosen $\lambda_2, W(\lambda_2)$ | $32, 32$ | $64, 64$ | $96, 96$ | $96, 96$ | $160, 160$ | $160, 160$ |
| $\tau$ | 0 | 0 | 0 | 0 | 0 | 0 |
| Step 7 of Algo. 5 can be omitted | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ |
| total table size in bytes | **188** | **288** | **720** | **912** | **5024** | **6304** |
| 32-bit muls | 5 | 13 | 14 | 24 | 34 | 52 |
| any $\gamma$, optimizing multiplication count | | | | | | |
| $\varepsilon$ | $-1$ | $-1$ | $+1$ | $-1$ | $+1$ | $+1$ |
| $\gamma$ | 13 | 13 | 14 | 14 | 14 | 14 |
| range of $q$ | $-2, 5$ | $-2, 6$ | $-24, 22$ | $-22, 26$ | $-355, 353$ | $-357, 353$ |
| min $\lambda_1$ satisfying (3.16) | 86 | 145 | 117 | 178 | 190 | 242 |
| min $\lambda_2$ satisfying (3.17) | 28 | 28 | 31 | 31 | 31 | 31 |
| chosen $\lambda_1, W(\lambda_1)$ | $95, 96$ | $159, 160$ | $125, 128$ | $191, 192$ | $190, 192$ | $253, 256$ |
| chosen $\lambda_2, W(\lambda_2)$ | $32, 32$ | $32, 32$ | $32, 32$ | $32, 32$ | $32, 32$ | $32, 32$ |
| $\tau$ | 0 | 0 | 0 | 0 | $-1$ | 0 |
| Step 7 of Algo. 5 can be omitted | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| total table size in bytes | 148 | 232 | 808 | 1232 | 17072 | 22808 |
| 32-bit muls | **5** | **9** | **10** | **16** | **16** | **36** |
| any $\gamma$, optimizing table size | | | | | | |
| $\varepsilon$ | $-1$ | $-1$ | $+1$ | $+1$ | $-1$ | $+1$ |
| $\gamma$ | 13 | 13 | 28 | 28 | 69 | 83 |
| range of $q$ | $-2, 5$ | $-2, 6$ | $-12, 11$ | $-12, 11$ | $-71, 73$ | $-60, 60$ |
| min $\lambda_1$ satisfying (3.16) | 86 | 145 | 117 | 178 | 190 | 242 |
| min $\lambda_2$ satisfying (3.17) | 28 | 28 | 63 | 63 | 158 | 191 |
| chosen $\lambda_1, W(\lambda_1)$ | $95, 96$ | $159, 160$ | $125, 128$ | $189, 192$ | $191, 192$ | $253, 256$ |
| chosen $\lambda_2, W(\lambda_2)$ | $32, 32$ | $32, 32$ | $64, 64$ | $64, 64$ | $160, 160$ | $192, 192$ |
| $\tau$ | 0 | 0 | 0 | 0 | 0 | 0 |
| Step 7 of Algo. 5 can be omitted | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| total table size in bytes | **148** | **232** | **608** | **800** | **4860** | **5864** |
| 32-bit muls | 5 | 9 | 12 | 28 | 34 | 56 |

Table 3.5 – Suggested parameters for Algorithm 5 on a 32-bit machine.

| | b32/d64 | b32/d128 | b64/d64 | b64/d128 | b128/d64 | b128/d128 |
|---|---|---|---|---|---|---|
| $W(p_2)$ | 64 | 64 | 64 | 64 | 128 | 128 |
| $W(p'_{10})$ | 64 | 128 | 64 | 128 | 64 | 128 |
| $\gamma = 2^k$, optimizing multiplication count | | | | | | |
| $\varepsilon$ | $+1$ | $-1$ | $+1$ | $-1$ | $+1$ | $+1$ |
| $\gamma$ | 16 | 16 | 16 | 16 | 16 | 16 |
| range of $q$ | $-3, 3$ | $-2, 5$ | $-21, 20$ | $-19, 23$ | $-311, 309$ | $-312, 309$ |
| min $\lambda_1$ satisfying (3.16) | 86 | 145 | 117 | 178 | 190 | 242 |
| min $\lambda_2$ satisfying (3.17) | 35 | 35 | 35 | 35 | 35 | 35 |
| chosen $\lambda_1, W(\lambda_1)$ | $125, 128$ | $191, 192$ | $125, 128$ | $191, 192$ | $190, 192$ | $253, 256$ |
| chosen $\lambda_2, W(\lambda_2)$ | $64, 64$ | $64, 64$ | $64, 64$ | $64, 64$ | $64, 64$ | $64, 64$ |
| $\tau$ | 0 | 0 | 0 | 0 | $-1$ | 0 |
| Step 7 of Algo. 5 can be omitted | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| total table size in bytes | 240 | 320 | 800 | 1160 | 15032 | 20032 |
| 64-bit muls | **3** | **5** | **3** | **5** | **5** | **10** |
| $\gamma = 2^k$, optimizing table size | | | | | | |
| $\varepsilon$ | $+1$ | $-1$ | $+1$ | $-1$ | $-1$ | $+1$ |
| $\gamma$ | 16 | 16 | 16 | 32 | 64 | 64 |
| range of $q$ | $-3, 3$ | $-2, 5$ | $-21, 20$ | $-9, 12$ | $-77, 78$ | $-78, 78$ |
| min $\lambda_1$ satisfying (3.16) | 86 | 145 | 117 | 178 | 190 | 242 |
| min $\lambda_2$ satisfying (3.17) | 35 | 35 | 35 | 72 | 147 | 147 |
| chosen $\lambda_1, W(\lambda_1)$ | $125, 128$ | $191, 192$ | $125, 128$ | $191, 192$ | $191, 192$ | $253, 256$ |
| chosen $\lambda_2, W(\lambda_2)$ | $64, 64$ | $64, 64$ | $64, 64$ | $128, 128$ | $192, 192$ | $192, 192$ |
| $\tau$ | 0 | 0 | 0 | 0 | 0 | 0 |
| Step 7 of Algo. 5 can be omitted | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ |
| total table size in bytes | **240** | **320** | **800** | **1040** | **5280** | **6560** |
| 64-bit muls | 3 | 5 | 3 | 7 | 9 | 14 |
| any $\gamma$, optimizing multiplication count | | | | | | |
| $\varepsilon$ | $+1$ | $-1$ | $+1$ | $-1$ | $+1$ | $+1$ |
| $\gamma$ | 13 | 20 | 28 | 28 | 28 | 28 |
| range of $q$ | $-4, 3$ | $-1, 4$ | $-12, 11$ | $-11, 13$ | $-177, 177$ | $-178, 177$ |
| min $\lambda_1$ satisfying (3.16) | 86 | 145 | 117 | 178 | 190 | 242 |
| min $\lambda_2$ satisfying (3.17) | 28 | 45 | 63 | 63 | 63 | 63 |
| chosen $\lambda_1, W(\lambda_1)$ | $125, 128$ | $191, 192$ | $125, 128$ | $191, 192$ | $190, 192$ | $253, 256$ |
| chosen $\lambda_2, W(\lambda_2)$ | $64, 64$ | $64, 64$ | $64, 64$ | $64, 64$ | $64, 64$ | $64, 64$ |
| $\tau$ | 0 | 0 | 0 | 0 | $-1$ | 0 |
| Step 7 of Algo. 5 can be omitted | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| total table size in bytes | 232 | 304 | 608 | 824 | 8744 | 11616 |
| 64-bit muls | **3** | **5** | **3** | **5** | **5** | **10** |
| any $\gamma$, optimizing table size | | | | | | |
| $\varepsilon$ | $+1$ | $-1$ | $+1$ | $+1$ | $-1$ | $+1$ |
| $\gamma$ | 13 | 20 | 28 | 28 | 82 | 83 |
| range of $q$ | $-4, 3$ | $-1, 4$ | $-12, 11$ | $-12, 11$ | $-60, 61$ | $-60, 60$ |
| min $\lambda_1$ satisfying (3.16) | 86 | 145 | 117 | 178 | 190 | 242 |
| min $\lambda_2$ satisfying (3.17) | 28 | 45 | 63 | 63 | 189 | 191 |
| chosen $\lambda_1, W(\lambda_1)$ | $125, 128$ | $191, 192$ | $125, 128$ | $189, 192$ | $191, 192$ | $253, 256$ |
| chosen $\lambda_2, W(\lambda_2)$ | $64, 64$ | $64, 64$ | $64, 64$ | $64, 64$ | $192, 192$ | $192, 192$ |
| $\tau$ | 0 | 0 | 0 | 0 | 0 | 0 |
| Step 7 of Algo. 5 can be omitted | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ |
| total table size in bytes | **232** | **304** | **608** | **800** | **4896** | **5864** |
| 64-bit muls | 3 | 5 | 3 | 7 | 9 | 14 |

Table 3.6 – Suggested parameters for Algorithm 5 on a 64-bit machine.

| | Naive method converting $x_{10}$ to binary64 *(incorrect)* min/avg/max | Naive method converting $x_2$ to decimal64 *(incorrect)* min/avg/max | **Direct method** (Section 3.3.5) min/avg/max | **Bipartite table method (manual implementation)** (Section 3.3.5) min/avg/max |
|---|---|---|---|---|
| Special cases ($\pm 0$, NaNs, Inf) | 18/–/164 | 21/–/178 | 12/–/82 | 11/–/74 |
| $x_2$, $x_{10}$ of opposite sign | 30/107/188 | 44/124/179 | 15/30/76 | 16/31/73 |
| $x_2$ normal, same sign, "easy" cases | 31/107/184 | 48/132/180 | 25/57/118 | 21/43/95 |
| $x_2$ subnormal, same sign, "easy" cases | 45/106/178 | 89/119/179 | 153/163/189 | 19/27/73 |
| $x_2$ normal, same sign, "hard" cases | 60/87/186 | 39/78/180 | 45/56/139 | 29/40/119 |
| $x_2$ subnormal, same sign, "hard" cases | 79/106/167 | 78/107/179 | 171/177/189 | 34/49/94 |

Table 3.7 – Timings (in cycles) for binary64–decimal64 comparisons.

### 3.3.7 Experimental results

For each combination of a binary basic IEEE format and a decimal one, we implemented the comparison algorithm consisting of the first step (Section 3.3.3) combined with the version of the second step based on a bipartite table (Section 3.3.5). We used the parameters suggested in the second part of Table 3.6, that is for the case when $\gamma$ is a power of two and one tries to reduce table size.

The implementation was aided by scripts that take as input the parameters from Tables 3.2 and 3.6 and produce C code with GNU extensions [241]. Non-standard extensions are used for 128-bit integer arithmetic and other basic integer operations. No further effort at optimization was made.

Our implementations, the code generator itself, as well as various scripts to select the parameters are available at

`http://hal.archives-ouvertes.fr/hal-01021928/.`

The code generator can easily be adapted to produce other variants of the algorithm.

We tested the implementations on random inputs generated as follows. For each decimal exponent and each quantum [117, 2.1.42], we generate a few random significands and round the resulting decimal numbers upwards and downwards to the considered binary format. (We exclude the decimal exponents that merely yield to underflow or overflow on the binary side.)

Table 3.8 reports the observed timings for our random test data. The generated code was executed on a system equipped with a quad-core Intel Core i5-3320M processor clocked at 2.6 GHz, running Linux 3.16 in 64 bit mode. We used the `gcc` 4.9.2 compiler, at optimization

| | b32 | b64 | b128 |
|---|---|---|---|
| d64 | 41 | 36 | 75 |
| d128 | 235 | 266 | 61 |

Table 3.8 – Average run time (in cycles) for the bipartite table method, using the parameters suggested in Table 3.6 for $\gamma = 2^k$, optimizing table size.

level `-O3`, with the `-march=native` flag. The measurements were performed using the Read Time Stamp Counter instruction (RDTSC) after pipeline serialization. The serialization and function call overhead was estimated by timing an empty function and subtracted off. The caches were preheated by additional comparison function calls, the results of which were discarded.

It can be observed that "balanced" comparisons (d64/b64, d128/b128) perform significantly better than "unbalanced" ones. A quick look at the generated assembly code suggests that there is room for performance improvements, especially in cases involving 128-bit floating point formats.

We performed more detailed experiments in the special case of binary64–decimal64 comparisons. In complement to the generated code, we wrote a version of the bipartite table method with some manual optimization, and implemented the method based on direct tabulation of $f(h)$ presented in Section 3.3.5. We tested these implementations thoroughly with test vectors that extensively cover all floating-point input classes (normal numbers, subnormals, Not-A-Numbers, zeros, and infinities) and exercise all possible values of the normalization parameter $\nu$ (cf. Section 3.3.3) as well as all possible outcomes for both of the algorithm's steps.

Finally, we compared them for performance to the naive comparison method where one of the inputs is converted to the radix of the other input. These experimental results are reported in Table 3.7. In the last two rows, "easy" cases are cases when the first step of our algorithm succeeds, while "hard" cases refer to those for which running the second step is necessary.

### 3.3.8 Conclusion

Though not foreseen in the IEEE 754-2008 Standard, exact comparisons between floating-point formats of different radices would enrich the current floating-point environment and make numerical software safer and easier to prove.

In this Section, we have investigated the feasibility of such comparisons. A simple test has been presented that eliminates most of the comparison inputs. For the remaining cases, two algorithms were proposed, a direct method and a technique based on a more compact bipartite table. For instance, in the case of binary64–decimal64, the bipartite table uses only 800 bytes of table space.

Both methods have been proven, implemented and thoroughly tested. According to our experiments, they outperform the naive comparison technique consisting in conversion of one of the inputs to the respectively other format. Furthermore, they always return a correct answer, which is not the case of the naive technique.

Since the algorithmic problems of exact binary to decimal comparison and correctly rounded radix conversion are related, future investigations should also consider the possible reuse of tables for both problems. Finally, one should mention that considerable additional work is required in order to enable mixed-radix comparisons in the case when the decimal floating-point number is stored in dense-packed-decimal representation.

## 3.4 Towards Mixed-Radix Computational Operations

In the last Section 3.3, we have presented a first way to link the disjoint worlds of binary and decimal floating-point arithmetic in IEEE754-2008: we presented algorithms to compare binary floating-point numbers with numbers in the decimal radix without endangering the boolean result by the oddities of rounded radix conversion. While this first enhancement is valuable, it does not seem to be enough: we would like to see the closedness of the IEEE754-2008 environment for all formats and radices, under all computational operations.

In this Section, we address exactly this question: is it possible to design algorithms for the basic computational operations, such as addition, multiplication or division, in a mixed-radix setting, i.e. with the operands and results as IEEE754-2008 floating-point numbers of both radices, at the user's choice. As a matter of course, this work must be guided by the goal of achieving correct rounding for such mixed-radix operations: as we explained in Section 3.1, the spirit of IEEE754 lies in this correct-rounding goal.

This Section is partly based, for the part in Section 3.4.2, on joint work that was eventually published [145] by Olga Kupriianova, a Ph.D. student we supervised, and, for the parts in Sections 3.4.1 and 3.4.3, on work published in the article [125].

### 3.4.1 Mixed-Radix Fused-Multiply-Add: Motivation and Challenges

The basic computational operations defined by IEEE754 are addition, subtraction, multiplication, division, square root, and, since 2008, Fused-Multiply-And-Add (FMA) [117]. This latter operation computes $a \times b + c$ with one, single, correct rounding. We wish to provide algorithms for a mixed-radix implementation of these computational operations, i.e. algorithms that take floating-point numbers of both radices –two and ten– supported by the IEEE754-2008 Standard [117] and compute a correctly rounded floating-point result, also in one of these two radices. These mixed-radix arithmetical operations do hence not correspond to one operator only but to quite a number of different implementations: for example, a mixed-radix addition comes (at least) in the variants *binary plus binary gives decimal, binary plus decimal gives binary, binary plus decimal gives decimal* and *decimal plus decimal gives binary*. This number of different mixed-radix operation flavors is still increased by the fact that IEEE754-2008 defines different formats for each radix, such as binary32, binary64, decimal64 or decimal128 [117]. In principle, even mixed-radix heterogeneous operations, like a *binary64 plus decimal128 gives binary32*-addition, might eventually be considered.

To get started, it is however easier to consider, for the time being, only mixed-radix operations with operands and results of two IEEE754 floating-point formats, one binary and one decimal. In order not to be mislead to corner-case effects, the two formats should provide comparable accuracy, i.e. have precisions that are comparable provided appropriate conversion. For instance, as detailed in Section 3.1, the IEEE754 binary64 format provides $k = 53$ bits of binary precision, which is equivalent to a basic relative error of $u = 2^{-53}$. The IEEE754 decimal format provides $k = 16$ digits of decimal precision, which is equivalent to a basic relative error of $u = 1/2 \times 10^{-16} = 2^{-54.15\cdots}$. The two formats hence provide comparable accuracy. We therefore choose the IEEE754 binary64 and decimal64 formats for our work on mixed-radix operations.

We are hence interested in algorithms for the families of mixed-radix operations with binary64 and decimal64 operands and outputs, covering the basic operations addition, subtraction, multiplication, division, square root and FMA. We want our algorithms to provide correct rounding in any of the IEEE754-2008 rounding modes [117] that might be chosen for the radix our mixed-radix operation compute their output in [14]. We also want our mixed-

14. As explained in Section 3.1, IEEE754-2008 defines several rounding modes and models the rounding-mode

radix operations to behave like any other IEEE754 computational operation and to signal the appropriate IEEE754 floating-point exceptions [117], which result in the setting of the appropriate IEEE754 flags[15] under default exception handling[16]. For the reasons set out already in Section 3.3.1, we therefore consider mixed-radix operations that end up being implemented by "emulation" using integer arithmetic. Similarly, as in Section 3.3.1, we will suppose that the IEEE754 decimal formats, such as the decimal64 format we use, are encoded in the IEEE754 binary encoding (BID), i.e. that access to a binary representation of the decimal significand is straightforward [117].

Given this setting, let us now consider where the foremost challenges of mixed-radix floating-point operations lie. Let us admit the following notations:

— Binary operands are denoted $2^E \cdot m$, where $E \in \mathbb{Z}$ is a binary exponent in some domain determined by the used format and $m \in \mathbb{Z}$ is a signed integer significand satisfying $2^{k-1} \leqslant |m| \leqslant 2^k - 1$, where $k$ is the precision, for instance $k = 53$ for binary64.

— Decimal operands are denoted $10^F \cdot n$, where $F \in \mathbb{Z}$ is a decimal exponent in some domain defined by the used format and $n \in \mathbb{Z}$ is a signed integer significand satisfying $1 \leqslant |n| \leqslant 10^l - 1$ where $l$ is the precision, for instance $l = 16$ for decimal64.

— The mixed-radix operation computes either a binary output $2^G \cdot p$, with $G \in \mathbb{Z}$ appropriately bounded and $p \in \mathbb{Z}$ a signed integer significand bounded as above, or a decimal output $10^H \cdot q$, with $H \in \mathbb{Z}$ bounded and $q \in \mathbb{Z}$ a signed integer significand bounded as before.

For most problems in computer arithmetic, addition is the most easy operation to consider. As we are going to see just below, for mixed-radix operations, it is not: multiplication is actually the most easy operation to implement. As a matter of fact, computing a correctly rounded result $2^G \cdot p$ resp. $10^H \cdot q$ to a mixed-radix multiplication actually means determining the two integers $G$ and $p$ (resp. $H$ and $q$) such that $2^G \cdot p$ is closest (in round-to-nearest mode) to $2^{E+F} \cdot 5^F \cdot m \cdot n$. Assuming that evaluating $G$ is not difficult, correctly rounding the mixed-radix multiplication therefore means determining $p$ closest to $2^{E+F-G} \cdot 5^F \cdot m \cdot n$.

Correctly determining which value of an integer $p$ is closest to $2^{E+F-G} \cdot 5^F \cdot m \cdot n$ requires correctly determining whether $2^{E+F-G} \cdot 5^F \cdot m \cdot n$ is above or below the mid-point between the integers around $2^{E+F-G} \cdot 5^F \cdot m \cdot n$. Similarly, correctly determining the largest integer just below or at $2^{E+F-G} \cdot 5^F \cdot m \cdot n$, i.e. performing round-down, means to be able to decide whether the exact value $2^{E+F-G} \cdot 5^F \cdot m \cdot n$ is just above, at or below an integer candidate $p$.

In other words, correct rounding of a mixed-radix multiplication means determining whether an approximation to $2^{E+F-G} \cdot 5^F \cdot m \cdot n$ is below or above the middle of two integers or below or above an integer. The set of the middle of the integers together with the set of the integers is described by $1/2 \cdot p'$, where $p' \in \mathbb{Z}$ is an integer.

Correct rounding for a mixed-radix multiplication in any rounding mode means deciding whether $2^{E+F-G+1} \cdot 5^F \cdot m \cdot n$ is below, at or above a certain integer $p'$. This decision problem

that applies to an operation essentially as global state of the floating-point environment. Thorough lecture of the IEEE754-2008 Standard actually reveals that the Standard distinguishes between the rounding-mode for the binary radix and the rounding-mode for the decimal radix and that the global state of the floating-point environment hence contains two current rounding-modes, one for each radix [117].

15. IEEE754-2008 also models its (sticky) floating-point flags as global state. Even though the rounding-mode is separate for both IEEE754 radices, the IEEE754-2008 global state for the flags is shared by the operations of both radices [117]. The inexact flag may for example be raised as a result of a binary or a decimal operation that is inexact.

16. See also Section 2.3 for details.

is quite the same as the one we have already described and handled in Section 3.3, where we first addressed mixed-radix algorithms, proposing a mixed-radix comparison. Indeed, whatever the accuracy requirements be, we will always be able to design algorithms to compute approximations to $2^{E+F-G+1} \cdot 5^F \cdot m \cdot n$. Hence, if we know a lower bound on the smallest non-zero distance between an integer $p'$ and $2^{E+F-G+1} \cdot 5^F \cdot m \cdot n$, it suffices to approximate this quantity $2^{E+F-G+1} \cdot 5^F \cdot m \cdot n$ with an error less than this non-zero distance to deduce a correct rounding.

Determining a lower bound on the non-zero distance of $2^{E+F-G+1} \cdot 5^F \cdot m \cdot n$ and an integer $p'$ is easy, utilizing the same techniques based on rational approximation as we used them in Section 3.3, Paragraph 3.3.4. Actually, determining that non-zero distance is equivalent to computing the lowest distance of $\frac{2^{G-E-F-1}}{5^F}$ to the fraction $\frac{m \cdot n}{p'}$, where $m, n$ and $p'$ are integers for which we know lower and upper bounds, typically clamping them into one binade or decade. Similarly, we know that $E, F$ and $G$ are integers with small ranges that, by the fact $m, n$ and $p'$ are small ranges, are closely interconnected. Hence we can compute the best (non-zero-error) approximations [49] of $\frac{2^{G-E-F-1}}{5^F}$ by fractions $\frac{h}{p'}$, taking for $h$ the bounds on the product $m \cdot n$, represented by $h$. This way, we can easily obtain a lower-bound on the least non-zero distance of $\frac{m \cdot n}{p'}$ with respect to $\frac{2^{G-E-F-1}}{5^F}$; this lower bound will be safe, even though it might not be tight, as the possible values for the numerator $h$ we are going to find might possibly not be products of two integers $m$ and $n$ with the given ranges [17].

Implementing mixed-radix floating-point multiplications is therefore not very difficult: it suffices to analyze the precise ranges of $E, F$ and $G$, as well as of $m, n$ and $p'$ corresponding to a set of (binary and decimal) input and output formats, compute the lower bounds on the possible approximation cases and write –in a straightforward way– approximation code with an accuracy better than to what is needed to overcome the worst case, depending on these lower bounds.

The situation for mixed-radix division and square root is quite similar to mixed-radix multiplication. In order to enable correctly rounded mixed-radix multiplication, we look for integers $h = m \cdot n$ and rounding boundary significands $p'$ such that $\frac{h}{p'}$ is close to $\frac{2^A}{5^B}$, where $A = G - E - F - 1$ and $B = F$. Both $A$ and $B$ are integers in quite a small range. For division, be it a division of a binary floating-point number by a decimal one or vice-versa and be it with a binary or a decimal result, we eventually look also for integers $h$ and $r$ derived from the input and rounding boundary significands $m$, $n$ and $p'$ such that $\frac{h}{r}$ is close to $\frac{2^A}{5^B}$, where $A$ and $B$ are still integers in small ranges: $A$ and $B$ are just derived differently from the input and output exponents. Square root is just slightly different: we must look for inputs $2^E \cdot m$ resp. $10^F \cdot n$ the square root of which is close to rounding boundaries $10^H \cdot q'$ resp. $2^G \cdot p'$. In other words we look for inputs $2^E \cdot m$ resp. $10^F \cdot n$ close to values $10^{2\,H} \cdot q'^2$ resp. $2^{2\,G} \cdot p'^2$. Taking in this case $h = q'^2$ resp. $h = p'^2$, we however end up looking for integers $m$, resp. $n$, and $h$ such that $\frac{m}{h}$, resp. $\frac{n}{h}$, is close to $\frac{2^A}{5^B}$, where $A$ and $B$ are still integers in small ranges. Therefore mixed-radix division and square root require some arithmetical algorithm design work but are no real challenge with existing approaches.

Among the five [18] IEEE754 basic operations, the real challenge for mixed-radix arithmetic comes from mixed-radix addition and subtraction [19]. Typically, for a floating-point addition,

17. We are just using the fact that the product of two integers is an integer, while, of course, not all integers are products of two integers.

18. six including FMA

19. As floating-point subtraction is just floating-point addition with a change in sign of the subtrahend operand,

two cases can be distinguished: the case when the two input operands are sufficiently close in terms of exponent that their aligned significands will lead to cancellation when subtracted one from the other and the other case when they are sufficiently far in terms of magnitude such that the result's order of magnitude is essentially the one of the bigger of the operands. The first of these two cases is commonly called *near path*, the latter *far path* [77,197].

Uniform-radix correctly rounded floating-point addition is based, for the near path, on the observation that when cancellation occurs, the floating-point operation becomes naturally exact, i.e. does not require any rounding [140,197]. This case corresponds to the case when Sterbenz' lemma can be applied [197,238]. For mixed-radix addition, there is no reason why this cancellation case should become exact: e.g. when both input operands are in decimal floating-point arithmetic, both $10$ and $-9.9$ are representable. Their sum of course is $0.1$ which clearly is not representable in binary floating-point arithmetic. A decimal plus decimal to binary mixed-radix addition can hence not suppose that no rounding occurs in near path cases. Furthermore, when the two operands to a mixed-radix addition are not in the radix, some sort of conversion of the binary operand to the decimal radix or vice versa will be needed. This conversion will always induce some error – at least in general[20]. While cancellation in the near-path makes the operation become exact for uniform-radix cases, the same cancellation in the mixed-radix case will amplify this conversion error:

$$a + b \cdot (1 + \varepsilon) = (a + b) \cdot \left( 1 + \frac{b}{a+b} \cdot \varepsilon \right), \quad |a + b| \ll |b| \,.$$

And even if we find ways to compute lower bounds on this catastrophic cancellation results of mixed-radix addition on its near path, i.e. if we find ways to make the conversion sufficiently accurate that some accuracy is left once the cancellation amplifies the error, the result of this (approximate) mixed-radix addition may itself be close to a rounding boundary of the output radix, preventing us to return the correctly rounded result with certainty.

On the far path, mixed-radix addition is no less difficult: assume, for example, that we want to implement a *binary plus binary gives decimal* addition. Denote the two operands with $a$ and $b$ and the result with $c$. Now suppose that we look for a binary floating-point number $a$ very close to a decimal floating-point number $c$. Such pairs of numbers $(a, c)$ are typically the cases for which radix conversion is hard to round in a directed rounding mode [49] or cases for which our mixed-radix comparison, presented in Section 3.3, needs to go for the accurate step. Hence $a$ is very close to $c$. Let $b$ now be the binary floating-point number closest to the distance between $c$ and $a$, i.e. let $b = \circ(c - a)$. Now suppose these values to be the input to a mixed-radix addition $a + b$, yielding a decimal number $c'$. As $a$ is very close to the decimal number $c$, and $b$ is close to the distance between $c$ and $a$, $a + b = c + (a - c) - \circ(c - a)$ will be extremely close to $c$, making correct rounding to decimal in a directed rounding mode very difficult.

As an example take the two IEEE754 binary64 numbers

$$a = 2^{-52} \cdot 9007167306718907 = 1.99999290611405799999999999272404238581 \ldots$$

we are going to continue the discussion only for "addition", subsuming subtraction.

20. This is not entirely correct. As 2 divides 10, a binary to decimal conversion, where the binary exponent range is bounded, can be made exact for any binary floating-point input, provided the decimal precision is large enough, typically in the order of several hundred decimal digits for IEEE754 binary64. We already used this fact in Section 3.2.3 and are going to exploit it again in Section 3.4.3.

and

$$b = 2^{-138} \cdot 2535301200456459 = 10^{-27} \cdot 7.27595761418\ldots$$

Their sum $a + b$ is

$$a + b = 1.99999290611405800000000000000000000000056\ldots$$

which makes directed rounding down to the IEEE754 decimal format extremely difficult: the rounded-down result is $10^{-15} \cdot 1999992906114058$, just slightly below $a + b$.

Even though we can construct exemplary cases where correct rounding of mixed-radix addition is hard to perform, the cases we can attain are only exemplary. We have no simple, systematic way to look for inputs to mixed-radix addition for which rounding is hard in round-to-nearest or a directed rounding mode. We did some research on trying to find algorithms to at least compute a safe, if not tight lower bound on the minimum non-zero distance between the inputs of a mixed-radix addition and rounding boundaries of its output. We are going to report on this research in Section 3.4.2 but we have to admit that the result we took away from this research effort is the understanding that such lower bounds cannot efficiently be computed.

Therefore we propose another approach to implementing a correctly rounded mixed-radix addition, without precomputing such lower bounds, "recycling" an idea we have already used in Section 3.2.3, where we could not precompute lower bounds either, as the input had arbitrary precision. We are going to present this approach in Section 3.4.3.

However, while we are at implementing a mixed-radix addition, with all the difficulties we encounter, we propose to immediately strive at designing a correctly rounded mixed-radix Fused-Multiply-And-Add (FMA) operation, that takes three operands $a, b, c$, each in either as a binary or a decimal floating-point number, and computes a correctly rounded result $d$, in either a binary or a decimal floating-point format. This result $d$ is the floating-point number closest (in round-to-nearest mode), or just below or above (in a directed rounding mode), to the exact value $a \times b + c$. We propose to implement FMA immediately for the following reasons:

— On the one hand, the multiplication part of an FMA can be made exact: exact multiplication of the significands is possible, the exponents to radices 2 and 5 are readily computed. Mixed-radix FMA is hence no more difficult than mixed-radix addition.

— On the other hand, once a mixed-radix FMA is implemented, providing the full set of mixed-radix addition, subtraction and multiplication operations is trivial. Indeed, it suffices to set certain operands to 0, 1 or $-1$.

— Finally, if some additional precision is accounted for the operands of a mixed-radix FMA, mixed-radix division and square root can also be reduced to mixed-radix FMA. In fact, it suffices to compute mixed-radix division or square root results with just enough accuracy to determine the closest rounding boundary $f$ in the output format[21]. The significand of such a rounding boundary takes just 1 bit more of precision, be the output format in binary or in decimal[22]. Correctly rounding a division $a/b$ or a square root of $a$ means deciding whether $a/b$ is less, equal or greater that rounding boundary $f$.

21. If determining the closest rounding boundary is difficult, correctly rounding the available approximation is not difficult in the first place [166].

22. This is trivial for binary. In decimal, the rounding boundaries that are integer significand midpoints are half-integers, which boils down to an integer mantissa of 1 bit more.

This means correctly computing the sign of $b \times f - a$ resp. $f \times f - a$, which is easy if a mixed-radix FMA is available.

A mixed-radix FMA hence encompasses all difficulties of mixed-radix floating-point arithmetic and gives us a way to easily provide all basic operations, $+, -, \times, /, \sqrt{}$ and FMA itself, with one algorithmic kernel.

This is the reason why we set out to try to compute hard-to-round cases for mixed-radix FMA first. We are going to report on this research in the next Section 3.4.2. Our report is however a negative one; no such hard-to-round cases search was possible. In the subsequent Section 3.4.3, we are then going to present an algorithm that provides for correct rounding of a mixed-radix FMA without knowledge of hard-to-round cases.

### 3.4.2 Hardly Feasible: Computing Hard-To-Round Cases

Implementing a fast but correctly rounded mixed-radix FMA, computing $a \times b + c$, where each of $a, b, c$ and the result are either a binary or a decimal IEEE754 floating-point number, would be easy if only we had knowledge of the particular input triplet $(a, b, c)$ for which the infinitely precise result $a \times b + c$ is closest to a rounding boundary $f$ of the output floating point format, without being precisely at a rounding boundary. Again, the rounding boundaries are points where the rounding changes from one floating-point number to the subsequent floating-point number; their set is formed by the floating-point numbers itself and the midpoint of floating-point numbers [166].

If we knew that hardest-to-round triplet $(a, b, c)$, we could simply determine its relative distance to the next rounding boundary $f$:

$$\eta = \left| \frac{a \times b + c}{f} - 1 \right|$$

and design our implementation around the following, well-known algorithmic scheme [197, 264]:

1. Approximate $a \times b + c$ with a relative error less than $\frac{1}{2}\,\eta$.
2. With the approximation, determine the closest rounding boundary $f$.
3. If the approximation is (relatively) closer to $f$ than $\eta$, the unknown, infinitely precise result must be at the rounding boundary. As weird as it sounds, correct rounding can hence be obtained by rounding the rounding boundary.
4. Otherwise, rounding the approximation yields to the same correct rounding as if the unknown, infinitely precise result were used; if it did not, we had a contradiction to the statement that $\eta$ corresponds to the triplet $(a, b, c)$ for which the infinitely precise result is closest to the rounding boundary [173, 197, 264].

Hence, we are driven to set out for computing this hardest-to-round cases. While we are at it, we can also compute other hard-to-round cases, as they serve for testing purposes [27, 173]. However, computing hard-to-round and the hardest-to-round case can of course not be done by exhaustively enumerating all possible input triplets $(a, b, c)$ and determining the relative distance to the closest rounding boundary: the sheer number of possible inputs, in the order of $2^{(64-3)\cdot 3}$ for $64$ bit wide formats, makes this approach infeasible [173].

The only hope we might have lies in the applicability of algorithms allowing to compute the smallest non-zero relative distance of a rational fraction $p/q$ to some real number $\alpha \in \mathbb{R}$;

where the numerator $p$ and the denominator $q$ are bounded in magnitude [49]. We had already applied this theory in Section 3.3 for mixed-radix comparison and we are going to use it again for bounding cancellation in near-path mixed-radix subtraction in Section 3.4.3. The algorithms developed for this kind of search allow to compute the closest, non-exact left- and right approximation [49] of $\alpha$ by $p/q$: they allow us to determine

$$\left| \frac{p}{q} - \alpha \right|,$$

where $p$ and $q$ are integers within known bounds, $p_{\text{min}} \leqslant p \leqslant p_{\text{max}}$ and $q_{\text{min}} \leqslant q \leqslant q_{\text{max}}$.

To start with, it is easy to see that we can encompass the result of the FMA's multiplication part, $a \times b$, as $2^{A''} \cdot 5^{B''} \cdot m$, where $m \in \mathbb{N}$ is an integer significand between some bounds, $m_{\text{min}} \leqslant m \leqslant m_{\text{max}}$, typically a binade. We can use this representation independently of the radix of the inputs. As a matter of course, the integer exponents $A'' \in \mathbb{Z}$ and $B'' \in \mathbb{Z}$ are also bounded; it is easy to deduce the bounds from the formats' description in the IEEE754-2008 Standard [117]. We can hence state $A''_{\text{min}} \leqslant A'' \leqslant A''_{\text{max}}$ and $B''_{\text{min}} \leqslant B'' \leqslant B''_{\text{max}}$.

Similar modeling is, of course, possible for the additive part of a mixed-radix FMA: we notate $c = 2^{C''} \cdot 5^{D''} \cdot n$, with $n \in \mathbb{N}$ such that $n_{\text{min}} \leqslant n \leqslant n_{\text{max}}$ and $C'', D'' \in \mathbb{Z}$ bounded by $C''_{\text{min}} \leqslant C'' \leqslant C''_{\text{max}}$ and $D''_{\text{min}} \leqslant D'' \leqslant D''_{\text{max}}$. Finally, the rounding boundary $f$ can also be expressed as $f = 2^E \cdot 5^F \cdot q$, this time with $q \in \mathbb{Z}$ such that $q_{\text{min}} \leqslant |q| \leqslant q_{\text{max}}$ and $E, F \in \mathbb{Z}$ with $E_{\text{min}} \leqslant E \leqslant E_{\text{max}}$ and $F_{\text{min}} \leqslant F \leqslant F_{\text{max}}$.

Given this modelization, computing the hardest-to-round case $\eta$ boils down to finding $A'', B'', C'', D'', E, F$ as well as $m, n, q$ within the given bounds such that

$$\eta = \left| \frac{2^{A''} \cdot 5^{B''} \cdot m \pm 2^{C''} \cdot 5^{D''} \cdot n}{2^E \cdot 5^F \cdot q} - 1 \right|$$

is minimal without being zero.

This problem can still be simplified a little bit: the exponents $E$ and $F$ can be combined with $A'', B'', C''$ and $D''$. Let $A' = A'' - E$, $B' = B'' - F$, $C' = C'' - E$ and $D' = D'' - F$. Then it suffices to consider the problem of finding the minimum non-zero value of

$$\eta = \left| \frac{2^{A'} \cdot 5^{B'} \cdot m \pm 2^{C'} \cdot 5^{D'} \cdot n}{q} - 1 \right|$$

where the bounds on $m, n, q$ are unchanged and where the bounds on $A', B', C'$ and $D'$ can be trivially deduced from the bounds on $A'', B'', C''$ and $D''$.

As some or all of the exponents $A', B', C'$ and $D'$ may become negative, we are not yet quite at an integer fraction problem. Therefore, it is reasonable to set

$$\begin{array}{rcl} \gamma & = & \min\left(A''_{\text{min}} - E''_{\text{max}}, C''_{\text{min}} - E''_{\text{max}}\right) \text{ and} \\ \delta & = & \min\left(B''_{\text{min}} - F''_{\text{max}}, D''_{\text{min}} - F''_{\text{max}}\right). \end{array}$$

With $A = A' - \gamma$, $B = B' - \delta$, $C = C' - \gamma$ and $D = D' - \gamma$ the problem becomes

$$\eta = \frac{1}{2^{-\gamma} \cdot 5^{-\delta}} \left| \frac{2^A \cdot 5^B \cdot m \pm 2^C \cdot 5^D \cdot n}{q} - 2^{-\gamma} \cdot 5^{-\delta} \right|$$

where we can now state that $0 \leqslant A \leqslant A_{\text{max}}$, $0 \leqslant B \leqslant B_{\text{max}}$, $0 \leqslant C \leqslant C_{\text{max}}$ and $0 \leqslant D \leqslant D_{\text{max}}$. The upper bounds are readily computed.

Some under-estimate for $\eta$ can still be computed for the addition case: we can consider the integer $z = 2^A \cdot 5^B \cdot m + 2^C \cdot 5^D \cdot n$ with no more information than some easily computed lower and upper bounds and then consider the integer fraction problem

$$\eta' = \left| \frac{z}{q} - \alpha \right|$$

where $\alpha$ is set to $2^{-\gamma} \cdot 5^{-\delta}$. As there are for different exponents, $A$, $B$, $C$ and $D$ involved and for each combination of them a different $z$ needs to be considered, bounded and input into an integer fraction problem, this is already pretty challenging, but still feasible, in particular, as in the case when one of the two terms $2^A \cdot 5^B \cdot m$ or $2^C \cdot 5^D \cdot n$ is preponderant several subsequent values of the exponents can be combined into one, slightly larger interval of admissible values for $z$.

For the subtraction case where $z$ would be given as $z = 2^A \cdot 5^B \cdot m - 2^C \cdot 5^D \cdot n$ we get to the point where there is the rub: no reasonable bounds can be found for $z$, in particular for the cases when $A$ and $B$ make $2^A \cdot 5^B \cdot m$ and $2^C \cdot 5^D \cdot n$ cancel.

Currently, no solution seems to exist for this problem or we ignore it, although we have spent considerable amount of time searching such a solution. This is why we ended up considering another approach, based on the same idea we used when we had no access to worst-case accuracy values for decimal-to-binary conversion in Section 3.2.3. We are going to explain this approach in the next Sections.

### 3.4.3 A Correctly Rounded Mixed-Radix Fused-Multiply-Add

The input of the mixed-radix FMA algorithm are either binary64 or decimal64 with the possibility of representing the midpoints by slightly increasing the precision. In a similar manner to what we stated already informally in Section 3.4.1, those two formats are defined by the following equations:

$$\begin{aligned} &2^E \cdot m;\ \text{with } 2^{54} \leqslant |m| < 2^{55}; \\ &-1130 \leqslant E \leqslant 969;\ m, E \in \mathbb{Z}, \end{aligned} \tag{3.27}$$

and

$$\begin{aligned} &10^F \cdot n = 2^J \cdot 5^K \cdot r;\ \text{with } 2^{54} \leqslant |r| < 2^{55};\ r \in \mathbb{Z} \\ &-452 \leqslant J \leqslant 385;\ -421 \leqslant K \leqslant 385;\ J, K \in \mathbb{Z}. \end{aligned} \tag{3.28}$$

Both formats can also be combined into a single representation of the mixed-radix unified format, with pessimist bounds on the exponents such as, given $a$ an input of the FMA we have:

$$\begin{aligned} &a = 2^{N_a} \cdot 5^{P_a} \cdot t_a;\ \text{with } 2^{54} \leqslant |t_a| < 2^{55};\ t_a \in \mathbb{Z} \\ &-1130 \leqslant N_a \leqslant 969;\ -421 \leqslant P_a \leqslant 385;\ N_a, P_a \in \mathbb{Z}. \end{aligned} \tag{3.29}$$

The algorithm that performs a fast and correctly rounded FMA defined with Algorithm 7 can be split in several parts. The first step is the decomposition of the binary64 and decimal64 inputs into the explicit format of sign, exponents and significand. Throughout this algorithm, we will analyze the bounds of this variables because we want to make sure that we can store them on machine words: for the exponent on a signed 32 bit integer machine word and for the significand on one or several unsigned $64$ bit integer machine words. With this internal format, we compute an error-free mixed-radix multiplication. Then we perform a test to decide which addition/subtraction algorithm to use, which we will describe in more details later. According to the result of this test, we convert the operands into an internal binary format and perform either a *far-path* addition/subtraction, or a *near-path* subtraction. We will then test if the final rounding in the output radix is possible with this fast-computed result. If the result is too close to a midpoint $f$ and we cannot decide what is the correct floating-point value that should take the result according to the current rounding mode [66]. In that case we will have an extra computation step, called the recovery phase, to get those last bits of precision needed to decide how to round the result.

```
Input: $a, b$ and $c$, binary64 or decimal64 numbers
Output: $R = 2^G \cdot p$ in binary or $R = 10^H \cdot q$ in decimal
$\psi \leftarrow a \times b$;
if it is an "addition" or $\left|\frac{\psi}{c}\right| \notin [\frac{1}{2}; 2]$ then
    $T_1 \leftarrow 2^{\gamma_1} \cdot v_1 = \psi \cdot (1 + \varepsilon_{T1_{FP}})$;
    $T_2 \leftarrow 2^{\gamma_2} \cdot v_2 = c \cdot (1 + \varepsilon_{T2_{FP}})$;
    $\varphi \leftarrow (T_1 \pm T_2)(1 + \varepsilon_{\varphi_{FP}})$ with the "far-path" binary addition/subtraction;
else
    $T_1 \leftarrow 2^{\Gamma_1} \cdot w_1 = \psi \cdot (1 + \varepsilon_{T1_{NP}})$;
    $T_2 \leftarrow 2^{\Gamma_2} \cdot w_2 = c \cdot (1 + \varepsilon_{T2_{NP}})$;
    $\varphi \leftarrow (T_1 - T_2)(1 + \varepsilon_{\varphi_{NP}})$ with the "near-path" binary subtraction;
end
$\rho \leftarrow 2^C \cdot g = \varphi$ resp. $\rho \leftarrow 10^H \cdot 2^{-10} \cdot q = \varphi \cdot (1 + \varepsilon_{\rho_D})$;
if $\rho = (a \times b + c)(1 + \varepsilon_\rho)$ can be rounded correctly then
    return $R \leftarrow \circ(\rho)$ correctly rounded into the output format;
else
    Compute integer rounding boundary significand $f$ such that
    $2^C \cdot 2^{10} \cdot 1/2 \cdot f$ resp. $10^H \cdot 2^{-10} \cdot 2^{10} \cdot 1/2 \cdot f$ is the nearest rounding boundary;
    $\alpha \leftarrow 2^{L-Z_{min}} \cdot 5^{M-F_{min}} \cdot (a \times b) + 2^{N_c-Z_{min}} \cdot 5^{P_c-F_{min}} \cdot c - 2^{F_2-Z_{min}} \cdot 5^{F_5-F_{min}} \cdot f \in \mathbb{Z}$;
    Correct $\rho$ using $f$ and the sign of $\alpha$;
    return $R \leftarrow \circ(\rho)$ correctly rounded into the output format;
end
```

**Algorithm 7:** Correctly Rounded Mixed-Radix FMA Algorithm

In the following section, we will describe further the mechanisms of the mixed-radix FMA algorithm. We will first describe the *far-path* and *near-path* addition/subtraction algorithms of the fast computation, and then explain how the rounding test and recovery phase work.

#### Multiplication and Ratio Test

The first operation performed by the FMA is pretty simple: the multiplication can be computed exactly in the mixed-radix unified format. According the definition of the mixed-radix unified format of the input operands $a$ and $b$ given in equation (3.29), we can compute the result of the multiplication such that

$$\begin{aligned} &\psi = a \times b = 2^L \cdot 5^M \cdot s;\ \text{with } 2^{109} \leqslant |s| < 2^{110};\ s \in \mathbb{Z} \\ &-2261 \leqslant L \leqslant 1938;\ -842 \leqslant M \leqslant 770;\ L, M \in \mathbb{Z}. \end{aligned} \tag{3.30}$$

We then want to perform the addition/subtraction between $\psi$ and $c$, the third input operand of the FMA that is represented in the format given in equation (3.29). Typically, for floating-point addition, two cases can be distinguished: the case when the two input operands are sufficiently close in terms of exponent that their aligned significands will lead to cancellation when subtracted such that the result's order of magnitude is essentially the one of the bigger of the operands.

We can distinguish those two cases by performing a test on $\psi$ and $c$. If the ratio $\frac{\psi}{c}$ is clearly outside of $[\frac{1}{2}; 2]$, the comparison returns a positive answer, and a negative result otherwise. This means that there might be inputs with a ratio outside of $[\frac{1}{2}; 2]$ for which the operation returns a negative answer but the operation ensures that when it returns a positive result value, the ratio is surely outside of $[\frac{1}{2}; 2]$.

Given the definitions of $\psi$ and $c$ given in equations (3.29) and (3.30), we want to ensure that either we have

$$2^{L-N_c} \cdot 5^{M-P_c} \cdot \frac{s}{t_c} < \frac{1}{2} \quad \text{or} \quad 2^{L-N_c} \cdot 5^{M-P_c} \cdot \frac{s}{t_c} > 2.$$

Therefore, ensuring that the ratio $\frac{\psi}{c}$ is clearly outside of $[\frac{1}{2}; 2]$ is equivalent to computing the following inequations

$$\begin{aligned} &(L - N_c) + \lfloor (M - P_c) \cdot \log_2(5) \rfloor + 1 + 56 < -1 \\ &\text{or } (L - N_c) + \lfloor (M - P_c) \cdot \log_2(5) \rfloor + 54 > 1 \end{aligned} \tag{3.31}$$

Given the precomputed value $\log_2(5)$, $\lfloor A \cdot \log_2(5) \rfloor$ with $A \in \mathbb{Z}$ in a small domain can be computed correctly as

$$\lfloor A \cdot \lfloor \log_2(5) \cdot 2^{48} \rfloor \cdot 2^{-48} \rfloor$$

which does not overflow on a $64$ bit signed integer variable [28]. Hence, we can easily compute on a $64$ bit signed integer the two values given in equation (3.32) and perform ratio test that gives us two cases.

### When no cancellation occurs (far-path)

**Far-path algorithm** When the ratio test states that the result is clearly outside of $[\frac{1}{2}; 2]$, we are sure that no cancellation can occur during the computation of the addition/subtraction. Thus we can use a *far-path* addition algorithm.

To perform the addition/subtraction of $\psi$ and $c$ represented in the unified mixed-radix format, we first want to perform a conversion to a $64$ bits of precision binary format. This means that we want to represent the significand $v$ stored on a $64$ bit unsigned integer such that $2^{63} \leqslant |v| \leqslant 2^{64} - 1$, and adapt the binary exponent $\gamma$. This way we can compute the result of the addition/subtraction in binary with correct rounding according to the current rounding mode, while delaying the mixed-radix considerations for the few cases, when rounding to the output format is hard.

As the operands $\psi$ and $c$ do not have the same significand precision, we might first think that we need two conversion algorithms from unified mixed-radix format to a $64$ bits of precision binary format. But it is easy to show that we can first represent them into a unified mixed-radix format with the same precision, here $64$ bits, such as

$$\begin{aligned} &2^{\lambda} \cdot 5^{\mu} \cdot \omega;\ \text{with } 2^{63} \leqslant |\omega| \leqslant 2^{64}-1;\ \omega \in \mathbb{Z} \\ &-2215 \leqslant \lambda \leqslant 1284;\ -842 \leqslant \mu \leqslant 770;\ \lambda, \mu \in \mathbb{Z}. \end{aligned} \tag{3.32}$$

We can then apply a unique conversion algorithm from this mixed-radix format to a binary variable $T$ represented with a $64$ bit unsigned integer significand compliant to the definition:

$$\begin{aligned} &T = 2^{\gamma} \cdot v;\ \text{with } 2^{63} \leqslant |v| \leqslant 2^{64}-1;\ v \in \mathbb{Z} \\ &-4171 \leqslant \gamma \leqslant 3772;\ \gamma \in \mathbb{Z}. \end{aligned} \tag{3.33}$$

Applying this conversion algorithm on $\psi$ and $c$ gives us the two variables $T_1 = 2^{\gamma_1} \cdot v_1$ and $T_2 = 2^{\gamma_2} \cdot v_2$ defined like in equation (3.33) as inputs for the *far-path* addition/subtraction algorithm, defined with Algorithm 8.

```
Input: $T_1 = 2^{\gamma_1} \cdot v_1$ and $T_2 = 2^{\gamma_2} \cdot v_2$
Output: $\varphi = 2^C \cdot g = (T_1 \pm T_2)(1 + \varepsilon_{\varphi_{FP}})$
Order $T_1$ and $T_2$ such that $\gamma_1' \geqslant \gamma_2'$;
if $\gamma_1' \geqslant \gamma_2' + 64$ then
    $g \leftarrow v_1'$;
    $C \leftarrow \gamma_1'$;
else
    $V_1 \leftarrow 2^{\gamma_1' - \gamma_2'} \cdot v_1'$;
    $V_2 \leftarrow v_2'$;
    $V \leftarrow V_1 \pm V_2$;
    $\sigma \leftarrow \texttt{lzc}(V)$;
    $\pi \leftarrow 2^{\sigma} \cdot V$;
    $g \leftarrow \lfloor 2^{-64} \cdot \pi \rfloor$;
    $C \leftarrow \gamma_2' + 64 - \sigma$;
end
return $\varphi = 2^C \cdot g$;
```

**Algorithm 8:** Far-Path Addition/Subtraction Algorithm

The overall philosophy of this algorithm is that when neither of the operands are zero, we can order $T_1$ and $T_2$ according to their exponents such that $\gamma_1' \geqslant \gamma_2'$. Absorption occurs when $\gamma_1' \geqslant \gamma_2' + 64$, in that case the result of the far-path addition is $2^{\gamma_1'} \cdot v_1'$. Otherwise, to perform the addition of the two $64$ bit unsigned integer exponents $v_1'$ and $v_2'$, we pose the intermediate $128$ bit unsigned integer variables $V_1$ and $V_2$ and align the exponents by shifting $v_1'$ to the left such that $V_1 = 2^{\gamma_1' - \gamma_2'} \cdot v_1'$. After performing the addition or subtraction on a $128$ bit unsigned integer variable $V = V_1 \pm V_2$, the significand is normalized with the function `lzc`$(V)$ that counts the number of leading zeros and rounded to a $64$ bit unsigned integer variable $g$. The exponent $C$ is set accordingly.

The result of the *far path* addition algorithm is then returned in the following binary format

$$\begin{aligned} &2^C \cdot g;\ \text{with } 2^{63} \leqslant |g| \leqslant 2^{64}-1;\ g \in \mathbb{Z} \\ &-4174 \leqslant C \leqslant 3836;\ C \in \mathbb{Z}. \end{aligned} \tag{3.34}$$

**Far-path error analysis** We will analyze the global relative error made during the computation of the far-path addition subtraction algorithm, from the conversion of the inputs $\psi$ and $c$ until the acquisition of the result $\varphi = 2^C \cdot g$. We shall denote this global error $\varepsilon_{\varphi_{FP}}$.

The first operation producing an error occurs during the computation of the equation (3.32). In the case of $\psi$, as the precision decreased from 110 bits to 64 bits, this operation results in an error $2^{\lambda_\psi} \cdot 5^{\mu_\psi} \cdot \omega_\psi = 2^L \cdot 5^M \cdot s \cdot (1+\varepsilon_\psi)$ and that error can be bounded by $|\varepsilon_\psi| \leqslant 2^{-63}$.

Then, given an input conform with the format described in equation (3.32), we compute the exponent $\gamma$ and significand $v$ as:

$$\gamma = \lambda + \lfloor \mu \cdot \log_2(5) \rfloor;\ v = \lfloor 2^{-64} \cdot \omega \cdot \tau_0(\mu) \rfloor \tag{3.35}$$

with the precomputed table $\tau_0(\mu) = \lfloor 5^\mu \cdot 2^{-\lfloor \mu \cdot \log_2(5) \rfloor + 63} \rfloor$, represented by a table of 64 bit unsigned integers. The relative error arising during the computation of the table can be bounded such that $|\varepsilon_{\tau_0(\mu)}| \leqslant 2^{-63}$.

The global relative error attached to the computation of the 64 bit precision binary variables $T_1$ and $T_2$ is in the worst case $2^\gamma \cdot v = 2^\lambda \cdot 5^\mu \cdot \omega \cdot (1+\varepsilon_v) \cdot (1+\varepsilon_{\tau_0(\mu)})$.

The rounding relative error of $v$ is a combination of the truncation error and the error attached to the computation of the table $\tau_0(\mu)$ such that $\varepsilon_v$ can be bounded by $|\varepsilon_v| \leqslant 2^{-62}$.

The global relative error for this part of the mixed-radix FMA algorithm includes the conversion error $\varepsilon_\psi$, the error on the precomputed table $\varepsilon_{\tau_0(\mu)}$ and the rounding error $\varepsilon_v$. With these three errors combined, we deduce that a pessimist bound for this global relative error is $|\varepsilon_{T_{FP}}| \leqslant 2^{-60.5}$.

Given $T_1$ and $T_2$ the inputs of the *far-path* addition algorithm, we want to bound the relative error $\varepsilon_{\varphi_{FP}}$ such that

$$\varphi = (T_1 \pm T_2)(1+\varepsilon_{\varphi_{FP}}). \tag{3.36}$$

The algorithm can compute the result in two different ways, firstly if $\gamma_1' \geqslant \gamma_2' + 64$, the result is $\varphi = 2^{\gamma_1'} \cdot v_1' \cdot (1+\varepsilon_{\varphi_{FP}})$ with $|\varepsilon_{\varphi_{FP}}| \leqslant 2^{-63}$.

Secondly, if we are in the case where $\gamma_1' < \gamma_2' + 64$, hence $\gamma_2' \leqslant \gamma_1' \leqslant \gamma_2' + 63$ because $\gamma_1', \gamma_2' \in \mathbb{Z}$, we can deduce the bounds on $V_1$ and $V_2$ defined in the algorithm 8 such that $2^{63} \leqslant V_1 < 2^{127}$ and $2^{63} \leqslant V_2 < 2^{64}$.

We want to compute bounds on $V = V_1 \pm V_2$. We have two cases: either it is an addition and we have

$$2^{64} \leqslant V_1 + V_2 < 2^{127} + 2^{64} < 2^{128},$$

or it is a subtraction, with a reduction to absurdity we can show that

$$2^{61} \leqslant \frac{1}{2} \cdot (1 - 2^{-59.49}) \cdot 2^{63} \leqslant V_1 + V_2 < 2^{127} + 2^{64} < 2^{128},$$

with the fact that

$$1 - 2^{-59.49} \leqslant \frac{1+\varepsilon_{T1_{FP}}}{1+\varepsilon_{T2_{FP}}} \leqslant 1 + 2^{-59.49}.$$

Following the steps of the algorithm, we then bound $\sigma$ according to its definition such that $0 \leqslant \sigma \leqslant 66$; thus $\pi$ is between $2^{127} \leqslant \pi \leqslant 2^{128}$ and finally we have

$$g = \lfloor 2^{-64} \cdot \pi \rfloor = 2^{-64} \cdot \pi \cdot \varepsilon_{\varphi_{FP}}$$

In the end we have $|\varepsilon_{\varphi_{FP}}| \leqslant 2^{-63}$.

**When cancellation does occur (near-path)**

When the ratio test states that the result may be inside of $[\frac{1}{2}; 2]$, cancellation may occur during the computation of the subtraction, we want thus to use the *near-path* subtraction algorithm.

A solution to compute this near-path algorithm is to use the same method as the far-path algorithm and perform the subtraction on binary operands. To do so, we need to determine the number of bits we need for the binary precision operands to compute the near-path subtraction in the worst case of cancellation and avoid loosing all accuracy.

**Worst case of cancellation** The worst case occurs when, without being zero, the subtraction between $\psi$ and $c$ such that $2^L \cdot 5^M \cdot s - 2^{N_c} \cdot 5^{P_c} \cdot t_c$ is relatively small. This is equivalent to the following equation:

$$\begin{aligned} 2^L \cdot 5^M \cdot s &\approx 2^{N_c} \cdot 5^{P_c} \cdot t \\ \frac{s}{t} &\approx 2^{N_c - L} \cdot 5^{P_c - M} \end{aligned} \tag{3.37}$$

with $2^{54} \leqslant |t| < 2^{55}$ and $2^{109} \leqslant |s| < 2^{110}$.

Let us define $X$ and $Y$ such that $X = N_c - L$ and $Y = P_c - M$; $X, Y \in \mathbb{Z}$. With this definition and the bounds of $M$, $L$, $P_c$ and $N_c$ given in equations (3.29) and (3.30), we can easily determine the bounds of $X$ and $Y$.

**Lemma 1.** *For all $X, Y \in \mathbb{Z}$ with $-3068 \leqslant X \leqslant 3230$ and $-1191 \leqslant Y \leqslant 1227$, for $s, t \in \mathbb{Z}$ with with $2^{54} \leqslant |t| < 2^{55}$ and $2^{109} \leqslant |s| < 2^{110}$*

$$\left| \frac{s}{t} - 2^X \cdot 5^Y \right| \geqslant \eta = 2^{-177.61}.$$

*Proof.* A proof and implementation of this method to compute the worst case of cancellation can be found in [28]. □

**Near-path algorithm** Subsequently, we deduce that to compute the near-path subtraction with at least 60 bits of precision, we need to represent the binary significands with at least $177 + 60 = 227$ bits of precision. This means that we can represent the significands of the operands of the near-path subtraction on a 256 bit unsigned integer, i.e. four 64 bits unsigned integers machine words.

Hence we first compute the conversion from the mixed-radix unified format variables $phi$ and $c$ to 256 bit precision variables, which we perform the near path subtraction with.

As for the far-path algorithm, we want to perform only one conversion to the binary internal format, therefore adapt the significand and exponent of $c$ to be conform with the definition of $\psi$ in equation (3.30). We can then perform the conversion to the 256 bit precision binary variable $T$ such that

$$\begin{aligned} &T = 2^{\Gamma} \cdot w; \text{ with } 2^{255} \leqslant |w| \leqslant 2^{256} - 1; \ w \in \mathbb{Z} \\ &- 4364 \leqslant \Gamma \leqslant 3578; \ \Gamma \in \mathbb{Z}. \end{aligned} \tag{3.38}$$

The near-path subtraction algorithm is quite similar to the far-path except that the alignment of the significands produce an error. We first order the two operands $T_1 = 2^{\Gamma_1} \cdot w_1$ and $T_2 = 2^{\Gamma_2} \cdot w_2$ such that $\Gamma_1 > \Gamma_2$. With the result $T_1'$ and $T_2'$, we align the significands such that

$$z_2 = \lfloor 2^{\Gamma_2' - \Gamma_1'} \cdot w_2' \rfloor; \ z_1 = w_1' \tag{3.39}$$

After ordering $z_1$ and $z_2$ such that $z_1' > z_2'$, we can compute the subtraction $d = z_1' - z_2'$, perform the count of leading zeros in $d$ with $\mathtt{lzc}(d)$ such that we can normalize the significand and exponent of the result as

$$g = \lfloor d \cdot 2^{l} \cdot 2^{-192} \rfloor; \text{ and } C = \Gamma_1' - l + 192. \tag{3.40}$$

The result of the *near-path* subtraction is then returned in the form as the result of the *far-path* addition defined in equation (3.34), except for the bounds of the exponent that are the following $-2296 \leqslant C \leqslant 3773$.

**Near-path error analysis** The first error that occurs during the computation of the near-path algorithm comes from the conversion from the mixed-radix unified format to the 256 bit precision binary format. We can denote this error $\varepsilon_{T_{NP}}$ and define it such that

$$2^{\Gamma} \cdot w = 2^{L} \cdot 5^{M} \cdot s \cdot (1 + \varepsilon_{T_{NP}}) \tag{3.41}$$

computed in the following way

$$\begin{aligned} &\Gamma = L - 18 + \lfloor M \cdot \log_2(5) \rfloor + \sigma \text{ with } 0 \leqslant \sigma \leqslant 1; \ \sigma \in \mathbb{N} \\ &\text{and } w = \lfloor 2^{-129} \cdot \tau_{0-3}(M) \cdot s \cdot 2^{18} \rfloor \cdot 2^{\sigma}. \end{aligned} \tag{3.42}$$

The table $\tau_{0-3}(M)$ is the 256 bit precision version of the table $\tau_0(\mu)$ used before such that we can show that we have $\tau_0(M) = \lfloor 2^{-192} \cdot \lfloor 2^{255 - \lfloor M \cdot \log_2(5) \rfloor} \cdot 5^{M} \rfloor \rfloor = \lfloor 2^{-192} \cdot \tau_{0-3}(M) \rfloor$.

We can then bound the relative error induced by the computation of the table such that $|\varepsilon_{\tau_{0-3}}| \leqslant 2^{-255}$, and the global relative error of the conversion algorithm such as $|\varepsilon_{T_{NP}})| \leqslant 2^{-251}$.

The result $\varphi$ of the near-path binary subtraction is subject to an error such that $\varphi = (T_1 - T_2)(1 + \varepsilon_{\varphi_{NP}})$. A first error appears during the computation of $z_2 = (2^{\Gamma_2' - \Gamma_1'} \cdot w_2') \cdot (1 + \varepsilon_{z_2})$, by noticing that $-2 \leqslant \Gamma_2' - \Gamma_1' \leqslant 0$ we can bound $\varepsilon_{z_2}$ such that $|\varepsilon_{z_2}| \leqslant 2^{-253}$.

This implies that $g = (d \cdot 2^{l} \cdot 2^{-192}) \cdot (1 + \varepsilon_{\varphi_{NP}})$ with $2^{255} \leqslant d \cdot 2^{l} \leqslant 2^{256} - 1$ and in the end we have that $|\varepsilon_{\varphi_{NP}}| \leqslant 2^{-63}$.

### Rounding Test and Recovery Phase

**Conversion to the output format** Once the multiplication and addition or subtraction are performed, we are left with a 64 bit variable of binary precision with an exponent bounded such that $2^{C} \cdot g$ with $2^{63} \leqslant |g| \leqslant 2^{64} - 1$ and $-4174 \leqslant C \leqslant 3836$. If we want a binary output for the FMA operation, we can directly perform the rounding test that informs us if the result can be correctly rounded into a binary64 floating-point number. If we want a decimal64 output, we have to perform a conversion from this $64$ bit precision binary number to a decimal such that

$$\begin{aligned} &10^H \cdot 2^{-10} \cdot q \text{ with } -1253 \leqslant H \leqslant 1159;\ q, H \in \mathbb{Z} \\ &10^{15} \cdot 2^{10} \leqslant q \leqslant (10^{16} - 1) \cdot 2^{10} < 2^{64} - 1. \end{aligned} \tag{3.43}$$

This algorithm computes the exponent $H = \lfloor (C + 63) \cdot \log_{10}(2) - 15 + \mu(C, g)$ where $\mu(C, g) \in \mathbb{N};\ \mu(C, g) = \lfloor C \cdot \log_{10}(2) + \log_{10}(g) - \lfloor (C + 63) \cdot \log_{10}(2) \rfloor \rfloor$. We can show that $0 \leqslant \mu(C, g) \leqslant 1$, implying that $\mu(C, g) \in 0, 1$. This proof is based on the observation that $\mu(C, g)$ is an increasing function in $g$. Thus for a given $C$, there exists $g^*(C) \in \mathbb{Z};\ 2^{63} \leqslant g^*(C) < 2^{64}$ such that

$$\mu(C, g) = \begin{cases} 0 \text{ if } g \leqslant g^*(C) \\ 1 \text{ if } g > g^*(C) \end{cases} \tag{3.44}$$

We can precompute such a $g^*(C)$ with a table of 64-bits unsigned integer words.

Once the exponent $H$ is set, we can compute the significand $q$ with a bounded precomputed table $t(H) = \lfloor 10^{-H} \cdot 2^{\lfloor H \cdot log_2(10) \rfloor + 127} \rfloor$ and multiply it by $g$. The value is then normalized and truncated properly to conform to the definition of the output of this operation described in equation (3.43).

We skipped the detailed error analysis of this algorithm for the sake of conciseness but the full pencil-and-paper proof will soon be available under the form of a research report. We can bound the relative error of this conversion such that $10^H \cdot 2^{-10} \cdot q = (2^C \cdot g) \cdot (1 + \varepsilon_{\rho_D})$ with $|\varepsilon_{\rho_D}| \leqslant 2^{-59.82}$.

**Rounding test** The goal of this test is to determine whether the output of the addition/-subtraction algorithm can be rounded from the $64$ bit precision in the output format to a a binary64, resp. decimal64 with any sign and rounding mode.

In the binary64 case, we have as input $2^C \cdot g = (\psi \pm c)(1 + \varepsilon_B^*)$ with $\varepsilon_B^*$ the global error combining all the previous errors such that $|\varepsilon_B^*| \leqslant 2^{-58.74}$ and $2^{63} \leqslant |g| \leqslant 2^{64} - 1$. The rounding test boils down to compute the "magic bound" $\beta_B = \lfloor (2^{64} \cdot \frac{\varepsilon_B^*}{1+\varepsilon_B^*} \rfloor$ and test if

$$|g - \lfloor 2^{-10} \cdot g \rceil 2^{10}| \geqslant \beta_B. \tag{3.45}$$

If this equation is not satisfied, then rounding to a 53-bit precision binary64 number is not possible. Therefore, we compute $f$ the binary rounding boundary closest to $2^C \cdot g$ such that $2^{TF} \cdot 5^{F_5} \cdot f = 2^C \cdot 2^{10} \cdot 1/2 \cdot f$ and $2^{54} \leqslant f \leqslant 2^{55} - 1$ and step into the recovery phase.

In the decimal64 case, we have as input $2^H \cdot 2^{-10} \cdot q = (\psi \pm c)(1 + \varepsilon_D^*)$ with $\varepsilon_D^*$ the global error combining all the previous errors such that $|\varepsilon_D^*| \leqslant 2^{-58.18}$ and $10^{15} \cdot 2^{10} \leqslant |q| \leqslant (10^{16} - 1) \cdot 2^{10}$. The rounding test boils down to compute the "magic bound" $\beta_D = \lfloor ((10^{16} - 1) \cdot 2^{10} \cdot \frac{\varepsilon_D^*}{1+\varepsilon_D^*} \rfloor$ and test if

$$|q - \lfloor 2^{-9} \cdot q \rceil 2^9| \geqslant \beta_D. \tag{3.46}$$

If this equation is not satisfied, then rounding to a 16-digit precision decimal64 number is not possible. Therefore we compute $f$ the binary rounding boundary closest to $2^H \cdot q$ such that $2^{TF} \cdot 5^{F_5} \cdot f = 10^H \cdot 2^{-10} \cdot 2^{10} \cdot 1/2 \cdot f$ and $2 \cdot 10^{15} \leqslant f \leqslant 2 \cdot (10^{16} - 1)$ and step into the recovery phase.

**Recovery phase** If the rounding test fails, that means that the result we are trying to round is too close to the rounding border $f$ to be able to decide to which nearest floating-point number the result must be rounded. As we have computed the value of this floating-point rounding border in the previous step, the recovery phase boils down to computing the sign of

$$\begin{aligned}\alpha' =&(-1)^{s_a} \cdot 2^{L-Z_{min}} \cdot 5^{M-F_{min}} \cdot s + (-1)^{s_b} \cdot 2^{N_c-Z_{min}} \\ &\cdot 5^{P_c-F_{min}} \cdot t_c - 2^{F_2-Z_{min}} \cdot 5^{F_5-F_{min}} \cdot f\end{aligned} \tag{3.47}$$

with guarantees on the exponents such that $0 \leqslant F_2 - Z_{min} \leqslant 2162;\ 0 \leqslant F_5 - F_{min} \leqslant 786;\ 0 \leqslant L - Z_{min} \leqslant 2282;\ 0 \leqslant M - F_{min} \leqslant 785;\ 0 \leqslant N_c - Z_{min} \leqslant 2342;\ 0 \leqslant P_c - F_{min} \leqslant 786$.

Given those bounds, we know that we need to compute positive or zero integer powers of 5, i.e. $5^k$, with $0 \leqslant k \leqslant 786$. Furthermore, we know that every $5^k$ with $0 \leqslant k \leqslant 786$ holds on $\lceil \log_2(5^{786}) \rceil = 1826$ bits, hence 29 words of 64 bits.

Finally, each of the terms $\alpha$ holds on respectively on $4218$, $4223$ and $4043$ bits, so whatever their sign is, they hold on a maximum of $4225$ bits. Thus, in order to compute $\alpha$ exactly and determine its sign, we need 67 words of 64 bits, leaving 63 "free" bits.

We then have three cases: if $\alpha = 0$ then the result is exactly the rounding border $f$, and will be rounded in the output format correctly by our final rounding algorithm, according to the current rounding mode. If $\alpha < 0$, then the result is below the rounding border $f$ and will round to the lower nearest floating-point number in the output format. By analogy, if $\alpha > 0$, then the result shall round to the higher nearest floating-point number in the output format.

Computing this recovery phase is therefore very costly, but, fortunately, this part of the FMA algorithm will only be computed sparsely.

**Final rounding to the output format** Final rounding to the supported output formats, binary64 and decimal64, takes either $(-1)^s \cdot 2^C \cdot g$, $2^{63} \leqslant g < 2^{64}$, or $(-1)^s \cdot 10^H \cdot 2^{-10} \cdot q$, $10^{15} \cdot 2^{10} \leqslant g < 10^{16} \cdot 2^{10}$, in input and rounds them to a binary64 or decimal64 floating-point number respectively. This rounding is performed in the current rounding mode set in the IEEE754 environment the code is run in. Binary rounding is done in the IEEE754 rounding mode for binary floating-point arithmetic, decimal rounding in the decimal mode. This final rounding also sets the inexact, underflow or overflow flags, when the rounding is respectively inexact, underflowed (tiny and inexact) or overflowed.

The difficulty in implementing this final operation is two-fold: first, the rounding must address all special cases, such as subnormal rounding, normal rounding, overflow, decimal denormalization at the least exponent etc. Second, the rounding must be performed with the appropriate rounding mode, access to which is difficult. It also must set the flags. We overcome both challenges by exactly computing binary64 resp. decimal64 numbers that, provided to a binary64 resp. decimal64 IEEE754 floating-point operation, round to the same value as the input data.

For binary64 rounding, we decompose $(-1)^s \cdot 2^C \cdot g$ into a normal binary64 value $2^A$, another normal binary64 value $2^B \cdot r$ and a normal or subnormal value $2^{C'} \cdot g'$ such that a binary64 FMA computes $2^A \times 2^B \cdot r + 2^{C'} \cdot g'$ and rounds to the same binary64 value as $(-1)^s \cdot 2^C \cdot g$. Typically $C' = C + 11$, $g' = \lfloor g \cdot 2^{-11} \rfloor$, $A + B = C$, $r = g - 2^{11} \cdot g'$.

For decimal64 rounding, we represent $(-1)^s \cdot 10^H \cdot 2^{-10} \cdot q$ exactly as an IEEE754 decimal128 floating-point value and perform an IEEE754 decimal128 to decimal64 format conversion, which rounds correctly and sets the flags[23]. This way of performing final decimal rounding

23. see note below in Section 3.4.4

is known to be sub-optimal in terms of performance but has the advantage of being easy to implement. We consider changing this part of our algorithm and perform decimal64 rounding manually, while detecting the appropriate rounding mode by executing the avatar decimal64 instruction $t = (-1)^s \cdot \left(10^{15} + 1\right)$ resp. $t' = (-1)^s \cdot \left(10^{15} + 2\right)$ added to $\delta$, with $\delta \in \{-3/4, -1/2, -1/4, 0, 1/4, 1/2, 3/4\}$, choosing $t, t'$ as well as $\delta$ depending on the parity of a truncated significand and the value of the rest that got truncated off.

### 3.4.4 Mixed-Radix FMA: Implementation and Test Results

**Reference implementations**

In order to perform software testing on our mixed-radix FMA implementation, we designed two other implementations of a mixed-radix FMA:

— one based on the GNU Multiple Precision Library (GMP) [102], with the same C calling interface as our optimized FMA implementation, and

— one written in Sollya [41], with an *ad-hoc* text-based interface.

While designing these two reference implementations, we set the following design goals: the GMP-based implementation should be as close as possible to what someone requiring a mixed-radix FMA would design in a limited timeframe, reusing existing software library. This implementation should be reasonably fast, but easy to design.

We addressed this design goal by converting all input operands, be they IEEE754 binary64 or decimal64 numbers, into GMP rational numbers, i.e. fractions of (long) integers. We then performed the multiplication and addition steps of the FMA in GMP rational numbers. GMP can perform these operations on rational numbers without any error, computing exact long products, sums and gcds as required, while allocating memory dynamically. Our reference implementation then rounds the rational number, call it $\frac{p}{q}$, by first determining the output exponent for a binary64 resp. decimal64 number, call it $E$, and then "rounding" the rational number $\frac{r}{s} = 2^{-E} \cdot \frac{p}{q}$ (resp. $\frac{r}{s} = 10^{-E} \cdot \frac{p}{q}$) to an integer significand by long division of the long integer $r$ by $s$, adapting the division rest's sign as required by the rounding mode. As the exponents of the input and output may grow relatively large, GMP is required to manipulate integers of size of about a couple of thousand bits in the worst case.

In the next Section 3.4.4, we give an overview of the testing of our optimized FMA implementation we performed, comparing its results to the GMP-based reference implementation, and also compare timing results.

While performing software development and intermediate testing of both our optimized implementation and the GMP-based one, we found certain input operands where the two implementations did not agree. In order to have a way to decide where to look out for issues, we designed yet another implementation of a mixed-radix FMA, in the interpreted numerical language Sollya [41]. In this tool, numerical expressions can be written as exact expressions, and evaluated to any precision without being hurt by spurious roundings. It is relatively easy to give definitions of the output exponent and significand of a mixed-radix FMA in terms of the input exponents and integer significands. Particularities due to subnormal rounding, underflow and overflow can be addressed through minimum and maximum computations.

We used our Sollya reference implementations to generate "gold" result test vectors, with which we then verified both our optimized FMA implementation and the GMP-based reference. After our software testing campaign, all our three implementations agreed.

While we spent quite some time on testing the implementation of our mixed-radix FMA algorithm against reference implementations, we would like to insist that we consider our algorithm to be correct only because we were able to prove its correctness. The detailed pencil-and-paper proof is too long to be included inhere but will be made available under the form of a research report soon.

### Test and Timing Results

We tested our mixed-radix FMA implementation in all its 14 different operand-result-format variants on a system based on an Intel i7-7500U quad-core processor, clocked at maximally 2.7GHz, running Debian/GNU Linux 4.9.0-5 in x86-64 mode. We compiled our code using gcc version 6.3.0 with optimization level 3 and setting the `-march=native` flag. On our system, this results in AVX instructions being used, including hardware binary64 FMA. The IEEE754-2008 decimal types and instructions come from our gcc's libgcc. The GMP-based reference implementation was based on GMP version 6.1.2. Sollya was at git commit 12bf2006c.

We tested each of the 14 variants on an appropriate test vector file containing at least 115000 test points, covering all possible cases of signs, signed zeros, infinities, NaNs, subnormal and normal numbers, as well as all possible binary and decimal rounding modes [24]. We found all our 14 mixed-radix FMA implementations to be correct with respect to their numerical outputs for all the outlined cases.

We also performed testing of the implementations concerning the correct setting of the IEEE754-2008 flags, as they are inexact, overflow, underflow, divide-by-zero and invalid [117]. We found all flags to be set correctly, i.e. our implementations correctly raised the appropriate IEEE754-2008 when the operation actually was inexact (i.e. implied rounding), overflowed, underflowed or was invalid (as for multiplication of zero by infinity). The implementation also was checked for correctly leaving the flag unchanged if it did not encounter the appropriate IEEE754-defined condition to raise the flag. One issue was found concerning this last check for the mixed-radix FMA implementations that return decimal results: for certain exact cases, the inexact flag was raised spuriously. We traced this undue raising of the inexact flag back to the implementation of the IEEE754 decimal128 to decimal64 format conversion that is contained in our version of libgcc. A bug has been filed against libgcc.

Concerning performance testing, we ran our 14 implementations on each of the entries of our result test vectors, timing the function call time in terms of machine cycles using the Intel `RDTSC` instruction after serialization with `cpuid`. We subtracted the measured time for calling an empty function with the same function prototype. We ran the timing measurements repeatedly and with preheated caches, in order to smooth out effects of modern superscalar processors. We made sure that the test vectors contained a representative, random subset of floating-point numbers, i.e. we made sure that special cases, such as addition and multiplication of zeros, NaNs etc. were represented but not over-represented.

We present our timing measurements under the form of histograms which relate a certain cycle count interval to the number of times this cycle count interval was encountered. We did the same performance testing which we run on our optimized FMA implementation on our GMP-based mixed-radix FMA implementation, too.

24. and rounding mode combinations, as IEEE754-2008 specifies separate current rounding modes for the binary and decimal operations' subsets [117]

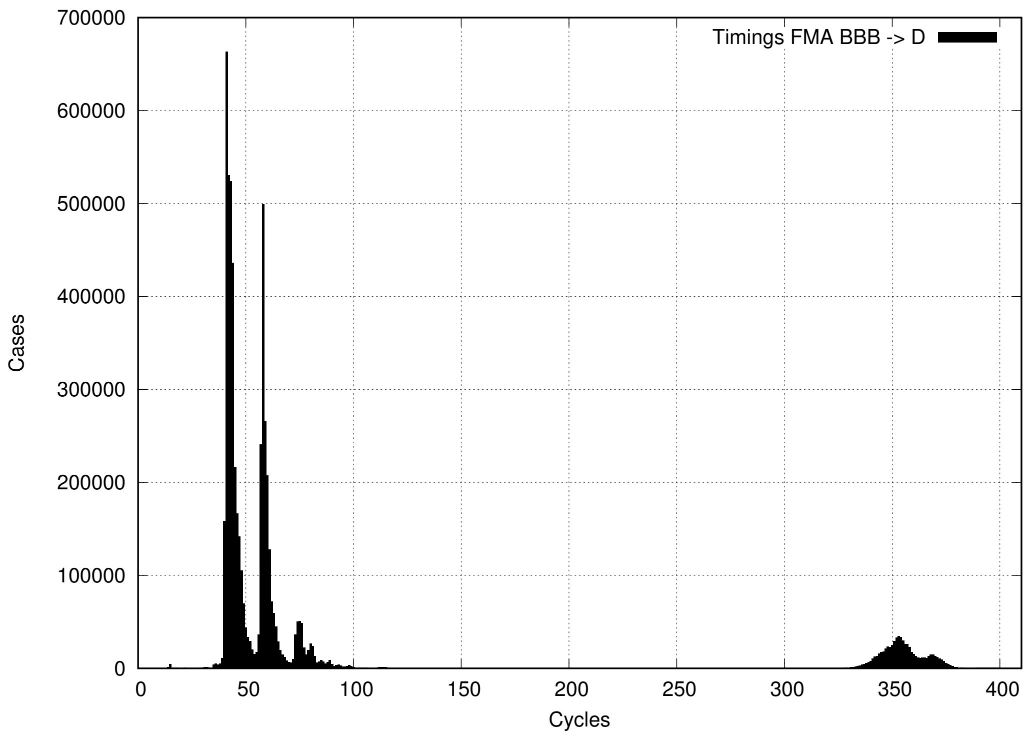

Figure 3.2 – Histogram of timing measurements for our implementation

For the sake of succinctness, we report only the performance testing results of one FMA function, called DBBB. This function takes all its arguments as IEEE754 binary64 numbers and returns a decimal64 number.

Figures 3.2 give the cycle count histograms for our DBBB FMA functions. It is easy to see that our implementations handle most cases in less than 50 cycles. Our implementations present a maximum call time of about 375 cycles. These call times correspond to cases when the rounding recovery phase gets used. As our algorithm is statically bounded in terms of memory consumption and makes no dynamic memory allocation, these maximum timing counts are hard bounds.

Figures 3.3 give the cycle counts for the GMP-based implementation for DBBB. For most cases, the GMP-based implementations is about 10 times slower than the implementation based on our algorithm presented in this section. The GMP-based maximum call times reach up to 46000 cycles in very rare cases, which is about 100 times slower than ours. This is due to the cost incurred by GMP's dynamical memory allocation mechanism.

### 3.4.5 Conclusion

In this Section, we have analyzed the feasibility of general computational mixed-radix operation. We have presented an algorithm to compute a correctly-rounded IEEE754 binary64 or decimal64 result to a mixed-radix Fused-Multiply-and-Add operation, taking any combination of IEEE754 binary64 and decimal64 numbers in argument. Our algorithm trivially allows for covering correctly rounded mixed-radix addition, subtraction and multiplication and is designed in a way that allows mixed-radix division and square root to be computed

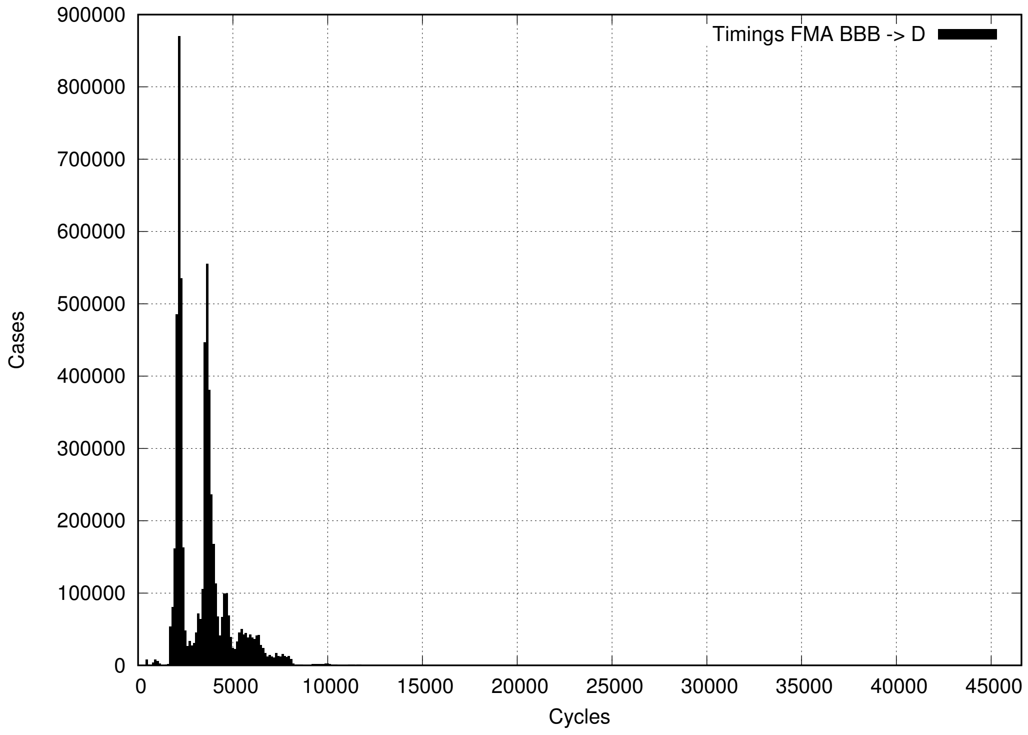

Figure 3.3 – Histogram of timing measurements for a reference implementation

with that FMA, too.

Our algorithm is based on a mixture of exact significand multiplication, finely precision-tuned radix conversion, binary addition and an exact rounding recovery phase that is launched, if rounding of an approximation would not allow a correctly rounded result to be computed. The recovery phase is based on the observation that all IEEE754 binary64 and decimal64 floating-point numbers and floating-point number mid-points are integer multiples of a fixed $2^Q \cdot 5^W$, with constant $Q, W \in \mathbb{Z}, Q, W \leqslant 0$.

Our implementation can handle most input cases in about $50$ machine cycles, which is a reasonable performance timing for a software-implemented floating-point operation. When the recovery phase gets used, latency stays below $400$ cycles. This bound is hard; our implementation does not require any dynamic memory allocation. It is hence suitable for integration into environments where dynamic memory management is not available.

Our implementation has been thoroughly tested against two other reference implementations, which we developed. Besides an issue with an undue IEEE754 inexact flag setting, which is outside our control, the test returns a *correct* verdict. A detailed pencil-and-paper proof for our algorithm has been written and will be made available under the form of a research report.

While we are convinced that our implementation is one of the first steps to paving the road to generalized mixed-radix floating-point arithmetic in IEEE754, quite a few points need to be addressed by future work:

— We subsumed all $14$ possible mixed-radix binary and decimal input/output format combinations for the FMA under one single core algorithm. It eased development and the correctness proof; it induces certain overhead due to exponent range over-estimation, though. Future work might perform a more detailed one-by-one analysis.

— We showcased our mixed-radix FMA for the IEEE754 formats binary64 and decimal64 only. All possible combinations taking into account also the binary32 and decimal128 formats need to be covered by future development.

— The necessity to fall back to exact arithmetic with large integer-based accumulators for hard-to-round cases is not very satisfactory. Future work should try to address the issue of precomputing worst-case precision bounds for mixed-radix FMA again.

## 3.5 Extended Precision Floating-Point Arithmetic

Extending IEEE754-2008 Standard with additional arithmetical functionality is not limited to providing operations that mix existing formats and radices of the Standard, as we did in the two Sections 3.3 and 3.4 that precede. We can also look into making floating-point arithmetic more precise, for example, to ease the development of precision-sensitive codes. In this Section, we are going to describe our approaches with respect to this aspect. The Section is based on [156].

### 3.5.1 Introduction

The IEEE754 Standard for Floating-Point (FP) Arithmetic defines several binary and decimal FP formats [117]. The basic ones, such as binary32 (single precision) and binary64 (double precision), are natively supported in hardware by current general-purpose processors. The IEEE754 binary128 (quad precision) format is supported in the current FP environment mainly through software emulation. The additional extended formats defined by IEEE754, with precisions beyond quad-precision, have no practical realization [117].

While binary32 and binary64 gear the majority of numerical software, some applications need support for precisions beyond quad-precision, i.e. with more than 113 bits of significand length. Examples for such applications include astronomy [79, 152, 153], computational physics [259] or experimental mathematics [11]. In some cases the use of extended precision might be overcome by subtle FP accuracy analysis and FP tricks [17,97]. The use of floating-point expansions, based on errorfree transformations [97,138,247,248], is also a possibility. In this approach, higher precision is obtained by representing accurate quantities by unevaluated sums of floating-point numbers [97,110,131,138,164,165,182,236,247,248]. As floating-point units are well optimized on current hardware, this approach may provide a good middle ground before using higher precision formats. Unfortuately, the use of expansion techniques often does not work “out of the box”. For this reason, some developers may simply prefer to speed up development with extended precision formats, used in some compute kernels, even if this means breaking a butterfly on a wheel [242].

As we will discuss in more detail below, such applications typically resort to arbitrary precision FP libraries, such as MPFR[25] or MPFUN[26]. These libraries are properly designed and optimized: they offer very good, if not optimal, performance in the precision range where arbitrary precision overhead becomes negligible [93]. This range typically starts at precisions

25. http://www.mpfr.org/
26. http://www.davidhbailey.com/dhbsoftware/

beyond a couple of hundreds to a thousand bits. This leaves a gap in the mid-precision range between quad-precision and a couple of hundred bits.

In this section, we propose the `libwidefloat` software designed to fill this gap. The package offers FP types with precisions between 64 and 512 bits, to be chosen at application compile time in 32 bit steps. Our FP types come with a 32 bit wide exponent, which essentially eliminates the need for support of subnormal numbers. They come with support for IEEE754-like infinities, Not-A-Numbers, rounding modes and FP flags [117]. The latter two features can be deactivated (resp. set to default round-to-nearest mode) at application compile time to allow for enhanced performance.

This Section is further organized as follows: in Section 4.4.1, we first present existing solutions to extended FP precision, analyzing the causes of their shortcomings in the mid-precision range. We then give an overview of modern compiler techniques that we will leverage to overcome these shortcomings (Section 3.5.3). In Section 3.5.4, we present our `libwidefloat` library, detailing the data-types and operations it supports. We briefly describe the algorithms used in the library in Section 3.5.5, give experimental results in Section 4.4.1, discuss future extension potential using C++ in Section 3.5.7 and conclude on wide-mantissa floating-point with Section 4.2.5.

### 3.5.2 Shortcomings of Existing Multiprecision Libraries

We have claimed that the existing arbitrary precision libraries, such as MPFR, are very well optimized for higher precisions (starting at about 1000 bits) but hindered by considerable overhead in the low- and mid-precision range. This overhead is due to several factors.

First, the very nature of arbitrary precision means that the length of the limb arrays that hold the FP significands of the different formats is not statically known. As a consequence, all variables having a type defined by one of these arbitrary precision libraries need not only be declared but also be initialized, using a call to a certain initialization function [86]. This function actually attributes a precision setting to the arbitrary precision variable and calls a memory allocation function such as `malloc` to provide space for the FP variable's significand. All accesses to a program's arbitrary precision variables' significands are in consequence indirect, requiring pointers to be followed onto the stack. Together with the dynamic nature of the significand length, this mostly prevents compilers from performing any reasonable optimization involving the arbitrary precision FP significands. In addition, for certain applications the requirement to allocate the arbitrary precision data on the memory heap and not hold them on the stack induces important overhead, as memory allocation may have significant cost with respect to computations for smaller precisions.

Second, the fact that existing arbitrary precision software comes packaged as software libraries adds to their overhead. While operations on IEEE754 binary32 and binary64 data-types are "inlined" as basic operations into the code, each and every use of an arbitrary precision operation involves a function call with all its cost in terms of register usage, variable spills, instruction cache misses etc. In addition, the correspondence between function names and the code behind them gets lost to optimization. Typically, when calling the arbitrary precision library, the compiler does not know of the functions' use cases and, when compiling the application code, it does not anything about the library functions but their name. Due to the use of assembly in the underlying libraries such as GMP[27], the arbitrary precision

27. see `https://gmplib.org/`

libraries even push this issue further to a point where the compiler can almost perform no optimization, even when using Link-Time-Optimization [86,95].

### 3.5.3 Leveraging Compiler Optimization Techniques

In order to take the observations described in the previous Section 4.4.1 into account, the following requirements to a software package for mid-precision FP arithmetic seem sensible.

First, the piece of software should not be packaged as a library to be linked in but should come in the form of code to be directly integrated into the application code. With modern compilers, such as `gcc`, this can be achieved by condensing the mid-precision FP library into header files that contain the functions provided by the software package declared `static` and `inline`. This way, the compiler[28] will be able to inline the FP functions into the application, propagating constants as far as possible, eliminating all unnecessary parts (such as NaN handling where no NaNs are possible) and optimizing loops by unrolling them.

Second, the software package should not use any micro-optimizations resorting to inline assembly or linking with objects written in assembly. While it is tempting for performance optimization at a function level, the use of assembly prevents compilers from propagating the information about the program it needs for optimization through the assembly sequence [95]. In most cases, modern compilers have sufficient support for doubled-precision integer arithmetic, in particular full $64 \times 64$ bit multiplication, through builtins (such as the `__uint128_t` type in `gcc`[29]), so that resorting to assembly is no longer necessary.

Finally, all wide integer and FP data-types to be declared by a FP software package optimized for the mid-precision range should be statically sized, in order to be held directly on the stack. In addition, the data should be properly aligned in order to avoid cache misses. As a matter of course, the endianess of wide integers, represented as limbs of a statically sized array should match the system's natural endianess, allowing the compiler to make use of up-converting and down-converting memory move operations [120]. In order to help the compiler, all accesses to these data-types involving pointers (or indexing in an array) should be done in a manner that allows for optimization. For instance, functions taking references on such data-types should be declared in a way that makes the compiler understand[30] that the pointers are not aliased (using the `restrict` keyword) and constant (using `const`) [122,123].

As a matter of course, the last requirement of using statically sized types in a FP mid-precision software package supporting more than one precision comes at the cost of a combinatorial explosion of functions to be provided. Indeed, if the ease of mixing FP types of different precisions that is provided by arbitrary precision libraries is to be maintained, where an arbitrary precision library provides one function for, say, addition, a software package with statically sized types has to provide several functions, one per combination of input and output types that is possible and sensible. In our work on `libwidefloat`, we have made extensive use of macro-programming and code generation to overcome this issue of combinatorial explosion.

28. see e.g. `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`
29. see e.g. `https://gcc.gnu.org/onlinedocs/gcc/_005f_005fint128.html`
30. see e.g. `https://gcc.gnu.org/onlinedocs/gcc/Restricted-Pointers.html`

### 3.5.4 The `libwidefloat` Library

The `libwidefloat` software was implemented with the shortcomings of the existing arbitrary precision libraries and modern compiler technology in mind. The software is implemented with two header files only, using modern C99/C11 [122, 123] but no assembly in order not to break the compiler's opportunities to optimize the code. All functions are declared in a way that allows them to be inlined and then optimized in their use context by the compiler.

The `libwidefloat` header starts by declaring statically sized data-types for `libwidefloat` FP values with precisions between 32 and 512 bits, by 32 bit increments. All these types support the basic IEEE754 FP data, such as positive and negative real numbers, signed zeros, signed infinities and Not-A-Number (NaN) data. The significand length is adapted in size according to the type's precision; the exponent width is kept at 32 bits for all precisions. Since this exponent width is pretty wide, we chose not to provide support for subnormal numbers in the `libwidefloat` types. The NaNs are signed but do not provide support for payload. They are all quiet NaNs; signaling NaNs are not supported.

Even though the `libwidefloat` types appear to be IEEE754-compliant from a programmer's perspective, their encoding is an *ad-hoc* one and does not respect the provisions of the IEEE754 Standard on extended precision types [117].

The `libwidefloat` software has provision for all five IEEE754 exception flags (inexact, invalid, overflow, underflow and division-by-zero) as well as for all four IEEE754 rounding modes (round-to-nearest-ties-to-even, round-down, round-up, round-towards-zero) [117]. The software can be configured at compile time in a way that allows this FP state, made of the flags and the rounding-mode, to be either

— held in a global variable,

— passed as a handle to any of the `libwidefloat` functions or

— omitted, assuming round-to-nearest as the rounding-mode and setting no flags.

The code is written in a way such that the compiler is able to eliminate all unnecessary computation steps that are not required when no flags are to be set and the rounding mode is statically set to round-to-nearest.

Currently, `libwidefloat` includes 1303 functions covering the following types of operations:

— Conversion between all combination of `libwidefloat` types. The up-conversions are exact, the down-conversions are correctly rounded.

— Conversion from and to the IEEE754 binary32 and binary64 formats [117]. These conversions are correctly rounded and signal all required exceptions, in particular inexact when rounding is necessary. These exceptions are signaled in and the rounding mode is taken from the FP environment of the type the conversion goes to. For instance, when converting from IEEE754 to `libwidefloat`, rounding is done according to the `libwidefloat` rounding mode and not the IEEE754 one.

— Conversion from and to signed and unsigned integer types. These conversions are provided in all variants (signaling inexact or not, taking the rounding mode from an immediate or from the FP state) that are defined by the IEEE754 Standard.

— Rounding to integral as defined by the IEEE754 Standard [117].

— Signaling and quiet comparisons, returning the comparison result as a enumerated type [117].

— Homogeneous-type absolute value and negation operations. The heterogeneous variants, where the operands' type differs from the one of the result can be provided through composition with a conversion operation [117].

— Operations to access the `libwidefloat` FP environment with respect to flags and rounding modes.

— And finally, Addition, Subtraction, Multiplication, Division, Fused-Multiply-And-Add (FMA) and Square Root with correct rounding. All these operations are provided in a homogeneous variant, where all operands and the result have the same type and in all heterogeneous variants, where the result is of a smaller type, such that all correctly rounded heterogeneous operations required by IEEE754 can be provided through composition with the conversion operations [117].

### 3.5.5 Algorithmic Choices in `libwidefloat`

There is no real novelty in the algorithms used in `libwidefloat`. In order to manage the combinatorial explosion due to support for the $16$ formats covering the precisions from $32$ to $512$ bits in homogeneous and heterogeneous operations, the final rounding step towards any of the supported formats has been factored into final rounding functions. These final rounding functions receive an intermediate result that has undergone already some rounding with respect to the mathematically exact result.

We shall hence explain the techniques used for intermediate and final rounding first, before detailing the algorithms behind multiplication, addition, FMA, division and square root.

Let us just note that `libwidefloat` also contains conversion functions from and to integer and the IEEE754 binary32 and binary64 formats. These conversions have been coded in *ad-hoc* ways, which are not described inhere.

#### Intermediate and Final Rounding

In order to provide correct rounding $\star(y) = \star(f(x))$ for a function $f$, an intermediate result $\widetilde{y}$ must be computed in such a way that $\star(\widetilde{y}) = \star(y)$ [197]. For all operations supported by `libwidefloat`, computing such an approximation $\widetilde{y}$ is pretty easy: it is either possible to compute the exactly representable result $f(x)$ (multiplication), or to compute a good approximation together with a flag indicating inexactness. Indeed, for division and square root, the backward error can be exactly computed and compared to zero to yield the inexactness predicate. For addition and FMA, it is just necessary to control the part of the significand that gets discarded during the alignment shift.

However, when computing the approximation $\widetilde{y}$ out of the exact result or a better approximation assorted with an inexact indication, pure truncation is not enough. Such a rounding scheme would suffer from the so-called double-rounding issue [21,197].

In order to overcome the double-rounding issue, `libwidefloat` uses rounding to odd for the intermediate results $\widetilde{y}$. This particular rounding mode is due to Boldo and Melquiond [21] and can be though of as a intermediate rounding longer than final precision where the last few additional bits represent the classical round, guard and sticky bits [77].

#### Multiplication

As already stated, multiplication in `libwidefloat` is based on an exact multiplication of the input significands, considered to be integer. At the current state of the library development, naive schoolbook multiplication using $\mathcal{O}(n^2)$ machine multiplications is used. The partial products are first represented in a carry-save format and then reduced [70].

The use of Karatsuba or Toom-Cook Multiplication [197] is left for future work but might prove necessary.

#### Addition, Subtraction and Fused-Multiply-Add

When implemented in software, Floating-Point Addition and Subtraction reduce to the same algorithm. The FMA operation can also share this part of code; the only difference is in whether one of the arguments to addition comes directly from one of the operands or from an intermediate exact product.

The core algorithm for addition and subtraction used in `libwidefloat` is a classical one [77,197] and very simple: after reordering of the FP operands for exponent, the significand of the lesser operand is shifted right for alignment, unless this shift would be larger than the output precision. The aligned significands are then added resp. subtracted. A normalization step then ensures correct construction of a FP significand in the case of elimination for subtractions. In software, due to overhead in branching and code bloat, it is sensible to run that normalization step in any case and not to distinguish between a near-path and a far-path.

#### Division and Square Root

The algorithms for division and square root are both based on an approximation to $1/\sqrt{z}$ in `libwidefloat`. We compute division $x/y$ as $x/(\sqrt{y})^2$ and square root $\sqrt{x}$ as $x \times 1/\sqrt{x}$.

An approximation to $1/\sqrt{x}$ can easily be computed using the Newton-Raphson iteration

$$y_{n+1} = y_n \cdot \frac{1}{2} \cdot \left(3 - x \cdot y_n^2\right)$$

seeded with a small table for $y_0$ [197]. In `libwidefloat` we use a $6$ bit approximation for $y_0$.

In an extended-precision software environment, it is useful not to perform the Newton iteration on software-simulated FP variables but purely on integer variables, using a fixed-point representation. We use this approach in `libwidefloat`. However, we chose to generate that code sequence instead of writing it manually in order not to have to determine all alignment shifts in that fixed-code point by hand.

### 3.5.6 Experimental Results

Our current implementation of `libwidefloat` provides $1303$ user-available functions. We have tested it for performance against MPFR 3.1.2-p3 based on GMP 6.0.0. We used `gcc` 4.9.2 on a 64bit Linux system featuring a 4 core Intel i7-5500U running at $2.4$ GHz. The optimization level was set to `-O3`.

There is no one-to-one relation between the functions provided by `libwidefloat` and those provided by MPFR. We therefore do not give performance results per function. We rather wrote two benchmark applications and compiled them against both `libwidefloat` and MPFR. The first benchmark application consists of a LU-decomposition, triangular system

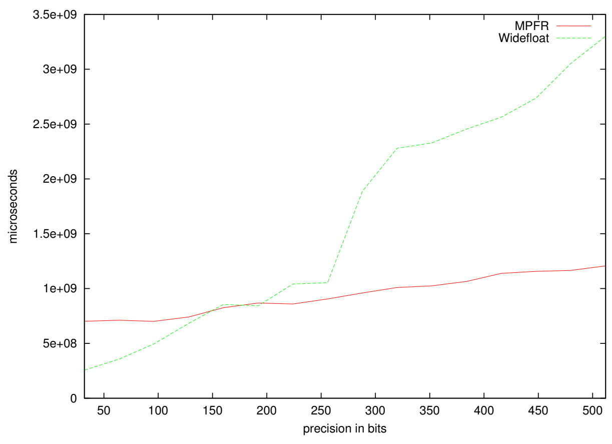

Figure 3.4 – Experimental results: Linear Algebra

solving and residue computation. The second one uses secant's method to approximate the zero of a high-degree polynomial.

All benchmark codes were run in all precisions supported by `libwidefloat` and compared to MPFR, which was run at the same precisions. While we focused on performance testing, we also compared the actual numerical results of `libwidefloat` and MPFR. On all benchmark codes and all precisions, the results were bitwise identical between both libraries, as expected, given that both libraries provide correct rounding.

We ran our LU-decomposition, triangular system solving and residue benchmark for matrix sizes $1000 \times 1000$. The matrices and respective right-hand sides were filled with random IEEE754 binary64 values between $-10^7$ and $10^7$. The time needed to allocate and free the different vectors and matrices was included in the performance testing. Matrix and vector-allocation for MPFR required calling the MPFR initialization procedures for each element of the matrix resp. vector; `libwidefloat` was not burdened by this requirement.

The results of the LU-decomposition benchmark are given in Figure 3.4. They show that `libwidefloat` performs well better than MPFR for small precisions but that starting with precisions around $224$ bits, the use of Karatsuba-multiplication helps in keeping MPFR's timings low, while `libwidefloat` exhibits quadratic behavior. We shall address this issue in our library in future releases.

In our second benchmark, we used secant's method to approximate a zero of a polynomial of degree $150$ up to the accuracy that was achievable within the given compute precision (which we made vary for the test). The polynomial was input as an expression tree and evaluated, using a recursive algorithm. This example is typical for numerical Computer Algebra Systems, such as Sollya[31] or Sage[32].

As shown with Figure 3.5, for that second expression-evaluation benchmark, `libwidefloat` typically performed better with respect to MPFR. However, our library is still hindered by the use of a naive multiplication algorithm for precisions higher than $320$ bits.

31. see `http://sollya.gforge.inria.fr/`
32. see `http://www.sagemath.org/`

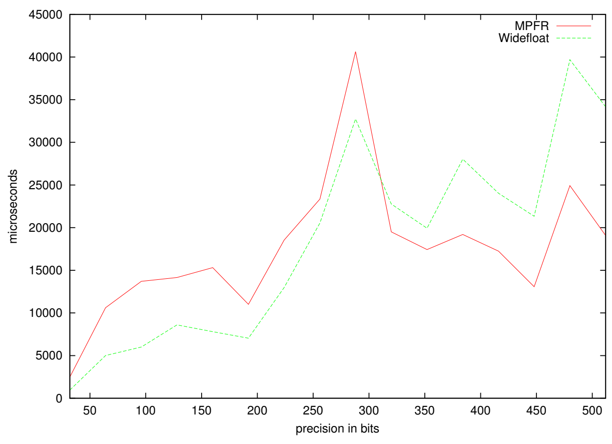

Figure 3.5 – Experimental results: Expression Evaluation

### 3.5.7 Enhancing Usability with C++

As we have just seen, our `libwidefloat` library provides 1303 user-available functions to provide support for 16 new formats covering the precisions from 32 to 512 bits in homogeneous and heterogeneous operations, along with support functions for conversion from and to the IEEE754-2008 formats binary32 and binary64. This large number of functions gives a wide variety of choice to the user but also makes their work of porting existing applications to wider formats really hard.

In order to alleviate this issue with `libwidefloat`, we worked together with an undergraduate student in order to come up with a C++ binding of `libwidefloat`. This C++ binding provides 16 classes for the different `libwidefloat` formats and defines methods for the infix operators $+$, $-$, $\star$, $/$, $=$ in a way that actually binds `libwidefloat`'s operation to these classes. The square root operation is provided as a `sqrt` method. All conversions from the IEEE754-2008 formats are provided as constructors, overloaded cast operators and as overloaded assignment operators. The conversions to IEEE754-2008 are implemented as casts and assignment operators. Some conversions from and to integers are also available.

In order to dominate the combinatorial explosion of possible combinations, template programming was first considered. The solution finally adopted is based on code generation again, though.

This C++ binding allows existing code to be ported easily to higher precisions. Some applications showcasing this ease in programming were developed by the student.

However, the C++ binding missed one of its initial goals: `libwidefloat` is written in a way that should allow a compiler to inline most of its operations. Ways to maintain this ability to be inlined exist for C++ libraries. The student's time being limited, the current design of the C++ binding breaks inlining support, which makes to calls to methods of the classes provided by this software pretty expensive with respect to the actual work done by the underlying `libwidefloat` operations. We shall continue working on improving this aspect in future work.

### 3.5.8 Conclusion

The `libwidefloat` software fills the gap between IEEE754 binary64 (or binary128 where available) and arbitrary precision libraries, which involve a certain amount of overhead in the mid-precision range. Our library leverages modern compilers' optimization techniques, such as constant propagation, code inlining and dead code elimination, to achieve this goal.

At the current state, `libwidefloat` supports all basic operations, like e.g. addition, $\mathrm{FMA}$, square root or comparisons. Development is ongoing: we are working on a C++ wrapper to allow for an easy drop-in-replacement of the IEEE754 binary64 format by `libwidefloat` formats. Further, we plan to use GNU's `printf`/`scanf` extension techniques to allow for easy input/output operations with `libwidefloat`. The development of the elementary intrinsic functions like the exponential, the logarithm or the trigonometric operations is also part of future work.

Experimentally, we see that `libwidefloat`'s performance still needs some improvement in the range between $256$ and $512$ bits. The current implementation of the library lacks sufficient performance due to the use of a naive (schoolbook) multiplication algorithm. We shall extend the library to use the Karatsuba or Toom-Cook multiplication algorithms in that higher precision range.

## 3.6 High-Level Operations: Faithful Rounding of 2-norms

In the last Section, we have seen an enhancement to IEEE754 in terms of additional formats, providing more precision than the classical, binary IEEE754 formats, such as binary32, binary64 or binary128.

In this Section, we shall now see how the IEEE754 Standard could also be extended: by adding support for high-level operations. Hence, the current gap between IEEE754 and standard libraries like BLAS [167] would level out over time. This Section is based on an article we published in [101].

### 3.6.1 Introduction

Computing the $l_2$-norm $\|\mathbf{x}\|_2 = \sqrt{\sum_{j=1}^{n} x_j^2}$ of a vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ is prevalent in scientific and engineering applications. This operation is part of the first (lowest) level of the Basic Linear Algebra Subroutine BLAS1. The formula $\sqrt{\sum_{j=1}^{n} x_j^2}$ is misleading in that it is not suitable for implementation, as the straightforward summing of squares can cause unwarranted (spurious) overflows or underflows, while $\|\mathbf{x}\|_2$ itself is well within the representable range of floating-point arithmetic. Common implementations such as the public version of LAPACK [8] released by `netlib`[33] essentially compute the $l_2$-norm as $\widehat{x} \times \|\mathbf{x}/\widehat{x}\|_2$ where $\widehat{x}$ is $\max_j |x_j|$. This computation requires $n$ divisions in total, which is significantly more expensive than the naive formula would suggest. Furthermore, in the worst-case scenario, the last $\log_{10}(n)$ digits of the result could be corrupted. Improving the accuracy of $l_2$-norm computation benefits computation not only in reduction of error, but also improving the chances of reproducible numerical results, should the $l_2$-norm computation be

33. `www.netlib.org`

done in parallel, with threads or vector-floating-point instructions [34].

In this Section, we present a new division-free $l_2$-norm algorithm that is amenable to straightforward parallel implementation. We prove that the algorithm always returns a faithfully rounded result, to be defined rigorously later. For now (and informally), this means that the result is accurate to within one bit of the underlying floating-point type. The algorithm is provably free from spurious overflow or underflow: that is, one of these exceptions is triggered only when the true value $\|\mathbf{x}\|_2$ calls for the event. Our implementation runs about $4.5$ times faster than the `netlib` version.

There are two main features of our algorithm. First, it accumulates the squares, $x_j^2$, using a pair of floating-point numbers, providing essentially the double of the underlying floating-point precision. Our technique is similar to the addition operator in the double-double library [178] but at almost twice the speed by exploiting the nonnegative nature of sum-of-squares. We provide rigorous analysis of the accuracy properties of our accumulation. Second, we eliminate all spurious exception by scaling the input data without using division. The technique is a "binning" method and is similar to the one proposed in [16] except [16] requires three bins and we use only two bins here, resulting in economy of registers usage and performance improvement. The accumulation and binning are amenable to straightforward parallel implementations. Our reference implementation uses data parallelism through SIMD instructions, but adding thread parallelism is straightforward.

We organize the rest of this Section as follows. Section 3.6.2 defines the key technical terms to make the Section self contained. We formulate our main problem and state our objective. We present and establish the main theorem in Section 3.6.3. The essence is that if $\|\mathbf{x}\|_2^2$ is computed to enough accuracy as a floating-point pair as "leading"-plus-"correction," then a standard IEEE-conforming square root on the "leading" part yields a faithfully-rounded $l_2$-norm. Section 3.6.4 presents a serial and a parallel $l_2$-norm algorithm without binning. In the absence of exceptions, we prove the numerical properties of these two algorithms that will lay the foundation for the actual binned algorithms that add the spurious-exception-free property. Section 3.6.5 presents the binned versions and the proofs of the faithful rounding and spurious-exception-free nature. Section 3.6.6 shows numerical and performance test results to corroborate with the theoretical analysis presented.

### 3.6.2 Background

Throughout this Section we consider a specific IEEE 754 binary floating-point type. Let $\mathbb{F}$ denote, in this Section, the entire set of finite values in this type, identified by three parameters $\varepsilon$, $e_{\min}$, and $e_{\max}$:

$$\mathbb{F} = \{\pm 2^{e+1} m\,\varepsilon \mid m \in \mathbb{N}, e_{\min} \leqslant e \leqslant e_{\max}, 0 \leqslant m\varepsilon < 1\}.$$

Remark that in contrast to our global definitions, which we gave in Section 3.1, we need to consider the boundedness of the exponent in a floating-point format in detail in this Section. For this reason, we slighlty change the notation behind $\mathbb{F}$.

For example, $(\varepsilon, e_{\min}, e_{\max})$ is $(2^{-24}, -126, 127)$ for binary32 format, and $(2^{-53}, -1022, 1023)$ for binary64 format. We assume $\varepsilon \leqslant 2^{-24}$ throughout this Section. Denote the smallest and

34. Clearly, a $l_2$-norm routine that returns the correctly rounded floating-point result is always reproducible.

largest positive normalized numbers by $\mathbb{F}_{\text{small}} = 2^{e_{\min}}$ and $\mathbb{F}_{\text{large}} = 2^{e_{\max}+1}(1-\varepsilon)$. For any real numbers $\alpha \in \mathbb{R}$, $|\alpha| \leqslant \mathbb{F}_{\text{large}}$, $\circ(\alpha)$ is the IEEE round-to-nearest-even function, $\circ : [-\mathbb{F}_{\text{large}}, \mathbb{F}_{\text{large}}] \to \mathbb{F}$. The rounding $\circ(\alpha)$ of a real number $\alpha$ is the floating-point number in $\mathbb{F}$ that is closest to $\alpha$, with a tie broken by choosing $\circ(\alpha)$ to have an even mantissa (least-significant bit being zero).

In order to define faithful rounding, we define the successor and predecessor functions:

$$\text{succ} : \{-\infty\} \cup \mathbb{F} \to \mathbb{F} \cup \{+\infty\}$$

where

$$\text{succ}(a) = \min\{\ b \in \mathbb{F} \cup \{+\infty\}\ \mid\ b > a\ \};$$

and

$$\text{pred} : \mathbb{F} \cup \{+\infty\} \to \mathbb{F} \cup \{-\infty\}$$

where

$$\text{pred}(a) = \max\{\ b \in \{-\infty\} \cup \mathbb{F}\ \mid\ b < a\ \}.$$

In this Section, we shall consider the ulp (units of last place) function to be defined for all real numbers $\alpha \in \mathbb{R}$, $|\alpha| \leqslant \mathbb{F}_{\text{large}}$ by

$$\text{ulp}(\alpha) = \begin{cases} 2^{e+1}\varepsilon & \text{if } |\alpha| \in [2^e, 2^{e+1}),\ e \geqslant e_{\min}, \\ 2^{e_{\min}+1}\varepsilon & \text{otherwise.} \end{cases}$$

This definition takes into account the existence of subnormal floating-point formats in the IEEE754 formats, which we often times consider apart from the main flow of the proofs. In this Section however, we intrinsically strive to manipulating input vectors with IEEE754 floating-point values that do contain subnormal numbers.

For any real number $\alpha$ and integer $k$ such that $\mathbb{F}_{\text{small}} \leqslant |\alpha|, |2^k\alpha| \leqslant \mathbb{F}_{\text{large}}$, $\text{ulp}(2^k\alpha) = 2^k\text{ulp}(\alpha)$, and $\text{ulp}(\alpha) \leqslant 2|\alpha|\varepsilon$ with $\text{ulp}(\alpha) = 2|\alpha|\varepsilon$ if and only if $\log_2(|\alpha|)$ is an integer.

It is easy to see that $\circ(\alpha)$ is at worst half a unit of last place away from $\alpha$: For a real number $|\alpha| \leqslant \mathbb{F}_{\text{large}}$, $|\alpha| \in [2^e, 2^{e+1}]$, $|\circ(\alpha) - \alpha| \leqslant 2^e\varepsilon$.

Given any $\alpha \in \mathbb{R}$, $|\alpha| \leqslant \mathbb{F}_{\text{large}}$, the faithful rounding, $\diamond(\alpha)$, of $\alpha$ is a set

$$\diamond(\alpha) = \{\ a \in \mathbb{F}\ \mid\ \text{pred}(a) < \alpha < \text{succ}(a)\ \}.$$

In particular, $\diamond(\alpha) = \{\alpha\}$ is the singleton containing $\alpha$, if $\alpha$ is already a floating-point number in $\mathbb{F}$. Otherwise, it contains just two elements – those closest to $\alpha$ from below and above.

This Section presents a parallelizable and division-free algorithm `AccuNrm2` such that for all vectors $\mathbf{x}$ of practical length, `AccuNrm2`$(\mathbf{x}) \in \diamond(\|\mathbf{x}\|_2)$ where $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, $x_j \in \mathbb{F}$, and $\|\mathbf{x}\|_2 \leqslant \mathbb{F}_{\text{large}}$. We wish to emphasize here that achieving faithful rounding is non-trivial. It can be shown that $\circ(\sqrt{\circ(\sigma)}) \in \diamond(\|\mathbf{x}\|_2)$, where $\sigma = \sum_j x_j^2 = \mathbf{x}^T\mathbf{x}$ is the exact sum of squares (or inner product). This says the correctly rounded square root of the correctly rounded sum of squares is a faithfully rounded $l_2$-norm. Nevertheless, computing the correctly rounded sums of squares is very expensive [201,231,232]. On the other hand, examples exist where $\circ(\sqrt{S}) \notin \diamond(\|\mathbf{x}\|_2)$ for some $S \in \diamond(\sigma)$. That is, computing the $\sigma$ to only slightly worse than the correctly rounded dot product can cause unfaithful rounding. Our algorithm in essence computes a floating-point number $S \approx \sigma$ very accurately and yet efficiently. Theorem 1 gives a condition on the accuracy of $S$ that guarantees $\circ(\sqrt{S})$ to be a faithful rounding of $\|\mathbf{x}\|_2$. The fact $\circ(\sqrt{\circ(\sigma)}) \in \diamond(\|\mathbf{x}\|_2)$ alluded to earlier follows trivially from Theorem 1 as well.

It is then important to accurately compute $\sigma = \sum_j x_j^2$. As we will see later, it is possible to transform this computation into a sum without loss of information (no rounding error). The problem is now transformed into the accurate computation of a sum. There is an abundant literature about floating-point summation (see [111, chap. 4], [138,201,228,231,232,262,263] and references thereof). For our purpose, we only need an accurate summation algorithm whose precision is doubled because the entries are nonnegative numbers. For such a precision, a straightforward adaptation of the algorithm `Sum2` [201] is very efficient since it requires $8(n-1)$ flops ($n$ being the size of the vector). The resulting sum has a relative error no more than on the order of $n^2\varepsilon^2$. Another choice is to use the double-double arithmetic [178]. The resulting sum is much more accurate, having a relative error no more than $2n\varepsilon^2/(1-2n\varepsilon^2)$. The cost, however, is $20(n-1)$ flops. The difference between the two algorithms comes from the "renormalization steps" that are present in the double-double library. As will be shown later, an error bound in the order of $n\varepsilon^2$ as opposed to $n^2\varepsilon^2$ is crucial if faithful rounding is to be guaranteed for a general vector length $n$. We therefore devise an algorithm that is faster than the addition operator in the double-double library. Instead of performing two renormalization steps in the addition of two double-double numbers, we perform only one renormalization step in a careful manner that allows us to deliver a sum whose relative error is bounded by $3n\varepsilon^2/(1-3n\varepsilon^2)$, at the cost of $11(n-1)$ flops, which is almost twice as fast as $20(n-1)$ flops.

A naive computation of the $l_2$-norm can cause spurious overflow and underflow. The netlib library addresses the overflow but not underflow. Spurious underflow can cause significant performance degradation. Moreover, the use of ordinary summation is not very accurate. Blue's work [16] uses 3 bins to address spurious exceptions, but the accuracy is comparable to the netlib version. Our algorithm now addresses both the problems of accuracy (faithful rounding) and spurious exceptions with better performance than netlib version.

We will state without proving the following elementary facts about floating-point arithmetic:

— Let $a, b \in \mathbb{F}$. The absolute error bound

$$|\circ(a \bullet b) - (a \bullet b)| \leqslant \mathrm{ulp}(a \bullet b)/2$$

holds for all $\bullet \in \{+, -, \times, \div\}$ provided $|a \bullet b| \leqslant \mathbb{F}_{\mathrm{large}}$. The relative error bound

$$\circ(a \bullet b) = (a \bullet b)(1+\delta), \quad |\delta| \leqslant \varepsilon$$

holds for $\bullet \in \{+, -\}$ as long as $|a \bullet b| \leqslant \mathbb{F}_{\mathrm{large}}$. For $\bullet \in \{\times, \div\}$, this relative error bound holds if $\mathbb{F}_{\mathrm{small}} \leqslant |a \bullet b| \leqslant \mathbb{F}_{\mathrm{large}}$.

— The notations $\oplus$, $\ominus$, $\otimes$, etc., stand for floating-point operations. For example, $c \leftarrow a \oplus b$ means $c = \circ(a+b)$. We distinguish between $\circ(a+b+c)$ and $a \oplus (b \oplus c)$; the former is rounding to nearest (once) the exact real number $a+b+c$ and the latter is $\circ(a+\circ(b+c))$.

— Rounding to nearest is monotonic. Given real numbers $\alpha, \beta \in [-\mathbb{F}_{\mathrm{large}}, \mathbb{F}_{\mathrm{large}}]$, $\alpha \leqslant \beta$ implies $\circ(\alpha) \leqslant \circ(\beta)$.

— For $a \in \mathbb{F}$, $0 \leqslant a < \mathbb{F}_{\mathrm{large}}$, $\mathrm{succ}(a) = a + \mathrm{ulp}(a)$.

- For $a, b \in \mathbb{F}$, $S = \circ(a+b) \in \mathbb{F}$, then $a + b - S \in \mathbb{F}$. In particular $s \overset{\text{def}}{=} a + b - S$ satisfies the relationship $S + s = a + b$ and $\circ(S + s) = S$. Furthermore, $S$ and $s$ can be computed from $a$ and $b$ by a sequence of instructions involving $\oplus$ and $\ominus$ (see for example [138] Thm B, page 236, and [71, 192]). We denote these by two functions $\mathrm{TwoSum}(a, b)$ and $\mathrm{FastTwoSum}(a, b)$. They deliver $S$ and $s$ where $S + s = a + b$ and $\circ(S + s) = S$. The former handles general $a$ and $b$ and requires 6 floating-point operations; the latter requires only 3 floating-point operations, but works only when $|a| \geqslant |b|$. Mapping $a, b$ to $S, s$ is usually called an error-free transformation as $a + b = S + s$ exactly.
- Similar to error-free transformations for sum, we have error-free transformations for product. For $a, b \in \mathbb{F}$, $|ab| \geqslant \mathbb{F}_{\text{small}}/\varepsilon$, and $P = \circ(a \times b) \in \mathbb{F}$, then $a \times b - P \in \mathbb{F}$. In particular $p \overset{\text{def}}{=} a \times b - P$ satisfies the relationship $P + p = a \times b$ and $\circ(P + p) = P$. $P$ and $p$ can be computed from $a$ and $b$ with the IEEE754-2008 fused multiply-add instruction: $P \leftarrow a \otimes b$ and $p \leftarrow \circ(a \times b - P)$. Alternatively, one can use a sequence of $\oplus$, $\ominus$ and $\otimes$ instructions as outlined in [71, 192]. We denote the exact product function that returns $P$ and $p$ by $\mathrm{TwoProd}(a, b)$.

### 3.6.3 Main Theorem

The goal is to compute $\|\mathbf{x}\|_2 = \sqrt{\sigma}$, $\sigma = \sum_{j=1}^{n} x_j^2$, faithfully. Our algorithm is built on the core case, when $\sigma$ is within the "normal" range of $[\mathbb{F}_{\text{small}}, \mathbb{F}_{\text{large}}]$. We first compute an accurate floating-point approximation, $S$, to $\sigma$. The IEEE square root (correctly rounded) $\circ(\sqrt{S})$ is returned as the final result. This section shows that if $S$ is accurate to within a specific threshold, the final result is faithful: $\circ(\sqrt{S}) \in \diamond(\sqrt{\sigma}) = \diamond(\|\mathbf{x}\|_2)$.

**Lemma 2.** *Let $\alpha, \alpha'$ be two real numbers in the interval $[2^e, 2^{e+1}]$ where $e_{\min} \leqslant e < e_{\max}$. If $|\alpha' - \alpha| < 2^e\varepsilon$, then correctly rounding $\alpha'$ is a faithful rounding of $\alpha$, that is, $\circ(\alpha') \in \diamond(\alpha)$.*

*Proof.* We observe that

$$\alpha' \in [2^e, 2^{e+1}] \implies a \overset{\text{def}}{=} \circ(\alpha') \in [2^e, 2^{e+1}] \cap \mathbb{F}.$$

Consequently,

$$|a - \alpha| \leqslant |a - \alpha'| + |\alpha' - \alpha| < \mathrm{ulp}(2^e).$$

Thus $\mathrm{succ}(a) \geqslant a + \mathrm{ulp}(2^e) > \alpha$. If $a \leqslant \alpha$, then $\mathrm{pred}(a) < \alpha$. Otherwise, $2^e < \alpha \leqslant 2^{e+1}$ and $\mathrm{pred}(a) = a - \mathrm{ulp}(2^e)$. Thus $\mathrm{pred}(a)$ is also strictly less than $\alpha$. As a result,

$$\mathrm{pred}(a) < \alpha < \mathrm{succ}(a)$$

and $a \in \diamond(\alpha)$ by definition. □

**Lemma 3.** *Let $\sigma \in [\mathbb{F}_{\text{small}}, \mathbb{F}_{\text{large}}]$ be a real number in the interval $[2^e, 2^{e+1})$ and $S, s \in \mathbb{F} \cap [\mathbb{F}_{\text{small}}, \mathbb{F}_{\text{large}}]$ where $\circ(S + s) = S$. If $|(S + s) - \sigma| < \sigma\varepsilon/2$, then $S \in [2^e, 2^{e+1}]$ and $|s| \leqslant 2^e\varepsilon$.*

*Proof.* From the assumptions,

$$S + s < 2^{e+1} + \frac{2^{e+1}\varepsilon}{2} = 2^{e+1} + \frac{\mathrm{ulp}(2^{e+1})}{4}.$$

Thus $\circ(S+s) \leqslant 2^{e+1}$. Similarly,

$$S+s > 2^e - \frac{2^e\varepsilon}{2} > 2^e - \frac{\text{ulp}(2^{e-1})}{2}.$$

Thus $\circ(S+s) \geqslant 2^e$. Consequently, $S \in [2^e, 2^{e+1}]$. To estimate $s$, consider the case $S+s \in [2^e, 2^{e+1}]$. In this case, $|s| = |\circ(S+s) - (S+s)| \leqslant 2^e\varepsilon$. Otherwise, $S+s \notin [2^e, 2^{e+1}]$ and $S \in [2^e, 2^{e+1}]$. Thus either $S = 2^e$ with $s < 0$, or $S = 2^{e+1}$ and $s > 0$. Moreover, summarizing the inequalities established so far, we have

$$2^e - \frac{2^e\varepsilon}{2} < S + s < 2^{e+1} + 2^e\varepsilon.$$

Thus either $0 < s < 2^e\varepsilon$ or $-2^e\varepsilon/2 < s < 0$, both implying $|s| < 2^e\varepsilon$, and the proof is complete. □

**Theorem 1.** *Let* $\sigma \in [\mathbb{F}_{\text{small}}, \mathbb{F}_{\text{large}}]$ *be a real number and* $S, s \in \mathbb{F}_\cap[\mathbb{F}_{\text{small}}, \mathbb{F}_{\text{large}}]$ *where* $\circ(S+s) = S$. *If* $|(S+s) - \sigma| < \varepsilon\sigma/8$, *then* $\circ(\sqrt{S}) \in \diamond(\sqrt{\sigma})$.

*Proof.* We establish this theorem by showing $\sqrt{\sigma}$ and $\sqrt{S}$ satisfy the conditions for $\alpha$ and $\alpha'$ in Lemma 2.

$$\begin{aligned} |(S+s)-\sigma| &< \varepsilon\sigma/8 \quad \text{implies} \\ S+s &> (1-\varepsilon/8)\sigma, \\ S &> (1-\varepsilon)(1-\varepsilon/8)\sigma, \\ &> (1-3\varepsilon/2)\sigma, \\ \sqrt{S} &> (1-3\varepsilon/2)\sqrt{\sigma}. \end{aligned} \tag{3.48}$$

The last equation holds as $\sqrt{1-3\varepsilon/2} \geqslant 1 - 3\varepsilon/2$. We can estimate $|\sqrt{S} - \sqrt{\sigma}|$ by

$$\begin{aligned} |\sqrt{S} - \sqrt{\sigma}| &= \frac{|S-\sigma|}{\sqrt{S}+\sqrt{\sigma}}, \\ &< \frac{\varepsilon\sigma/8 + |s|}{2\sqrt{\sigma}(1-3\varepsilon/4)}, \quad \text{using Equation (3.48)}, \\ &< \left(\frac{\varepsilon}{16}\sqrt{\sigma} + \frac{|s|}{2\sqrt{\sigma}}\right)(1+\varepsilon). \end{aligned} \tag{3.49}$$

The value $\sigma$ must lie in a unique interval of the form $\sigma \in [2^{2e}, 2^{2e+2})$, corresponding to $\sqrt{\sigma} \in [2^e, 2^{e+1}) \subset [2^e, 2^{e+1}]$. By Lemma 3, $S \in [2^{2e}, 2^{2e+2}]$ and thus $\sqrt{S} \in [2^e, 2^{e+1}]$. By virtue of Lemma 2, the theorem here is established provided $|\sqrt{S} - \sqrt{\sigma}| < 2^e\varepsilon$. A simple tabulation suffices.

| $\sigma/2^{2e}$ | $\sqrt{\sigma}/2^e$ | $\lvert s\rvert/2^{2e}$ | $\lvert\sqrt{S}-\sqrt{\sigma}\rvert/(2^e\varepsilon)$ |
|---|---|---|---|
| $\in [1,2)$ | $\in [1,\sqrt{2})$ | $\leqslant \varepsilon$ | $< \left(\frac{\sqrt{2}}{16} + \frac{1}{2}\right)(1+\varepsilon)$ |
| $\in [2,4)$ | $\in [\sqrt{2},2)$ | $\leqslant 2\varepsilon$ | $< \left(\frac{1}{8} + \frac{1}{\sqrt{2}}\right)(1+\varepsilon)$ |

Clearly, $|\sqrt{S} - \sqrt{\sigma}| < 2^e\varepsilon$ always and $\circ(\sqrt{S}) \in \diamond(\sqrt{\sigma})$ as claimed. □

### 3.6.4 Core Case

Let $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$ be the vector in question. The core algorithm considers the case of normal range in the sense that $\sigma = \mathbf{x}^T\mathbf{x}$ is safely away from overflow and that the least significant bit of each of the $x_j^2$ does not underflow. More formally, we consider $\mathbb{F}_{\text{small}}/\varepsilon^2 \leqslant x_j^2 \leqslant \mathbb{F}_{\text{large}}$ for all $j$ and $\sigma \leqslant \mathbb{F}_{\text{large}}/2$.

We use a data type, double-FP, consisting of two floating-point numbers. We denote them, for example, by $\mathbf{A} = [A, a]$. $\mathbf{A}$ is the double-FP variable, and the actual pairs of floating-point numbers are $A$ and $a$. They have the characteristics $\circ(A+a) = A$, which means that $a$ is a "tail" part to add extra precision to $A$. The mathematical sum of the two components represents a value of at least twice the precision of the underlying floating-point number. Here is the core algorithm for computing the sum of squares $\sum_j x_j^2$.

```
function SumOfSquares($\mathbf{x}$) // Accurate accumulation
    $\mathbf{S} \leftarrow [0,0]$
    for $j = 1, 2, \ldots, n$ do:
        $\mathbf{P} \leftarrow \text{TwoProd}(x_j, x_j)$
        // $\mathbf{P} = [P,p]$, $P + p = x_j^2$ exactly
        $\mathbf{S} \leftarrow$ SumNonNeg($\mathbf{S}, \mathbf{P}$)
    return S
end SumOfSquares
```

```
function SumNonNeg($\mathbf{A}, \mathbf{B}$) // $[A,a] + [B,b]$
// error bound of this operation $3\varepsilon^2$ (Theorem 2)
// $\mathbf{A} = [A,a], \mathbf{B} = [B,b]$ nonnegative: $A + a, B + b \geqslant 0$
    $\mathbf{H} \leftarrow \text{TwoSum}(A, B)$
    // $\mathbf{H} = [H,h]$, $H + h = A + B$ exactly
    $c \leftarrow a \oplus b$       // $c = a + b + \delta_c$
    $d \leftarrow h \oplus c$       // $d = h + c + \delta_d$.
    $\mathbf{S} \leftarrow \text{FastTwoSum}(H, d)$
    // $\mathbf{S} = [S,s]$, $S + s = H + d$ exactly
    //see Section 3.6.2 for TwoSum and FastTwoSum.
    return S
end SumNonNeg
```

Theorem 1 guarantees that if $[S, s]$ returned by `SumOfSquares` satisfies $|(S + s) - \sigma| < \varepsilon\sigma/8$, then $\circ(\sqrt{S})$ is a faithful rounding of $\sqrt{\sigma} = \|\mathbf{x}\|_2$. Since `SumOfSquares` is summing $n$ double-FP type, standard error analysis (see for example Chapter 3 of [111]) shows that the relative error is bounded by $\Delta_{n-1}(\delta)$ where $\Delta_\ell(\delta) = \ell\delta/(1 - \ell\delta)$ and $\delta$ is the relative error bound on the underlying addition operation, which is the `SumNonNeg` function. Theorem 2 shows that this $\delta$ is $3\varepsilon^2$. From that we can deduce the length limit of $\mathbf{x}$ within which $|(S + s) - \sigma| < \varepsilon\sigma/8$.

**Theorem 2.** *Let* $\mathbf{S} = [S, s]$ *be the result from applying* `SumNonNeg` *on nonnegatives* $\mathbf{A} = [A, a]$ *and* $\mathbf{B} = [B, b]$. *Let* $\alpha = A + a \geqslant 0$, $\beta = B + b \geqslant 0$ *denote the exact input values, and* $\sigma = \alpha + \beta$ *denote the exact sum. Then* $|(S + s) - \sigma| \leqslant 3\varepsilon^2\sigma$.

*Proof.* The theorem clearly holds if one of $\alpha$ or $\beta$ is zero. Moreover, it is clear that `SumNonNeg` is insensitive to the order of its two input arguments. It suffices therefore to consider $\alpha \geqslant \beta > 0$. There are only two rounding errors in the entire function: $\delta_c = c - (a+b) = \circ(a+b) - (a+b)$ and $\delta_d = d - (h + c) = \circ(h + c) - (h + c)$. More precisely, $S + s = H + d$ and

$$H + d = H + h + a + b + \delta_c + \delta_d = \sigma + \delta_c + \delta_d.$$

Thus $|(S+s)-\sigma| \leqslant |\delta_c| + |\delta_d|$. The rest of the proof establishes the fact $|\delta_c| + |\delta_d| \leqslant 3\varepsilon^2\sigma$.

We use this fact heavily: For any real number $\gamma$ where $\gamma \in \mathbb{R} \cap [\mathbb{F}_{\text{small}}, \mathbb{F}_{\text{large}}]$, $|\circ(\gamma) - \gamma| \leqslant \text{ulp}(\gamma)/2$. Let

$$\alpha = 2^{e_\alpha}(1+f_\alpha),\ \beta = 2^{e_\beta}(1+f_\beta),\ \text{ and }\ \sigma = 2^{e_\sigma}(1+f_\sigma)$$

where $0 \leqslant f_\alpha, f_\beta, f_\sigma < 1$.

$$\sigma \geqslant \alpha \geqslant \beta \implies \text{ulp}(\sigma) \geqslant \text{ulp}(\alpha) \geqslant \text{ulp}(\beta).$$

Because $|a| \leqslant \text{ulp}(\alpha)/2$, $|a| \leqslant \text{ulp}(\sigma)/2$. Similarly, $|b| \leqslant \text{ulp}(\sigma)/2$. Therefore, $A+B = \sigma - (a+b) \leqslant \sigma + \text{ulp}(\sigma)$, which implies $\text{ulp}(A+B) \leqslant 2\text{ulp}(\sigma)$.

We note that $|\delta_c| = |\circ(a+b) - (a+b)| \leqslant \text{ulp}(a+b)/2$ and $|a+b| \leqslant \text{ulp}(\sigma)$. But $|a+b| = \text{ulp}(\sigma)$ only when $|a| = |b| = \text{ulp}(\sigma)/2$, which implies $a+b$ is representable exactly in $\mathbb{F}$ and $\delta_c = 0$. When $|a+b| < \text{ulp}(\sigma)$, we have $\text{ulp}(a+b) \leqslant \text{ulp}(\sigma/2)$. Hence

$$|\delta_c| \leqslant \frac{1}{2}\text{ulp}(\text{ulp}(\sigma/2)) = \frac{\text{ulp}(\sigma)}{2}\varepsilon \leqslant \sigma\varepsilon^2, \tag{3.50}$$

using basic properties of the ulp function stated in Section 3.6.2.

We now show that $|\delta_d| \leqslant 2\sigma\varepsilon^2$.

$$\begin{aligned} |\delta_d| &= |\circ(h+c) - (h+c)|, \\ &\leqslant \frac{1}{2}\text{ulp}(h+c), \\ &\leqslant \frac{1}{2}\text{ulp}(\text{ulp}(A+B)/2 + |c|), \\ &\leqslant \frac{1}{2}\text{ulp}(\text{ulp}(\sigma) + |c|). \end{aligned} \tag{3.51}$$

To complete the estimate on $|\delta_d|$, we analyze $|c|$. There are only two possibilities: $e_\alpha = e_\beta$ or $e_\alpha \geqslant e_\beta + 1$. We show that either situation leads to $|\delta_d| \leqslant 2\sigma\varepsilon^2$.

Consider the case of $e_\alpha \geqslant e_\beta + 1$. We have $|a| \leqslant \text{ulp}(\sigma)/2$ and $|b| \leqslant \text{ulp}(\sigma)/4$. Therefore,

$$|c| = |\circ(a+b)| \leqslant |a+b|(1+\varepsilon) \leqslant \frac{3}{4}(1+\varepsilon)\text{ulp}(\sigma).$$

Equation (3.51) implies

$$\begin{aligned} |\delta_d| &\leqslant \frac{1}{2}\text{ulp}(\text{ulp}(\sigma) + \frac{3}{4}(1+\varepsilon)\text{ulp}(\sigma)), \\ &\leqslant \frac{1}{2}\text{ulp}(\text{ulp}(\sigma)), \\ &= 2\sigma\varepsilon^2. \end{aligned} \tag{3.52}$$

Consider the case of $e_\alpha = e_\beta$. In this situation, we must have $e_\sigma = e_\alpha + 1$ and $\text{ulp}(\alpha) = \text{ulp}(\beta) = \text{ulp}(\sigma)/2$. As a result, $|a| + |b| \leqslant \text{ulp}(\sigma)/2$ and $|c| \leqslant (1+\varepsilon)\text{ulp}(\sigma)/2$, so

$$\begin{aligned} |\delta_d| &\leqslant \frac{1}{2}\text{ulp}(\text{ulp}(\sigma) + \frac{1}{2}(1+\varepsilon)\text{ulp}(\sigma)), \\ &\leqslant \frac{1}{2}\text{ulp}(\text{ulp}(\sigma)), \\ &= 2\sigma\varepsilon^2. \end{aligned} \tag{3.53}$$

Equations (3.50), (3.52) and (3.53) together show that $|\delta_c| + |\delta_d| \leqslant 3\sigma\varepsilon^2$, and the theorem is proved. $\square$

**Theorem 3.** *Let $n$ be the length of the vector $\mathbf{x}$, $x_j^2 \in [\mathbb{F}_{\text{small}}/\varepsilon^2, \mathbb{F}_{\text{large}}/2]$ for all $j$, and $\sigma \leqslant \mathbb{F}_{\text{large}}/2$ where $\sigma = \sum_j x_j^2$. Let* `SumOfSquares`$(\mathbf{x})$ *return the result $[S, s]$. Then*

$$|(S + s) - \sigma| \leqslant \Delta_{n-1}(3\varepsilon^2)\sigma$$

*where $\Delta_\ell(\delta) = \ell\delta/(1 - \ell\delta)$. In particular, if the length $n$ satisfies $n < ((24 + \varepsilon)\varepsilon)^{-1}$, then*

$$|(S + s) - \sigma| < \varepsilon\sigma/8.$$

*Proof.* From Theorem 2 and standard error bound on adding $n$ nonnegative floating-point types [111] with an addition operation of relative error bounded by $3\varepsilon^2$,

$$|(S + s) - \sigma| \leqslant \Delta_{n-1}(3\varepsilon^2).$$

Because $\Delta_\ell(\delta)$ is an increasing function in $\ell$ in the range $0 \leqslant \ell < 1/\delta$, for $n < ((24 + 3\varepsilon)\varepsilon)^{-1}$,

$$\Delta_{n-1}(3\varepsilon^2) < \Delta_n(3\varepsilon^2) < \Delta_L(3\varepsilon^2) = \frac{\varepsilon}{8},$$

where $L = ((24 + 3\varepsilon)\varepsilon)^{-1}$. $\square$

We parallelize `SumOfSquares` in an obvious manner: partition the input vector $\mathbf{x}$ to $\tau$ subvectors of roughly equal length. Perform the sum of squares on each subvector in parallel. This parallelism can be realized either at the thread level or data level, the latter using vector instructions such as SSE. The partial sums of squares are then accumulated in a serial manner.

```
function SumOfSquaresP($\mathbf{x}$) // Parallel SumOfSquares
    Partition $\mathbf{x}$ into $\tau$ portions, $\mathbf{x}^{(t)}$, $t = 1, 2, \ldots, \tau$
    // length of each $\mathbf{x}^{(t)}$ is no more than $m = \lceil n/\tau \rceil$.
    $\mathbf{S}^{(t)} \leftarrow$ SumOfSquares($\mathbf{x}^{(t)}$),    $t = 1, 2, \ldots, \tau$.
    // In parallel, each $\mathbf{S}^{(t)} = [S^{(t)}, s^{(t)}]$ is a double-FP.
    $\mathbf{S} \leftarrow [0, 0]$;   $\mathbf{S} \leftarrow$ SumNonNeg($\mathbf{S}, \mathbf{S}^{(t)}$),    $t = 1, 2, \ldots, \tau$.
    // In serial, summing the $\tau$ partial sums of squares
    // $\mathbf{S} = [S, s]$ at this point; $S + s \approx \sum_j^n x_j^2$.
    return S
end SumOfSquaresP
```

**Theorem 4.** *Let $n$ be the length of $\mathbf{x}$ and $\mathbf{S} = [S, s]$ be the result of* `SumOfSquaresP`$(\mathbf{x})$ *with $\tau$ portions and $m = \lceil n/\tau \rceil$. Then*

$$|(S + s) - \sigma| \leqslant \Delta_{m+\tau}(3\varepsilon^3)\sigma.$$

*In particular,*

$$|(S + s) - \sigma| \leqslant \Delta_{n-1}(3\varepsilon^3)$$

*whenever $m + \tau \leqslant n - 1$.*

*Proof.* We document the two stages of errors with the following notations. Let $\sigma^{(t)} = (\mathbf{x}^{(t)})^T\mathbf{x}^{(t)}$, $t = 1, 2, \ldots, \tau$, and $\sigma = \sum_{t=1}^{\tau}\sigma^{(t)}$ denote the exact partial and exact complete inner products, respectively. Let

$$\begin{array}{ll} \widetilde{\sigma}^{(t)} = S^{(t)} + s^{(t)}, & \text{approx partials } 1 \leqslant t \leqslant \tau, \\ \widetilde{\sigma} = \sum_{t=1}^{\tau}\widetilde{\sigma}^{(t)}, & \text{exact sum of approx partials,} \\ \widetilde{\widetilde{\sigma}} = S + s, & \text{approx sum of approx partials.} \end{array}$$

For the computed partials, where we are summing no more than $m = \lceil n/\tau \rceil$ double-FP types, we have

$$|\widetilde{\sigma}^{(t)} - \sigma^{(t)}| \leqslant \Delta_{m-1}(3\varepsilon^2)\,\sigma^{(t)}, \quad t = 1, 2, \ldots, \tau,$$

and

$$\left|\sum_{t=1}^{\tau}(\widetilde{\sigma}^{(t)} - \sigma^{(t)})\right| \leqslant \Delta_{m-1}(3\varepsilon^2)\sum_{j=1}^{\tau}\sigma^{(t)}.$$

This implies

$$\begin{array}{rcl} |\widetilde{\sigma} - \sigma| & \leqslant & \Delta_{m-1}(3\varepsilon^2)\,\sigma, \qquad (3.54) \\ \widetilde{\sigma} & \leqslant & (1 + \Delta_{m-1}(3\varepsilon^2))\,\sigma. \qquad (3.55) \end{array}$$

Similarly,

$$|\widetilde{\widetilde{\sigma}} - \widetilde{\sigma}| \leqslant \Delta_{\tau-1}(3\varepsilon^2)\,\widetilde{\sigma}. \qquad (3.56)$$

Combining Equations (3.54) through (3.56),

$$\begin{array}{rcll} \dfrac{|\widetilde{\widetilde{\sigma}} - \sigma|}{\sigma} & \leqslant & \Delta_{\tau-1}(3\varepsilon^2)\,(\widetilde{\sigma}/\sigma) + \Delta_m(3\varepsilon^2), & \\ & \leqslant & \Delta_{\tau-1}(3\varepsilon^2)\,(1 + \Delta_{m-1}(3\varepsilon^2)) + \Delta_m(3\varepsilon^2), & \\ & \leqslant & \Delta_\tau(3\varepsilon^2) + \Delta_m(3\varepsilon^2), & (3.57) \\ & \leqslant & \Delta_{m+\tau}(3\varepsilon^2). & (3.58) \end{array}$$

Equation (3.57) follows from its preceding line as long as $3\varepsilon^2 \leqslant (n + m + 2)^{-1}$, and Equation (3.58) follows from Equation (3.57) because $\Delta_\ell(3\varepsilon^2) + \Delta_{\ell'}(3\varepsilon^2) \leqslant \Delta_{\ell+\ell'}(3\varepsilon^2)$. Finally, whenever $m + \tau \leqslant n - 1$, $\Delta_{m+\tau}(3\varepsilon^2) \leqslant \Delta_{n-1}(3\varepsilon^2)$. The proof is now complete. □

We remark that for any non-trivial length $n$ and practical value of $\tau$, $m + \tau \leqslant n - 1$ always as $m + \tau \approx n/\tau$. This says that a parallel sum of squares is in general more accurate than the serial version. In particular, as long as $n < ((24 + \varepsilon)\varepsilon)^{-1}$ and $[S, s]$ is obtained from with `SumOfSquares` or `SumOfSquaresP`, an IEEE conforming square root evaluation sqrt($S$) produces a faithfully rounded $\|\mathbf{x}\|_2$ for $\mathbf{x}$, whose elements fall in the core range discussed here.

### 3.6.5 General Case

That the simple accumulation of $\sigma = \sum_{j=1}^{n} x_j^2$ is susceptible to spurious exceptions can be illustrated by the simple example of $n = 8$, $x_j = \circ(2\sqrt{\mathbb{F}_{\text{large}}})$ for $j \leqslant 4$ and $x_j = \circ(\sqrt{\mathbb{F}_{\text{small}}}/2)$, $j > 4$. While $\|\mathbf{x}\|_2$ is approximately $4\sqrt{\mathbb{F}_{\text{large}}}$, overflows in computing $x_j^2$ for $j \leqslant 4$ leads to a computed $\sigma$ of $+\infty$, rendering the final computed $l_2$-norm completely wrong. This is why general-purpose software such as LAPACK's public release essentially computes $\widehat{x}\sqrt{\sum_j (x_j/\widehat{x})^2}$ where $\widehat{x} = \max_j |x_j|$. This strategy indeed eliminates all spurious overflows and ensures a correct numerical result. This algorithm, however, can still trigger spurious underflows, which, while harmless to the final numerical result, would significantly degrade performance on computing platforms that handle underflow via a trapping mechanism.

We present here an algorithm that computes $\|\mathbf{x}\|_2$ faithfully without any spurious exceptions. Let $\sigma = \sum_j x_j^2$ denote the sum of squares. Our algorithm computes $Z$, a faithful rounding of a scaled $l_2$-norm $\|\widehat{\mathbf{x}}\|_2 = \gamma^{-m/2}\|\mathbf{x}\|_2$, that is $Z \in \diamond(\|\widehat{\mathbf{x}}\|_2)$. The factor $\gamma^{-m/2}$ is chosen so that $\gamma^m$ is an even power of 2, $\gamma^{-m/2} \in \mathbb{F}$, and $\|\widehat{\mathbf{x}}\|_2 \in [\mathbb{F}_{\text{small}}, \mathbb{F}_{\text{large}}]$. The final result is returned, naturally, as $\gamma^{m/2} \otimes Z$. If this multiplication generates an overflow or underflow, the exception is non-spurious. In short, the two parts of our algorithm are computing $Z$ and computing $\gamma^{m/2} \otimes Z$. While the latter seems trivial, it actually requires analysis as we need to guarantee $\gamma^{m/2} \otimes Z \in \diamond(\|\mathbf{x}\|_2)$ given $Z \in \diamond(\|\widehat{\mathbf{x}}\|_2)$ – even if the last multiplication generates an exception. We address this second part of the algorithm in the next theorem and devote the remaining section to the computation of $Z$.

**Theorem 5.** *Let $\zeta \in [\mathbb{F}_{\text{small}}, \mathbb{F}_{\text{large}}] \cup \{0\}$ be a real number and $Z$ is a faithful rounding of $\zeta$, that is, $Z \in \diamond(\zeta)$. Then $\circ(2^t Z) \in \diamond(2^t\zeta)$ for any scaling $2^t \in \mathbb{F}$, $t$ being an integer.*

*Proof.* The theorem holds if $\zeta$ is a floating-point point number to begin with. This is because $\diamond(\zeta) = \{\zeta\}$, $Z = \zeta$, and correctly rounding $2^t\zeta$ is a faithful rounding of $2^t\zeta$.

It suffices to consider $\zeta \notin \mathbb{F}$. In particular, $\mathbb{F}_{\text{small}} < \zeta < \mathbb{F}_{\text{large}}$. As $\zeta$ is not a floating-point number and $\mathbb{F}_{\text{small}} < \zeta < \mathbb{F}_{\text{large}}$, $\diamond(\zeta)$ has exactly two floating-point numbers, $a$ and $b$, where

$$\mathbb{F}_{\text{small}} \leqslant a < \zeta < b \leqslant \mathbb{F}_{\text{large}}, \; b = \text{succ}(a) = a + \text{ulp}(a).$$

Represent $a$ as $a = 2^{e_a} + 2^{e_a} m \text{ulp}(2^{e_a})$, $0 \leqslant m \leqslant \varepsilon^{-1} - 1$. In particular, $b \in [2^{e_a}, 2^{e_a+1}]$. Moreover, $Z$ is either $a$ or $b$. Since the theorem is trivially true for $2^t = 1$, it suffices to consider the two cases of $t < 0$ and $t > 0$.

Consider $t < 0$. If $e_a + t \geqslant e_{\min}$, $2^t a$ and $2^t b$ are both floating-point numbers, $2^t a < 2^t\zeta < 2^t b$ and $2^t b = \text{succ}(2^t a)$. Therefore $\circ(2^t Z) \in \{\circ(2^t a), \circ(2^t b)\} = \diamond(2^t\zeta)$. In case when $e_a + t < e_{\min}$, because $\circ(2^t a) \leqslant \circ(2^t\zeta) \leqslant \circ(2^t b)$ by monotonicity of correct rounding, we either have $\circ(2^t a) = \circ(2^t b)$ or $0 \leqslant \circ(2^t a) < \circ(2^t b)$. In the former case, $\circ(2^t a) = \circ(2^t\zeta) = \circ(2^t b)$ and thus $\circ(2^t Z) = \circ(2^t a) = \circ(2^t b) \in \diamond(2^t\zeta)$. In the latter case, we note that $\circ(2^t b) \leqslant 2^{e_{\min}}$. This is because $b \in [2^{e_a}, 2^{e_a+1}]$. Let $u = 2^{e_{\min}+1}\varepsilon$ denote the unit of last place in the underflow region $[0, 2^{e_{\min}}]$. Both $a$ and $b$ are integer multiples of $u$, and $|\circ(2^t a) - 2^t a|, |\circ(2^t b) - 2^t b| \leqslant u/2$ imply

$$\begin{aligned} |\circ(2^t a) - \circ(2^t b)| &\leqslant u + |2^t(b-a)|, \\ &\leqslant u + 2^{e_a+t+1}\varepsilon, \\ &\leqslant 3u/2. \end{aligned}$$

Thus $|\circ(2^t a) - \circ(2^t b)| = u$ and there are no floating-point numbers between $\circ(2^t a)$ and $\circ(2^t b)$.

Note that

$$\begin{array}{rcl} |\circ(2^t a) - 2^t\zeta| & \leqslant & |\circ(2^t a) - 2^t a| + 2^t|a-\zeta| \\ & < & \dfrac{u}{2} + 2^{e_a+t+1}\varepsilon \\ & < & u. \end{array}$$

Similarly, $|\circ(2^t b) - 2^t\zeta| < u$. Hence, we must have $2^t\zeta$ in between $\circ(2^t a)$ and $\circ(2^t b)$ and once again $\circ(2^t Z) \in \{\circ(2^t a), \circ(2^t b)\} = \diamond(2^t\zeta)$.

The case of $t > 0$ is simple. If $2^t a \geqslant \mathbb{F}_{\text{large}}$, then $2^t\zeta > \mathbb{F}_{\text{large}}$ and $\diamond(2^t\zeta) = \{\mathbb{F}_{\text{large}}, +\infty\}$ by definition. On the other hand, $\circ(2^t Z) \in \{\circ(2^t a), \circ(2^t b)\} \subseteq \{\mathbb{F}_{\text{large}}, +\infty\}$. If $2^t a < \mathbb{F}_{\text{large}}$, then $2^t a$ is a floating-point number and $2^t a \leqslant \text{pred}(\mathbb{F}_{\text{large}})$. This means that $2^t b \leqslant \mathbb{F}_{\text{large}}$ is also a floating-point number. Consequently,

$$\circ(2^t a) = 2^t a = \text{pred}(2^t b) = \text{pred}(\circ(2^t b)).$$

Finally, by monotonicity of correct rounding, $2^t a < 2^t\zeta < 2^t b$ implies

$$\circ(2^t a) \leqslant \circ(2^t\zeta) \leqslant \circ(2^t b).$$

This means that $\diamond(2^t\zeta) = \{\circ(2^t a), \circ(2^t b)\}$ and indeed $\circ(2^t Z) \in \diamond(2^t\zeta)$. The proof is thus complete. □

We turn now to the computation of a scaled $l_2$-norm. A known strategy [16] partitions the input data into three bins. The data in each bin are scaled by a common scale factor so that sums of squares of elements in each bin incur no spurious exceptions. The final result is constructed by appropriate combination of the partial sums-of-squares from the bins. We follow the same approach but with two enhancements. First, while we will explain our approach using three bins, we will point out later that our implementation keeps only two bins of data at any given time. This is important as the bins are in practice kept in the scarce SIMD vector registers. Second, our combination of the binned partial sums of squares are done in a way that guarantees a faithfully rounded scaled $l_2$-norm.

The basic idea is to divide the entire input range of $|x_j|$ into three "equal" subranges, where sums of squares of data in the middle (interior) subrange does not generate exceptions. Data in the two exterior ranges are scaled into the interior subrange. Now the specifics. Define the even integer $E$ by

$$E = \min\{e \mid 3e \geqslant e_{\max} - e_{\min} - \log_2(\varepsilon), e \text{ is even }\}.$$

From $E$, we define a scale factor $\gamma = 2^{-E}$ and use the following notations.

$$\begin{array}{rclcrcl} E_{\text{hi}} & = & e_{\max} + 1 - E, & & \beta_{\text{hi}} & = & 2^{E_{\text{hi}}}, \\ E_{\text{lo}} & = & e_{\max} + 1 - 2E, & & \beta_{\text{lo}} & = & 2^{E_{\text{lo}}}. \end{array}$$

In particular, $\gamma\beta_{\text{hi}} = \beta_{\text{lo}}$. We tabulate the specific values for binary32 and binary64 here.

| | $E$ | $E_{\text{hi}}$ | $E_{\text{lo}}$ | $\beta_{\text{hi}}\beta_{\text{lo}}$ |
|---|---|---|---|---|
| binary32 | 94 | 34 | $-60$ | $2^{-26} = \varepsilon/4$ |
| binary64 | 700 | 324 | $-376$ | $2^{-52} = 2\varepsilon$ |

Given the input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T$, the three bins are

$$\begin{array}{rcllcll} \mathcal{A} &=& \{ & \gamma x_j & | & |x_j| \geqslant \beta_{\text{hi}} & \}, \\ \mathcal{B} &=& \{ & x_j & | & \beta_{\text{lo}} \leqslant |x_j| < \beta_{\text{hi}} & \}, \\ \mathcal{C} &=& \{ & x_j/\gamma & | & |x_j| < \beta_{\text{lo}} & \}. \end{array}$$

By design $\beta_{\text{lo}} \leqslant |\widehat{x}_j| < \beta_{\text{hi}}$ for $\widehat{x}_j \in \mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$. Denote the partial, scaled, sums-of-squares as

$$\widehat{\sigma}_{\mathcal{A}} \overset{\text{def}}{=} \sum_{\widehat{x}_j \in \mathcal{A}} \widehat{x}_j^2,\ \widehat{\sigma}_{\mathcal{B}} \overset{\text{def}}{=} \sum_{\widehat{x}_j \in \mathcal{B}} \widehat{x}_j^2,\ \text{and } \widehat{\sigma}_{\mathcal{C}} \overset{\text{def}}{=} \sum_{\widehat{x}_j \in \mathcal{C}} \widehat{x}_j^2.$$

Clearly, the "bin-sums" are in the range

$$\widehat{\sigma}_{\mathcal{A}}, \widehat{\sigma}_{\mathcal{B}}, \widehat{\sigma}_{\mathcal{C}} \in \{0\} \cup [\beta_{\text{lo}}^2, n\beta_{\text{hi}}^2) \subseteq \{0\} \cup [\beta_{\text{lo}}^2, \beta_{\text{hi}}^2/\varepsilon), \tag{3.59}$$

by assuming $n \leqslant 1/\varepsilon$. A bin-sum is zero if and only if the corresponding bin is empty. Furthermore,

$$\sigma = \sum_j x_j^2 = \gamma^{-2}\,\widehat{\sigma}_{\mathcal{A}} + \widehat{\sigma}_{\mathcal{B}} + \gamma^2\,\widehat{\sigma}_{\mathcal{C}}. \tag{3.60}$$

A straightforward implementation can collect all three bins, and invoke the `SumOfSquares` function (or the parallel version) on each of the bins, followed by some appropriate combination method to arrive at the final result. Our implementation in fact only keeps and processes two bins. The observation is that $\mathcal{A}$ and $\mathcal{C}$ are never needed simultaneously. If $\mathcal{A}$ is nonempty, then Equation (3.59) and the fact that $\gamma\beta_{\text{hi}} = \beta_{\text{lo}}$ show that $\gamma^2\widehat{\sigma}_{\mathcal{C}}/(\gamma^{-2}\widehat{\sigma}_{\mathcal{A}}) \leqslant \gamma^2/\varepsilon \ll \varepsilon^2$. Neglecting $\gamma^2\widehat{\sigma}_{\mathcal{C}}$ altogether incurs a relative error (much) less than $\varepsilon^2$.

Briefly speaking, our implementation starts the binning process by keeping the interior (middle) bin and one exterior bin. When the first element belonging to $\mathcal{A}$ appears, the existing exterior bin is replaced with that element, and elements from $\mathcal{C}$ are never collected from that point onward. Denote the two actually maintained bins by $\mathcal{U}$ and $\mathcal{V}$, then

$$\sigma_2 = \gamma^k\,(\widehat{\sigma}_{\mathcal{U}} + \gamma^2\widehat{\sigma}_{\mathcal{V}}),$$

where $k = -2$ if $\mathcal{U}$ corresponds to $\mathcal{A}$, and $k = 0$ if $\mathcal{U}$ corresponds to $\mathcal{B}$. In either case, the "two-bin" sum of squares $\sigma_2$ satisfies

$$|\sigma_2 - \sigma| \leqslant \varepsilon^2\sigma,\ \text{and } \sigma_2 \leqslant (1+\varepsilon^2)\sigma. \tag{3.61}$$

To compute $\sigma_2$, the `SumOfSquares` function is applied to each of the two resulting bins, yielding two double-FP variables $\mathbf{U} = [U, u]$ and $\mathbf{V} = [V, v]$. Both $U+u$ and $V+v$ approximate their targets with high relative accuracies:

$$\frac{|(U+u) - \widehat{\sigma}_{\mathcal{U}}|}{\widehat{\sigma}_{\mathcal{U}}}, \frac{|(V+v) - \widehat{\sigma}_{\mathcal{V}}|}{\widehat{\sigma}_{\mathcal{V}}} \leqslant \Delta_{n-1}(3\varepsilon^2),$$

where $\Delta_\ell(\delta) = \ell\delta/(1-\ell\delta)$. Using $n$ instead of $n-1$ for simplicity, we have

$$\left|\gamma^k[(U+u)+\gamma^2(V+v)] - \sigma_2\right| \leqslant \Delta_n(3\varepsilon^2)\sigma_2, \tag{3.62}$$

and

$$\gamma^k[(U+u)+\gamma^2(V+v)] \leqslant (1+\Delta_n(3\varepsilon^2))\sigma_2. \tag{3.63}$$

We handle $\gamma^k[(U+u)+\gamma^2(V+v)]$ as follows. For nonzero $U$ and $V$, $\beta_{\mathrm{lo}}^2 \leqslant U+u, V+v < \beta_{\mathrm{hi}}^2/\varepsilon$ (as we assume $n \leqslant 1/\varepsilon$). If $U \geqslant \beta_{\mathrm{lo}}^2/\varepsilon^3$,

$$\gamma^2(V+v) < \gamma^2\beta_{\mathrm{hi}}^2/\varepsilon = \beta_{\mathrm{lo}}^2/\varepsilon \leqslant \varepsilon^2 U \leqslant \varepsilon^2(1+\varepsilon)(U+u).$$

Similarly, if $V \leqslant \beta_{\mathrm{lo}}^2\varepsilon^2/\gamma^2 = \beta_{\mathrm{hi}}^2\varepsilon^2$,

$$\gamma^2(V+v) \leqslant \gamma^2(1+\varepsilon)V < \beta_{\mathrm{lo}}^2\varepsilon^2(1+\varepsilon) \leqslant \varepsilon^2(1+\varepsilon)(U+u).$$

Thus if $U \geqslant \beta_{\mathrm{lo}}^2/\varepsilon^3$ or $V \leqslant \beta_{\mathrm{hi}}^2\varepsilon^2$, the error by dropping the $\mathbf{V}$ term is in the order of $\varepsilon^2$:

$$\frac{\gamma^2(V+v)}{(U+u)+\gamma^2(V+v)} \leqslant \varepsilon^2(1+\varepsilon). \tag{3.64}$$

If $U < \beta_{\mathrm{lo}}^2/\varepsilon^3$ and $V > \beta_{\mathrm{hi}}^2\varepsilon^2$, then neither $(U+u)/\gamma$ nor $\gamma(V+v)$ raises exceptions. This is because $U < \beta_{\mathrm{lo}}^2/\varepsilon^3 \implies U/\gamma < \beta_{\mathrm{hi}}\beta_{\mathrm{lo}}/\varepsilon^2 \approx 1/\varepsilon$. Similarly, $V > \beta_{\mathrm{hi}}^2\varepsilon^2 \implies \gamma V > \beta_{\mathrm{hi}}\beta_{\mathrm{lo}}\varepsilon^2 \approx \varepsilon^3$. These discussions are expressed in the function `SumOfSquaresBins`.

```
function SumOfSquaresBins($\mathbf{x}$) // general inputs
    Obtain bins $\mathcal{U}$, $\mathcal{V}$, and integer $k$ as discussed
    // $\gamma^k(\widehat{\sigma}_{\mathcal{U}} + \gamma^2\widehat{\sigma}_{\mathcal{V}})$ approximates $\sum_j x_j^2$ accurately
    // $k = -2$ if $\mathcal{U}$ is $\mathcal{A}$, $k = 0$ if $\mathcal{U}$ is $\mathcal{B}$
    // Note that $k = -2$ if and only if bin $\mathcal{A}$ is nonempty
    $[U, u] \leftarrow$ SumOfSquaresP($\mathbf{x}^{(\mathcal{U})}$);
    $[V, v] \leftarrow$ SumOfSquaresP($\mathbf{x}^{(\mathcal{V})}$);
    if $U = 0$   // $\mathcal{A}$ and $\mathcal{B}$ are both empty
        $m \leftarrow 2$, $[S, s] \leftarrow [V, v]$,
        return $m$ and $\mathbf{S} = [S, s]$.
    if $U \geqslant \beta_{\mathrm{lo}}^2/\varepsilon^3$ or $V \leqslant \beta_{\mathrm{hi}}^2\varepsilon^2$
        $m \leftarrow k$, $[S, s] \leftarrow [U, u]$
        return $m$ and $\mathbf{S} = [S, s]$
    if $|v| \leqslant \beta_{\mathrm{hi}}^2\varepsilon^2$, $v \leftarrow 0$.
    $[U, u] \leftarrow [\gamma^{-1}U, \gamma^{-1}u]$; $[V, v] \leftarrow [\gamma V, \gamma v]$; $m \leftarrow k + 1$;
    $[S, s] \leftarrow$ SumNonNeg($[U, u]$, $[V, v]$)
    return $m$ and $\mathbf{S} = [S, s]$
end SumOfSquaresBins
```

**Theorem 6.** *Let* `SumOfSquaresBins`$(\mathbf{x})$ *return* $m$ *and* $\mathbf{S} = [S, s]$, *and* $\widehat{\sigma} = \gamma^{-m}\sigma = \gamma^{-m}\sum_j x_j^2$. *If the length* $n$ *of* $\mathbf{x}$ *satisfies* $n + 3 < ((24+\varepsilon)\varepsilon)^{-1}$, *then* $\circ(\sqrt{S}) \in \diamond(\|\widehat{\mathbf{x}}\|_2)$.

*Proof.* We group the total errors incurred in computing $\sigma$ as $\gamma^m\widehat{\sigma}$ into three stages. In Stage 1, the value $\sigma$, which is exactly represented by $\gamma$ and the three bin-sums (Equation (3.60)), is approximated by $\sigma_2 = \gamma^k(\widehat{\sigma}_{\mathcal{U}} + \gamma^2\widehat{\sigma}_{\mathcal{V}})$. In Stage 2, the two bin-sums $\widehat{\sigma}_{\mathcal{U}}$ and $\widehat{\sigma}_{\mathcal{V}}$ are approximated by the double-FP $[U, u]$ and $[V, v]$. Finally, in Stage 3, $\gamma^k((U+u)+\gamma^2(V+v))$ is approximated as $\gamma^m(S+s)$ by possibly dropping $v$ or both $V$ and $v$ and the use of `SumNonNeg`.

Consider the Stage-3 error. There are three possible points of exit in the procedure `SumOfSquaresBins`. The first point of exit corresponds to a zero Stage-3 error as there is actually only one nonempty bin. The seconding point of exit corresponds to Stage-3 error bounded by $\varepsilon^2(1+\varepsilon)$ as given by Equation (3.64). If the last point of exit is taken, Stage-3 error consists of one part that is due to a single application of `SumNonNeg`, which is bounded by $3\varepsilon^2$ (Theorem 2), and one due to possibly dropping the $v$ term, which is bounded by $\varepsilon^2(1+\varepsilon)$ (Equation (3.64)). Thus we bound the error in Stage 3 conservatively by $5\varepsilon^2$:

$$\frac{\left|\gamma^m(S+s) - \gamma^k[(U+u)+\gamma^2(V+v)]\right|}{\gamma^k[(U+u)+\gamma^2(V+v)]} \leqslant 5\varepsilon^2, \tag{3.65}$$

and

$$\gamma^m(S+s) \leqslant (1+5\varepsilon^2)\,\gamma^k[(U+u)+\gamma^2(V+v)]. \tag{3.66}$$

Stage 1 and Stage 2 errors have already been discussed in Equations (3.61) through (3.63). Putting these together,

$$\begin{aligned} &|\gamma^m(S+s) - \sigma|/\sigma \\ \leqslant\; & \left|\gamma^m(S+s) - \gamma^k[(U+u)+\gamma^2(V+v)]\right|/\sigma + \\ & \left|\gamma^k[(U+u)+\gamma^2(V+v)] - \sigma_2\right|/\sigma + |\sigma_2 - \sigma|/\sigma, \\ \leqslant\; & 5\varepsilon^2(1+\Delta_n(3\varepsilon^2))(1+\varepsilon^2) + \Delta_n(3\varepsilon^2)(1+\varepsilon^2) + \varepsilon^2, \\ \leqslant\; & \Delta_n(3\varepsilon^2) + 7\varepsilon^2. \end{aligned}$$

Consequently,

$$\begin{aligned} |(S+s) - \widehat{\sigma}| \;&\leqslant\; (\Delta_n(3\varepsilon^2) + 7\varepsilon^2)\,\widehat{\sigma}, \\ &\leqslant\; (\Delta_n(3\varepsilon^2) + 9\varepsilon^2)\,\widehat{\sigma}, \\ &\leqslant\; \Delta_{n+3}(3\varepsilon^2)\widehat{\sigma}. \end{aligned} \tag{3.67}$$

From Theorems 3 and 4, $n+3 < ((24+3\varepsilon)\varepsilon)^{-1}$ implies

$$|(S+s) - \widehat{\sigma}| < \varepsilon\widehat{\sigma}/8,$$

a condition that guarantees, by Theorem 1, that $\circ(\sqrt{S}) \in \diamond(\sqrt{\widehat{\sigma}}) = \diamond(\|\widehat{\mathbf{x}}\|_2)$. □

`AccuNrm2` below is the straightforward synthesis of the previous discussions. Theorem 7 that follows is a formal statement that summarizes the technical results of this Section.

```
**function** AccuNrm2(x) // general faithful $l_2$-norm
    $(m, \mathbf{S}) \leftarrow$ SumOfSquaresBins(x)
    // $m = -2, -1, 0, 1, 2$ and $\gamma^m(S+s) \approx \sum_j x_j^2$
    $Z \leftarrow$ sqrt($S$)
    **return** $\gamma^{m/2} \otimes Z$ // $\circ(\gamma^{m/2} Z)$
**end** AccuNrm2
```

**Theorem 7.** *The function* AccuNrm2$(\mathbf{x})$ *returns a faithfully rounded* $l_2$*-norm without spurious exception as long as* $n + 3 < ((24 + 3\varepsilon)\varepsilon)^{-1}$.

*Proof.* This is a direct consequence of Theorems 5 and 6. □

### 3.6.6 Numerical Results

| | result underflows | normal result | result overflows |
|---|---|---|---|
| `NaiveNorm` | 134.8 | 46.41 | 47.14 |
| `NetlibNorm` | 470.0 | 155.9 | 161.2 |
| `MPFRNorm` | 2688 | 1084 | 1095 |
| `FaithfulNorm` | 288.2 | 34.54 | 35.19 |

Table 3.9 – Computation time per vector element in cycles

| | vectors with normal results | | vectors for which results underflow | | vectors with entries around 1.0 | | vectors with chosen "half-ulp" entries | |
|---|---|---|---|---|---|---|---|---|
| | $n = 10^3$ | $n = 10^7$ | $n = 10^3$ | $n = 10^7$ | $n = 10^3$ | $n = 10^7$ | $n = 10^3$ | $n = 10^7$ |
| `NaiveNorm` | $\infty$ | $\infty$ | $8.84 \cdot 10^{12}$ | $5.46 \cdot 10^{10}$ | 7.73 | 861 | 250 | $2.50 \cdot 10^6$ |
| `NetlibNorm` | 2.01 | 524 | 0.496 | 0.698 | 7.58 | 609 | 250 | $2.50 \cdot 10^6$ |
| `MPFRNorm` | 0.494 | 0.481 | 0.490 | 0.498 | 0.468 | 0.497 | 0.0749 | 0.484 |
| `FaithfulNorm` | 0.620 | 0.628 | 0.497 | 0.499 | 0.605 | 0.701 | 0.0749 | 0.484 |

Table 3.10 – Maximum error in ulps for random vectors of various types and lengths $n$

We implemented and tested our faithfully rounded, division-free $l_2$-norm with underflow and overflow avoidance. We used IEEE754 binary64 as working-precision and we restricted ourselves to a SIMD environment. Implementation and testing in a threaded environment is left to future work. In the tables shown, we refer to our implementation by `FaithfulNorm`.

For our algorithm, we were able to implement it without any branches in the main accumulation loop by the use of instructions predicated with appropriate masks. The resulting program is branch-free in its inner loop and therefore well suited for pipelined processors.

We compared the implementation of our faithfully rounded $l_2$-norm with implementations for other approaches with respect to both accuracy and performance. To do so, we implemented a naive $l_2$-norm, called `NaiveNorm`, that plainly uses working precision for squaring the $x_i$ and accumulating these squares, without any underflow and overflow avoidance. We further implemented the algorithm found in `netlib` [8]; we call this implementation `NetlibNorm`. Finally, we implemented another faithfully rounded $l_2$-norm using the arbitrary precision library MPFR [86]. This implementation is simply based on an exact accumulation of the squares $x_i^2$ in an accumulator that provides enough precision. We refer to it under the name `MPFRNorm`.

We performed testing on a 4-core Intel Core i7 at 2.67 GHz with 4Gb of RAM. All implementations are in C, using builtins for access to SIMD instructions. They were compiled using gcc version 4.8.1 and options `-std=c99 -O3 -msse3`. Timings are given in cycles per element.

We did accuracy testing for random vectors of various lengths $n$ and input types. The testing results are resumed in Table 3.10. For each length $n$ indicated in the Table 3.10, $500$ test runs were performed.

First, we considered random vectors chosen such that the final $l_2$-norm result is a normal floating-point number. Second, we tested the algorithms on random vectors for which the final result gradually underflows. Third, we performed testing on vectors with inputs around $1.0$, i.e. where underflow or overflow avoidance is not necessary. Finally, we constructed input vectors with "half-ulp" entries $x_j$ for which the $x_j^2$ are near the midpoint of two consecutive working-precision floating-point numbers: for such inputs any algorithm using no more than working precision for the squarings $x_j^2$ will systematically suffer a $0.5\,\text{ulp}$ error per element.

Testing shows that both the `NaiveNorm` and `NetlibNorm` implementations fail to provide faithfully rounded results. It establishes also that both our `FaithfulNorm` as well as the `MPFRNorm` algorithm do yield faithfully rounded results.

In cases when the final $l_2$-norm does not overflow, our algorithm `FaithfulNorm` returns a result with an error well below $1\text{ulp}$, typically $0.7\text{ulp}$. As expected, the maximum error does not vary with vector length, whereas it does for the `netlib` $l_2$-norm.

Accuracy testing also shows that although the `netlib` $l_2$-norm gives acceptable, while not faithfully rounded, results in general, it may, for particular inputs, suffer a tremendously high error, as large as about $0.25\,n\,\text{ulp}$ for a vector of length $n$.

With respect to performance testing, Table 3.9 reports the performance numbers we observed. We used vectors of floating-point numbers picked at random in various domains of interest. We tested for vectors for which the final $l_2$-norm result gradually underflows, overflows or stays in the range of normal floating-point numbers. Measurements were done for varying vector lengths. Given some minimal vector length (a couple of dozen elements), vector length had no influence on computation time per element.

These performance results clearly speak in favor of our `FaithfulNorm` implementation. For cases when the final result is a normal floating-point number, our implementation is about $4.5$ times faster than the `netlib` implementation. As already explained, this is due to several factors: avoidance of spurious underflow, no use of expensive divisions and an algorithm that is branch-free in the inner loop.

We shall however mention that our algorithm does have lower performance than the `netlib` for cases when the vector length (very) is small – typically less than a dozen elements. In this case our algorithm has a much higher static overhead due to the pretty complicated final reduction of the bins and square root computation. In future work, we shall address this problem with a call-out to a specialized $l_2$-norm for very small vectors.

### 3.6.7 Conclusion

In this Section, we presented an efficient algorithm to compute the faithful rounding of the $l_2$-norm of a floating-point vector. While our algorithm is very accurate, it is also faster than previous algorithms like the one of `netlib` that gives no information about the accuracy of the result. Moreover, our algorithm avoids spurious overflow and underflow. It is also suitable for parallel implementations.

## 3.7 With and beyond IEEE754: Things Done and Things To Do

In this Chapter, we have analyzed what is needed on the software side to support IEEE754-2008 and what extensions could be envisioned for future extensions of the Standard. We have seen that the implementation of the so-called heterogeneous floating-point operations is possible with a sequence of homogeneous operations, that have already been supported by the 1985 version of the IEEE754 Standard. We have seen how this support can be integrated into a support library, such as our `libieee754`, which is, however, still ongoing work.

We came up with an algorithm that is able to convert arbitrarily long decimal character sequences to binary floating-point numbers with correct rounding, not requiring dynamic memory allocation anymore, as did existing approaches. We built upon this technique to propose and implement a mixed-radix fused-multiply-add operation, taking arguments in a mix of IEEE754-2008 binary64 and decimal64 formats and returning a correctly rounded result also in one of these two formats. This work on mixed-radix arithmetic, the inclusion of which would make the Standard complete in terms of combination of formats in operations, was also based on preliminary work on mixed-radix comparisons, which we also presented in detail. We then went on and proposed other extensions of the Standard's floating-point arithmetic, in two directions: proposing other, more precise formats for floating-point arithmetic while still maintaining statically known memory consumption –which arbitrary precision libraries such as MPFR can clearly not guarantee– and by discussing high level operations on vectors of floating-point numbers, for instance a faithfully rounded vector 2-norm algorithm.

While these lines are written, the IEEE754-2008 is undergoing revision once again. The new version of the Standard is expected to be published in 2019. No major changes are allowed to happen in this new version: the revision committee is bound to keeping to the same wording, adding no new "shall" statements in this revision cycle[35]. It is in the next revision, scheduled for about 2028, that major changes can be considered. Whatever these changes to IEEE754 might bring, one point continues to grow more and more important: the IEEE754 Standard has grown from a 20 pages document published in 1985 [116] to a 70 pages document in 2008 [117] that requires support for 354 operations for just the two most common ones of its formats, binary32 and binary64. The Standard is however specified with English only, which has made some discussions for the 2019 revision extremely tedious[36]. In our opinion, we must strive at formalizing the specifications of IEEE754 in a way that allows for

— automatic generation of a reference implementation out of the formalization, even if this implementation presented issues with respect to compute performance,

— automatic generation of test vectors for implementations in support libraries such as our `libieee754`,

— automatic checking for inconsistencies of the Standard's specifications, and

— a sustainable way to extend the Standard, putting the burden of the tedious details on automatic checkers.

As a matter of course, various formalizations of the IEEE754 Standard exist, in particular the well known Flocq formalization [22]. To our knowledge, these formalizations cannot be

35. see `https://development.standards.ieee.org/get-file/P754.pdf?t=86948000003`

36. see e.g. the email thread started 2018-12-23 about adding a comma, archived at `https://listserv.ieee.org/cgi-bin/wa?A0=STDS-754`

translated, in an automatic manner, into a reference implementation and, most importantly, they do not allow for easy changing of certain of the Standard's characteristics, in an attempt to "play" with various competing future extensions. We shall work towards that goal with our future research, too.

# Chapter 4

# Floating-Point Code Generation

*Einer staunte darüber, wie leicht er den Weg der Ewigkeit ging; er raste ihn nämlich abwärts.*

Franz Kafka

## 4.1 Introduction

In the last Chapter 3, we have tried to support and in particular to extend the floating-point arithmetic environment, as it is defined e.g. by the IEEE754-2008 Standard. We have proposed several algorithms, e.g. for mixed-radix comparisons and even mixed-radix arithmetical operations. These extensions, which we consider sensible, might one day be considered for inclusion into the IEEE754 Standard. Some of these extensions would add quite a number of different operations to the Standard, even just one mixed-radix FMA for one binary and one decimal formats adds $2^4 - 2 = 14$ operations. However, all the extensions add nothing but a finite, pretty small number of operations to the environment.

In contrast, the number of arithmetical operations that can be defined for mathematical functions $f : \mathbb{R} \to \mathbb{R}$ is naturally infinite, as infinitely many such functions exist. Even for a finite set of common functions, as they are covered by classical mathematical libraries (`libm`s), a huge number of implementations exists: these implementation may differ in accuracy, input and output formats, as well as coverage of the definition domain of the implemented function. Implementing all these variants, which we shall call flavors, is extremely tedious; implementing an infinite, open-ended set of mathematical functions is, of course, impossible.

If we want to include mathematical functions in an open-ended manner in a floating-point compute environment, we must hence pass onto another level, stop thinking about algorithms that perform arithmetic and continue with algorithms that produce, i.e. compute, the implementations of the various functions we may consider. In this Chapter, we shall do so, studying the extension of the floating-point environment with *code generation*.

As we shall see at the end of this Chapter 4, in Section 4.5, our work on code generation for mathematical function seamlessly translates to code generation for digital signal processing filters, to which we dedicate the next Chapter 5.

This Chapter is based on the following articles we published: [32, 146–149, 157, 163]. For the chapter in this work, we extended this material quite extensively with some currently unpublished work. Technologically, all this work heavily relies on the Sollya tool [41], which we continue to develop but chose not to describe in detail in this work. Financewise, an important part of the work presented here was financed by the ANR project Metalibm.

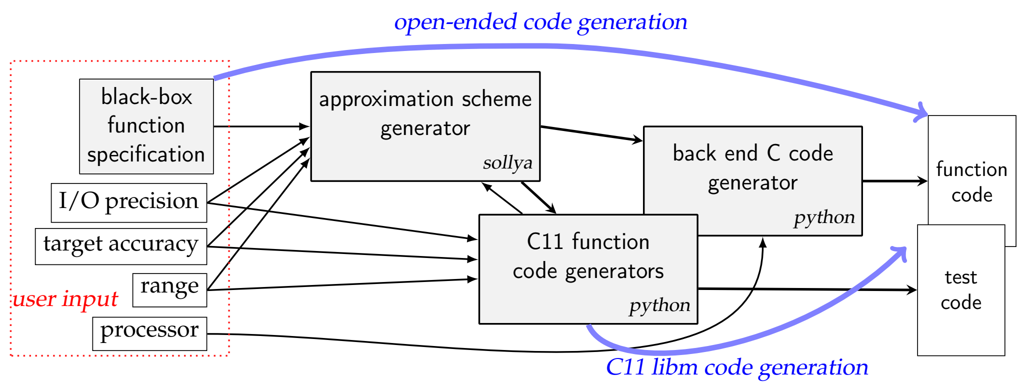
open-ended code generation
black-box function specification
I/O precision
target accuracy
range
processor
user input
approximation scheme generator
sollya
back end C code generator
python
C11 function code generators
python
function code
test code
C11 libm code generation

Figure 4.1 – The proposed Metalibm tool suite

## 4.2 Implementation of Mathematical Functions

In this Section 4.2, we give an overview on techniques in the development of which we have participated. In the subsequent Sections 4.3 and 4.4, we shall detail some aspects of what we understand by open-ended code generation and of use-cases for code generation for mathematical functions. In this Section, we both present the piece of software we developed for open-ended code generation, called metalibm-lutetia [1] and a "competing" software approach, metalibm-lugdunum, developed by other scientists with a more specific application scope. Section 4.4.2 however will make it clear that both metalibm-lutetia and metalibm-lugdunum are part of a single scientific effort and that the initial distinction is nothing but smoke and mirrors: we shall show how we can leverage metalibm-lutetia to provide input for metalibm-lugdunum, while also taking profit from metalibm-lugdnunum's performance optimization capabilities for metalibm-lutetia. Both pieces of software have become one single code generator.

### 4.2.1 Introduction & motivation

#### Standard mathematical libraries

The standard mathematical library (`libm`) provides, in a small set of precisions (single and double precision), a small set of mathematical functions:

— *elementary* functions [196] such as exponential and logarithm, sine, cosine, tangent and their inverses, hyperbolic functions and their inverses;

— and other "special" functions useful in various computing contexts, such as the power function $x^y$, erf, $\Gamma$ or the Airy function [4].

Strictly speaking, the `libm` is specified in the C language standard (currently ISO/IEC 9899:2011). Among the other languages, some (C++, Fortran, Python) use the same library, and some redefine it in a more or less compatible way. The 2008 revision of the IEEE 754 floating-point (FP) standard [117] has attempted a common standardization. In addition, language standards also specify elementary functions on complex numbers, but for historical

1. or simply metalibm if the distinction is not needed as the context is clear

reasons not in the "mathematical library" section. Also, many operating systems or libraries offer, for convenience, more functions than strictly required by the `libm`. In this work we understand the term "`libm`" in its widest sense and address all these functions, and more.

**A matter of performance**

Performance of `libm` functions is of critical importance, in particular in scientific and financial computing. For example, profiling the SPICE electronic simulator shows that it spends most of its time in the evaluation of elementary functions [134]. The same holds for large-scale simulation and analysis code run at CERN [9, 119, 209].

The problem is that the optimal implementation of each function is dependent on the technology. Mathematical support is provided by a combination of hardware and software, and the optimal repartition between hardware and software has evolved with time [206]. For instance, the first 80287 mathematical coprocessor, in 1985, included support for a range of elementary functions (albeit in microcode) in addition to the basic operations. It was later found that software could outperform such microcode [65]. For instance, as memory got cheaper, large tables of precomputed values [90, 239] could be used to speed up the computation of a function. Furthermore, progress in compiler technology allowed for a deeper integration of elementary functions software in the overall compilation process [52, 185, 186], another case for software-based functions. Today, even the relevance of hardware division and square root is disputed. At the same time, the table-based algorithms of the 90s are being replaced with high-degree polynomials [52] that behave better in the context of current highly parallel, memory-constrained multicores.

**Limits of the library approach**

**Productivity of `libm` development** Writing a `libm` requires fine-tuned instruction selection and scheduling for performance, and sophisticated FP techniques for accuracy. Re-optimization of these mutually dependent goals for each new processor is a time-consuming and error-prone task. Besides, it is desirable that each function comes in several variants corresponding to a range of constraints on performance (e.g. optimized for throughput or optimized for latency) or accuracy. Some processor and system manufacturers (Intel, AMD, ARM, NVIDIA, HP, Apple) therefore employ teams of engineers dedicated to `libm` maintenance.

With more limited manpower, the open-source mathematical libraries (most notably in the GNU `glibc` [179] and in Newlib [2]) lag behind in performance, in particular as they have to support a much wider range of processors. However, the most accurate implementations are found in open-source efforts, with several correctly rounded functions provided by libraries such as IBM LibUltim [264] (now in the `glibc`) and CRLibm [64].

**Versatility of `libm` use** Another problem is that the `libm` functions do not meet all the needs of users. First, the choice is limited. Only a fraction of the functions used by CERN simulations [3] is offered by the standard `libm`. The rest is programmed when needed, sometimes without the expertise of `libm` developers. This leads to duplicated efforts and, often, poorer quality (performance- or accuracy-wise) than `libm` function.

2. `https://sourceware.org/newlib/`
3. `http://project-mathlibs.web.cern.ch/project-mathlibs/mathTable.html`

Second, in the specific context of an application, the functions in the `libm` are often overkill. A generic library must be specified with the widest and best possible parameters in mind: argument range is specified “as far as the input and output formats allow” and accuracy “as accurate as the output format allows”. However, if a programmer knows for instance that the input to a cosine will remain in a period, and that about three decimal digits of accuracy will be enough considering the accuracy of the inputs, the `libm` `cosf` function is too accurate (7 decimal digits) on too large a range.

The present Section claims that the solution to both previous issues (productivity, and versatility) is to automate `libm` development so that functions can be generated on-demand for a wider set of contexts. This solution has already proven effective for other classes of mathematical software, most notably the ATLAS [257] project for linear algebra, and the FFTW [89] and SPIRAL [212] projects for fast Fourier transforms.

**Use cases for generators of mathematical functions**

We are currently focusing on two main use cases where we need a `libm` generator or *metalibm*, both illustrated by Figure 4.1.

The first one targets the widest audience of programmers. It is a push-button approach that will try to generate code on a given domain and for a given precision for an arbitrary univariate function with continuous derivatives up to some order. The function may be provided as a mathematical expression, or even as an external library that is used as a black box. We call this approach the open-ended approach, in the sense that the function that can be input to the generator is arbitrary – which does not mean that the generator will always succeed in handling it. Section 4.2.3 will describe how this generator has evolved from simple polynomial approximations to the generation of more sophisticated evaluations schemes, including attempts to range reduction. Here, the criterion of success is that the generated code is better than whatever other approach the programmer would have to use (composition of `libm` function, numerical integration if the function is defined by an integral, etc). “Better” may mean faster, or more accurate, or better behaved in corner cases, etc.

Still, the `libm` is here to stay, and we also want to address the needs of `libm` developments. Although the techniques presented in Section 4.2.3 can eventually be extended to capture all the functions of C11, it is currently not the case. There is a lot of human expertise that we are not yet able to automate. In particular, bivariate functions as `atan2` or `pow`, and some special functions, are currently out of reach. The second use case focuses on assisting people who have this expertise, not yet on replacing them. It targets a much narrower audience of programmers, those in charge of providing the actual `libm` functionality to an operating system or compiler. Here the criterion of success is that the generated code is of comparable quality to hand-written code, but obtained much faster.

The chosen approach, reviewed in Section 4.4.2, consists in giving to the `libm` developer the keys to the back end. We thus aim at offering a development framework in which the relevant evaluation schemes may be described. The challenge here is to find a proper balance between two conflicting goals: 1/ This framework should be able to capture every trick that `libm` developers use, otherwise they will not adopt it. 2/ It should nevertheless raise the level of abstraction, so that a single description generates code for the variety of code flavors and targets we want to address. And it should be fully scriptable to enable design-space exploration, but always under full control of the `libm` developer. Section 4.4.2 reviews a technical solution that matches these goals.

This second use case can be viewed as a pragmatic, bottom-up approach, where we embed existing hand-crafted code in a framework to make it more generic. The first, open-ended use case is more ambitious, more high-level, and top-down from the most abstract mathematical description of a function. These two approaches do meet as illustrated on Figure 4.1, and we do not claim that there is a clear border between them. For instance, the `libm` developer working directly with the back end will nevertheless invoke the evaluation scheme generator, to solve sub-problems, typically after range reduction.

### 4.2.2 Background on elementary function implementation

In order to understand the challenge of writing a code generator for mathematical functions, it is useful to know how they are implemented manually. The hardware provides FP additions, multiplications, bit-level manipulations, comparisons, and memory for precomputed tables. A generic technique exploiting all this is to approximate a function by tables and (possibly piecewise) polynomials.

#### Polynomial approximation

There are many ways [196] to compute an approximation polynomial $p$ for a given function $f$. For our purpose, the best choice is in general a variant [26] of Remez algorithm that computes the minimax polynomial on a given interval $I$, *i.e.* the polynomial minimizing the approximation error

$$\varepsilon_{\text{appr}} = \|f - p\|_{\infty}^{I} = \max_{x \in I} |f - p|$$

among all the polynomials of a given degree $d$ with FP coefficients of a given format.

For a given $f$ and $I$, the larger the degree $d$, the better the approximation (the smaller $\varepsilon_{\text{appr}}$). When implementing a `libm`-like mathematical function, we have an upper bound constraint on $\varepsilon_{\text{appr}}$, and we look for the smallest $d$ that satisfies this bound.

#### Argument reduction

When it is not possible to compute a polynomial of a small enough degree, the implementation domain $I$ may be reduced. There are two basic techniques for this: split the domain into smaller subdomains, or use specific properties of the function. In both cases the approximation problem is reduced to approximating a (possibly different) function $f_r$ on a smaller domain.

**Domain splitting** The main idea here is to partition $I$ into subdomains, so that on each of them we can find a minimax polynomial approximation of small degree. The splitting can be uniform, logarithmic [168], or arbitrary [147]. In the two first cases, the polynomial is selected based on a few bits from the input $x$. In the last case, several `if-else` statements are used in general to select the right interval index; some techniques to overcome this vectorization bottleneck through an additional helper-polynomial have been proposed, for instance, with our approach published in [149]. We shall detail both the splitting process as well as the index-selection technique below.

**Function-specific argument reduction** For some elementary functions, specific algorithms of argument reduction may be applied [196, 239, 240]. They are based on mathematical properties such as $b^x \cdot b^y = b^{x+y}$. For example, the exponential function $e^x$ can be computed as follows [239]. First, $e^x$ is rewritten as

$$e^x = 2^E \cdot 2^{k-E} \cdot 2^{x \log_2 e - k} \quad .$$

Given a parameter $w \in \mathbb{N}$, we may compute $k = \lfloor 2^w x \log_2 e \rceil 2^{-w}$ and $E = \lfloor k \rfloor$. Thus, $E \in \mathbb{Z}$ (it will provide the exponent of the result) and $k \in 2^{-w}\mathbb{Z}$. We may then compute $r = x \log_2 e - k$, a small value. Finally the exponential is rewritten as $e^x = 2^E 2^{k-E} e^r$, where the values of $2^{k-E}$ may be read in a precomputed table of size $2^w$ (indexed by $k$), while $e^r$ may be well approximated by a polynomial, possibly with additional splitting.

### Existing `libm` development tools

A mathematical function generator must build code with guaranteed accuracy. For this, it can rely on several tools that have already been used in manual `libm` development.

Sollya [41] is a numerical toolbox for `libm` developers with a focus on safe FP computations. In particular, it provides state-of-the-art polynomial approximation [26], safe algorithms to compute $\varepsilon_{\text{approx}} = \|f - p\|_\infty^I$ [43] as well as a scripting language.

Gappa [64] is a formal proof assistant that is able to manage the accumulation of rounding errors in most of `libm` codes. Compared to [64], in the present work the Gappa proof scripts are not written by hand, but generated along with the C code. Interestingly, Gappa is itself a code generator (it generates Coq or HOL formal proofs).

## 4.2.3 Approximation of black-box functions

### Overview

This Section describes an open-source code generator written in Sollya. It inputs a parametrization file and produces corresponding C code. The parametrization file includes the function $f$, its implementation domain $I$, the desired accuracy $\bar{\varepsilon}$, maximum degree $d_{max}$ for approximation polynomials, etc.

The function may be specified as a mathematical expression, or an external library. However, an important feature of the approach is that it doesn't require an explicit formula for the function. The proposed tool only accesses the function as a numerical black box. All the tool requires from this black box is to be able to return (in acceptable time) an arbitrarily accurate numerical evaluation of the function and its first few derivatives. This way, the approach works for functions described not only as closed-form expressions, but also as integrals, inverse functions, etc.

This black-box encapsulation of the function will not prevent the tool to exploit the function-specific range reductions presented in Section 4.2.2. What follows below shows that the required mathematical properties can be detected and exploited from a black-box function.

The tool also supports higher-than-double accuracy, using double-double and triple-double arithmetic.

**Different levels of approximation schemes**

We can structure function generation process in open-ended generator in three levels.

**First level (polynomial approximation)** This level approximates a function $f$ by a polynomial $p$ with an error bounded by $\bar{\varepsilon}$, and generates C code and Gappa proof.

**Second level (piece-wise polynomial approximation)** This level uses domain splitting and piecewise polynomial approximation. For the most efficient way of performing this step, we have developed an algorithmic technique. We presented a code generation-time algorithm [147] that allows the initial domain to be split in such a way that the number of resulting sub-intervals is minimized, typically by maximizing the width of the sub-intervals while keeping the polynomials' degrees bounded by some *a priori* value.

This work on optimized domain splitting [147] can be summed up as follows:

— **Equal-Width-Splits:** The simplest way to perform the domain splitting is to take some large $k$ and split the domain $I$ into $k$ equal parts. For the mentioned example we may take $k = 50$; the diagram of the corresponding degrees may be found on Figure 4.2. This approach works but produces too many subdomains and, as there is head-room between $d_{max}$ and the real polynomial degrees, the splitting can be improved.

  Instead of a uniform splitting one can use a splitting algorithm that takes into account the function behavior, for example bisection (Figure 4.3) that gives 23 subdomains. Even if we reduce the number of subdomains $k$ in the uniform splitting, the least number in the uniform splitting is 45, so bisection procedure saves memory.

  However, a naive bisection procedure may be improved: on some of the subdomains there is still head-room between actual polynomial degree and $d_{max}$ and, in some cases, the actual degrees on the adjacent subdomains differ too much, so the diagram of the degrees is not regular (see Figure 4.4).

  We need some theoretical background to perform an optimized splitting that regularizes the diagram of the polynomial degrees on each of the subdomains as much as it can.

— **Minimax polynomials:** It was mentioned that the function implementations use approximations and our code generator computes polynomial approximations. In this

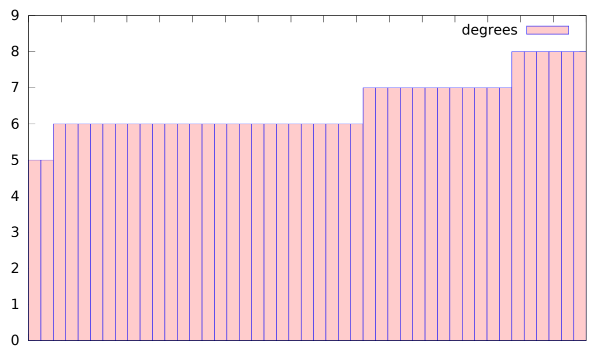

Figure 4.2 – Naive $k$-equal split for asin function. Domains and degrees.

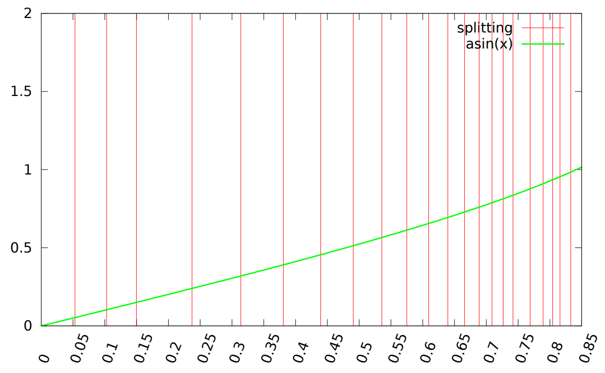

Figure 4.3 – Bisection splitting for asin function.

Section we consider the theoretical base to find or to estimate the best polynomial approximation.
There is a lot of methods to compute a polynomial approximation, but the most accurate result is obtained by computing a minimax polynomial. The minimax polynomial $p$ for a function $f$ on a given interval $I$ minimizes the approximation error

$$\widetilde{\varepsilon} = \|f - p\|_{\infty}^{I} = \max_{x \in I} |f - p|$$

among all the polynomials of a given degree $d$. The same is applicable for relative error. Remez algorithm [214] with a small modification is used to find a minimax polynomial [26]. The classical algorithm produces real coefficients, and rounding them to floating-point numbers yields to loss of accuracy. The algorithm proposed in [26] and implemented in Sollya [41] finds a minimax polynomial among all the polynomials with floating-point coefficients.

— **Remez algorithm:** Remez algorithm has quadratic convergence to a minimax polynomial when the function $f$ is twice differentiable and with additional conditions for approximation points $x_i$. We do not explain here the whole algorithm, we just give an idea. It is an iterative algorithm and first $n + 2$ points $x_0, \ldots, x_{n+1}$ from $[a, b]$ has to be chosen. Then in a loop there are four actions repeated until the needed approx-

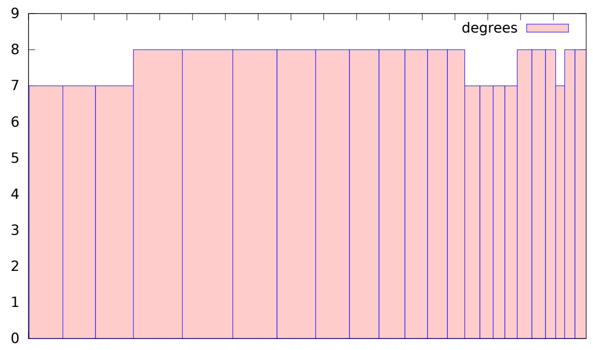

Figure 4.4 – Polynomial degrees for bisection splitting of function asin on domain $[0; 0.75]$

imation accuracy is reached. First, an approximation $p$ of $f$ has to be built on the chosen $n+2$ points. Then, to compute the current accuracy, we have to compute $\varepsilon = \max_{i=0,\dots,n+1} |p(x_i) - f(x_i)|$, we compute or estimate the value of $\|p - f\|_\infty$, take another set of $n+2$ points and repeat the loop. On the first step, Chebyshev nodes are often chosen as the set of $n+2$ points:

$$x_i = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{(n+1-i)\pi}{n+1}\right),\ i = 0,\dots,n+1.$$

It is an expensive algorithm, we have to perform a lot of function evaluations, compute infinite norms and make comparisons. The infinite norm is computed with the algorithm from [40]. For the example of asin Remez approximation procedure makes about 9000 function evaluations for each of the subdomains.

— **Theorem of de la Vallée-Poussin** However, to know error bounds for the best polynomial approximation, it is not always mandatory to compute this approximation itself. We may skip several computation steps estimating the bounds for the approximation error as it will be shown later.

**Theorem 8** (of de la Vallée-Poussin)**.** *Let be $f$ a continuous function $f \in C_{[a,b]}$, $p$ its approximation polynomial on $n+2$ points $x_0 < x_1 < \dots < x_{n+1}$ from $[a,b]$ such that the error $f - p$ has a local extremum and its sign alternates between two successive points $x_i$, then the optimal error $\mu$ verifies*

$$\min_{i=0,1\dots,n+1} |f(x_i) - p(x_i)| \leqslant \mu \leqslant \max_{i=0,1\dots,n+1} |f(x_i) - p(x_i)|.$$

The mentioned approximation points $x_i$ for polynomial $p$ may be chosen as Chebyshev's nodes [37]: from the alternation Chebyshev theorem, in this case the approximation error oscillates perfectly between its extrema at least $n+2$ times. Theorem of de la Vallée-Poussin takes a polynomial $p$ with an error oscillating $n+2$ times and claims that the quality of the approximation $p$ is related to the quality of the oscillations [38].

This theorem allows us to check the quality of the approximation: polynomial $p$ is considered as the best approximation if $\varepsilon$ and $\|f - p\|_\infty$ are sufficiently close.

```
1 Procedure checkIfSufficientDegree($f, I, n, \bar\varepsilon$):
  Input  : function $f$, domain $I = [a; b]$, degree $n$, target accuracy $\bar\varepsilon$
  Output : true in the case of success, false in the case of fail
  $x_i \leftarrow \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{(n+1-i)\pi}{n+1}\right),\ i = 0,\dots,n+1$;
  $p \leftarrow$ interpolate($f, x_i$);
  $m \leftarrow \min_{x_i} |f(x_i) - p(x_i)|$ ;
  $M \leftarrow \max_{x_i} |f(x_i) - p(x_i)|$ ;
  if $\bar\varepsilon \geqslant M$ then $r \leftarrow$ true;
  if $\bar\varepsilon \leqslant m$ then $r \leftarrow$ false;
  if $\bar\varepsilon > m \wedge \bar\varepsilon < M$ then
      $p \leftarrow$ remez($f, I, n, \bar\varepsilon$);
      $\delta \leftarrow$ supnorm($f - p, I$);
      $r \leftarrow \delta \leqslant \bar\varepsilon$ ;
  end
  return $r$;
```

As Remez is an iterative algorithm, on each step we may check where the current approximation error is situated relatively to the optimal error. This theorem allows to write a procedure `checkIfSufficientDegree` that checks if it is possible to compute a polynomial of degree $d$ that approximates a function $f$ on an interval $I$ with error $\varepsilon$. You may find a pseudo-code for the mentioned procedure below. It starts with computation of Chebyshev approximation polynomial of degree $d_{max}$ and then it obtains the bounds for optimal approximation (from de la Vallée-Poussin theorem). When the target accuracy is larger than the upper bound for the approximation error, the method returns true, when the target accuracy is lower than the lower bound, it returns false. In the case when the target accuracy is between the bounds, it is not clear and a Remez iteration is needed.

The procedure `checkIfSufficientDegree` is useful to compute domain splitting based on bisection procedure.

— **Towards a new splitting algorithm:** The upper-mentioned theory gives a base for a domain splitting algorithm. We know how to solve a non-standard approximation problem: for a given function $F$ on an interval $I$ compute a polynomial of an unknown degree $d$ that approximates $f$ on $I$ with an error $\varepsilon, \varepsilon \leqslant \widetilde{\varepsilon}$. We may use la Vallée-Poussin theorem to compute the splitting. We will need to compute a polynomial approximation $p$ in Chebyshev nodes and check the infinite norm of $p$ against target error. This can be done with `checkIfSufficientDegree` procedure.

The first improvement of the linear split on $k$ equal subdomains is bisection. There is a set of parameters for the algorithm: a function $f$, needed approximation error $\widetilde{\varepsilon}$, minimal width of the subdomain $w_{min}$, maximum bound for the polynomial degree $d_{max}$ and the domain $I$. The algorithm returns a list of split points.

So, we start to check if it is possible to approximate the function $f$ on the whole domain by a minimax polynomial of degree $d_{max}$ with the error bounded by $\widetilde{\varepsilon}$. If `checkIfSufficientDegree` returns true, then the splitting is computed and the empty list has to be returned. If the checking procedure returned false, we have to split the interval $I$ into two equal non-overlapping parts $I_{left}$ and $I_{right}$ and to repeat it recursively for the left part. We continue to bisect the current interval until `checkIfSufficientDegree` returns true for all the parameters and current interval $I_{left}$. In this case we append $m = \sup(I_{left})$ to the list of split points and repeat the procedure for the rest of the initial interval, i.e. $[m, b]$. The algorithm returns error, if the size of the currently checked interval is less than $w_{min}$.

Here is the pseudo-code for the bisection splitting. It uses previously explained procedure `checkIfSufficientDegree` for an interval, if it returns false, it splits interval

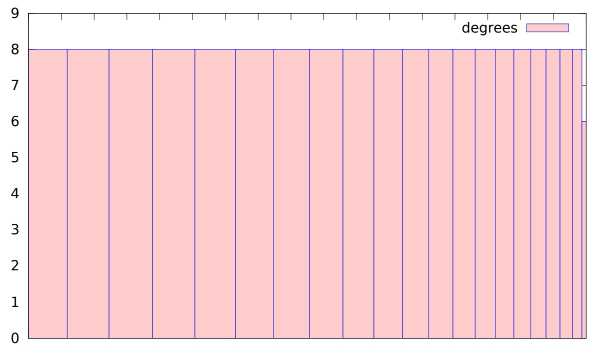

Figure 4.5 – Polynomial degrees for improved bisection splitting for asin example.

into two equal parts. The procedure is repeated recursively.

```
1 Procedure computeOptimizedSplitting($f, I, d, \overline{\varepsilon}$):
  Input   :function $f$, domain $I = [a; b]$, max. degree $d$, target accuracy $\overline{\varepsilon}$
  Output:list $\ell$ of points in $I$ where domain needs to be split
  if checkIfSufficientDegree($f, I, d, \overline{\varepsilon}$) then return $\ell = []$;
  $m \leftarrow b$;
  while $\neg$ checkIfSufficientDegree($f, [a; m], d, \overline{\varepsilon}$) do $m \leftarrow (a + m)/2$ ;
  $J \leftarrow [m; b]$;
  $\ell \leftarrow$prepend($m$, computeOptimizedSplitting($f, J, d, \overline{\varepsilon}$));
  return $\ell$;
```

Thus, the algorithm returns only splitting points inside the initial interval $I$, the resulting list does not contain its borders $a, b$.

For the asin example bisection splits the domain into 23 subdomains and the degrees diagram is on the Figure 4.4.

— **Improved bisection:** The bisection produces less intervals than the naive linear approach, but it is still not optimal: some intervals may be merged together to reduce the headroom between $d_{max}$ and actual polynomial degree. The improved version of splitting is based on the bisection, but then, as soon as we find a suitable interval on the left, we try to move its right border on some $\delta$. And then we repeat for the rest of the initial interval.

This algorithm contains two procedures: bisection and enlarging. While the algorithm `computeOptimizedSplitting` simply splits the interval in its middle through the assignment $J \leftarrow [m; b]$;, the improved bisection algorithm uses a subroutine call at this point to find the locally widest interval.

For the asin example, this improved bisection method produces 21 subdomains; Figure 4.5 shows the corresponding polynomial degrees diagram. The degrees on 20 of the intervals are equal to 8, and only on the last small interval the obtained degree is 6.

— **Left-to-right and right-to-left directions:** As on each step of the algorithm we try to enlarge the leftmost suitable interval, we may have a situation when the degrees on the first intervals are close to $d_{max}$, but on the last one (or even several last intervals) corresponding polynomial degree is small. A similar algorithm may be obtained, when instead of the leftmost suitable intervals we take the rightmost suitable intervals. In the first case we compute the split points from left to right, in the second case from right to left. For right-to-left direction we enlarge the intervals from bisection procedure by moving their left borders down. In this case we may have low degrees on several first intervals, while, on the other intervals, the degree is close to $d_{max}$.

This approach of computing optimized splittings into sub-domains is implemented inside the open-ended function code generator metalibm-lutetia. More detail can be found in [149].

**Third level (exploiting algebraic properties)** This level attempts to detect mathematical properties, and uses them to generate argument reduction code. Properties that are currently detected by the tool include

$$\begin{gathered} f(x+y) = f(x)f(y) \\ f(x+C) = f(x) \\ f(x)+f(y) = f(xy) \\ f(x) = f(-x);\ f(x) = -f(x). \end{gathered}$$

They respectively correspond to exponential functions $b^{x+y} = b^x b^y$, periodic functions, e.g. $\sin(x + 2\pi k) = \sin(x)$, logarithms $\log_b(x) + \log_b(y) = \log_b(xy)$ and symmetrical functions (both even and odd). When such a property is detected, the tool uses the corresponding argument reduction. Then it needs an approximation scheme in the reduced domain. For this, it performs a recursive call to the first or second level.

As the tool only accesses the function as a black box, it is unable to prove that the property is actually true. However, all it needs is to ensure that it is true up to some accuracy (the accuracy needed for the range reduction to work). This can be done purely numerically.

As an example, let us show how to detect the property $f(x + y) = f(x)f(y)$, which corresponds to a family of exponential functions $b^{x+y} = b^x b^y$ for some unknown base $b \in \mathbb{R}$.

First, two different points $x$ and $y$ are chosen in $I$, and the tool checks if there exists $|\varepsilon| < \bar{\varepsilon}$ such that $f(x + y) = f(x)f(y)(1 + \varepsilon)$. If not, the property is not true. If yes, the tool deduces a candidate $b = \exp\left(\frac{\ln(f(x))}{x}\right)$ for some random $x \in I$. Only in this case it checks that the property is true up to the required accuracy, by computing

$$\widetilde{\varepsilon} = \left\| \frac{b^x}{f(x)} - 1 \right\|_{\infty}^{I}$$

and checking if $\widetilde{\varepsilon} \leqslant \bar{\varepsilon}$.

Other properties can be detected in the same way. For instance, periodic functions may be detected with a numerical zero search for $C$ such that $f(x_0 + C) - f(x_0) = 0$, followed by a computation of $\|f(x + C) - f(x)\|_{\infty}^{I}$.

However, handling the error bound $\widetilde{\varepsilon}$ requires some analysis for certain function properties.

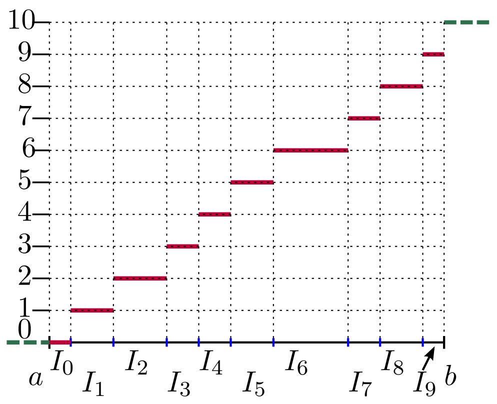

Figure 4.6 – Piecewise-constant mapping function $M(x)$

## Reconstruction

The final step in mathematical functions implementation is reconstruction, i.e. the sequence of operators (or just a formula) needed to be executed to evaluate the function at some point $x$ from the initial domain $I$. The reconstruction may be tricky if the generation was done on the second level, and the produced code has to be vectorizable. To make the reconstruction vectorizable, some mapping function that returns an index of the corresponding interval has to be exhibited [149].

The approach we proposed to replace branches by polynomial evaluation when determining the index of a function's input value $x$ in the sub-domains [149] globally follows the following ideas:

— **Classical reconstruction:** Classically, when argument reduction has resulted to simple domain splitting, possibly optimized with the method described above [147], the corresponding reconstruction step contains several `if-else` statements to pick the right approximating polynomial for the function evaluation. To make the code vectorizable, branching during the computations of the finite function values has to be avoided.

Consider here that the domain splitting procedure returns a set of non-overlapping intervals $\{I_k\}_{k=0}^{N}$, such that $I_k = [a_k, a_{k+1}]$, so the adjacent intervals have the only point in the intersection $I_k \cap I_{k+1} = \{a_{k+1}\}$. The initial implementation domain is $[a, b] = I = \bigcup_{k=0}^{N} I_k$. Then, to compute $f(x)$ we execute `if-else` statements to determine subdomain's index $k$, where $x \in I_k$.

The code may be written without branches with the use of a mapping function that returns subdomain's index for each input value from $I$:

$$M(x) = k,\ x \in I_k,\ k = 0, 1, \ldots, N.$$

The function $M(x)$ is a piecewise-constant function, as it is shown on Figure 4.6.

For a naive domain splitting, when $I$ is divided into $N$ equal intervals we may use a linear function $m(x) = \frac{N}{b-a}(x - a)$, and then the mapping function is easily computed as $M(x) = \lfloor m(x) \rfloor$. However, this splitting is not optimal, and a more sophisticated splitting algorithm is used instead [147].

— **How to compute polynomial mapping:** For a non-regular domain splitting as the one that is currently used in Metalibm the mapping function may be computed with an

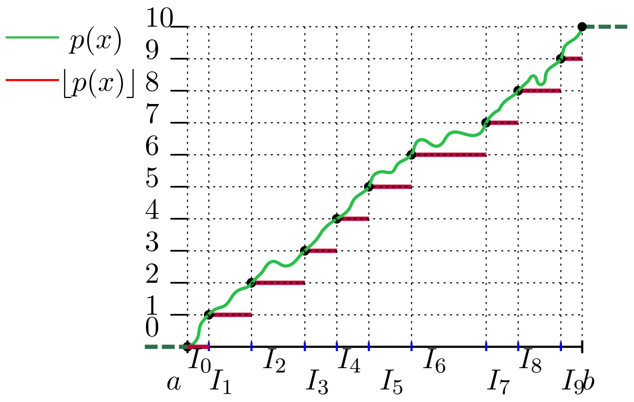

Figure 4.7 – Mapping function and a corresponding polynomial $p(x)$

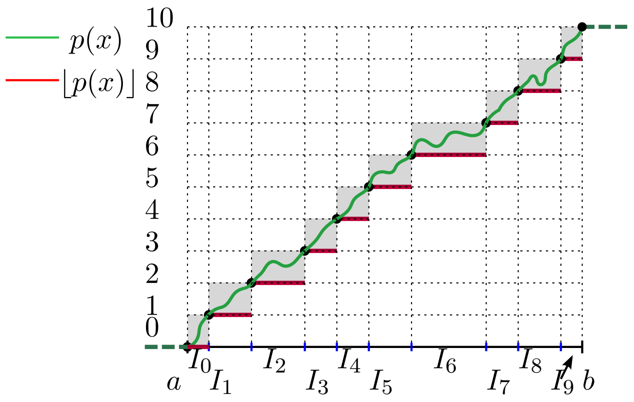

Figure 4.8 – Admissible ranges for the polynomial values.

interpolation polynomial $p(x)$. This polynomial passes through the points $(a_k, k)$, $k = 0, \ldots, N$, where $\{a_k\}$ are the splitpoints. Once the polynomial coefficients are computed, the mapping function can be computed as

$$M(x) = \lfloor p(x) \rfloor, \; x \in I.$$

Thereby, we obtain some conditions for this polynomial. Such a polynomial is shown on Figure 4.7.

— **Conditions for the polynomial:**

As the mapping function stays constant on a subdomain $I_k$, the admissible range for the polynomial values on this subdomain is $[k, k+1)$. Thus, the task is to compute an interpolation polynomial $p$ on the points $(a_k, k)$, $k = 0, \ldots, N$ for which the following holds:

$$p(x) \in [k, k+1), \; x \in [a_k, a_{k+1}]. \tag{4.1}$$

A suitable polynomial $p$ as well as the conditions (4.1) are shown on Figure 4.8. As the classical interpolation procedure guarantees only that $p(a_k) = k$ by construction of the polynomial, conditions (4.1) have to be checked *a posteriori*. This can be done in Sollya [41] with the evaluation of this polynomial $p(x)$ over an interval $[a_k, a_{k+1}]$.

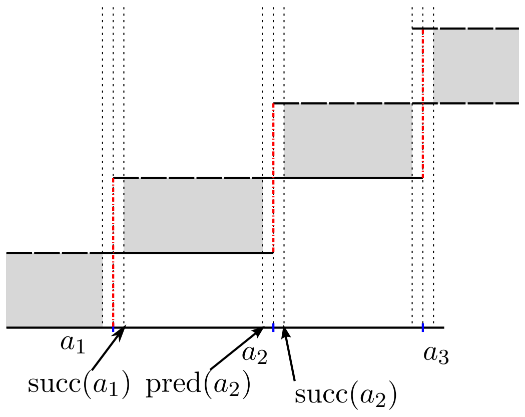

Figure 4.9 – Modified floating-point conditions for polynomial.

There is a certain ambiguity for the values of mapping function in the splitpoints $\{a_k\}$. In splitpoints the two polynomials corresponding to the adjacent subdomains have the same value $p_{k-1}(a_k) = p_k(a_k) = k$. To get the index of the approximating polynomial at the point $a_k$, we may admit $M(a_k) = k - 1$ or $M(a_k) = k$. Only in the "corner" splitpoints $a_0$ and $a_N$ there is no ambiguity for the values of mapping function.

As all the computations are performed in floating-point numbers, the interpolation conditions $p(a_k) = k$ are no longer satisfied because of roundings. Taking into account the ambiguity of the mapping function in the splitpoints, conditions (4.1) have to be modified a little. As the set of floating-point numbers is discrete, for a given floating-point number $a$, it is possible to find its predecessor $\mathrm{pred}(a)$ and successor $\mathrm{succ}(a)$. This means that the admissible ranges for polynomial values from (4.1) should be narrowed to the following:

$$p(x) \in [k, k+1), \text{ where } x \in [\mathrm{succ}(a_k), \mathrm{pred}(a_{k+1})] \subset I_k,\ 0 \leqslant k \leqslant N-1. \quad (4.2)$$

The conditions for the splitpoints should be added then.

$$p(x) \in [k-1, k+1), \text{ where } x = a_k, k = 1, \ldots, N-1. \quad (4.3)$$

The modified conditions for the polynomial ranges are shown on Figure 4.9 with gray rectangles, the range of polynomial values in split points is illustrated with a red line.

— **The choice of the interpolation points:**

The interpolation points may be chosen in several different ways. With the set of splitpoints $\{a_k\}_{k=0}^{N}$ we compute four different polynomials. First, we may use "inner" polynomial with $N-1$ points $\{a_k\}_{k=1}^{N-1}$. Then we can compute "left" and "right" polynomial with $N$ points $\{a_k\}_{k=0}^{N-1}$ or $\{a_k\}_{k=1}^{N}$. And the last variant here is to compute a polynomial of degree $N$ using all $N+1$ splitpoints. When *a posteriori* conditions are not verified for all the four polynomials (4.2) through (4.3), we may add some interpolation points. However, as the addition of new interpolation points raises the degree of the polynomial according to Runge's phenomenon, it will oscillate in the ends, which means that the conditions (4.2) through (4.3) are rarely verified.

— **Towards *a priori* conditions:**
As the conditions for the polynomial values are checked only *a posteriori*, there is no guarantee that the polynomial for mapping function exists for arbitrary splitting. Contrariwise, our method finds it only for few splittings. When there are some points where the polynomial exceeds the admissible range, we can add them to the interpolation points, recompute the polynomial $p(x)$ and recheck the conditions (4.2) through (4.3)). However, due to Runge's phenomenon the polynomial begins to oscillate [37] and the conditions are not verified. The choice of the interpolation points for this polynomial remains an open problem.
However, the ranges for the polynomial values are still checked *a posteriori* and there are some values out of the admissible range. These conditions may be taken into account if we operate intervals instead of points. The classical interpolation problem is a system of linear equations with Vandermonde's matrix:

$$\begin{pmatrix} 1 & x_0 & \cdots & x_1^N \\ 1 & x_1 & \cdots & x_1^N \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & \cdots & x_N^N \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{pmatrix}, \tag{4.4}$$

where $(x_i, y_i), i = 0, \ldots, N$ are the interpolation points and $c_0, \ldots, c_N$ are the unknown polynomial coefficients. Computing the unknown coefficients means solving the system (4.4). When we use intervals instead of points to compute the interpolation polynomial, we take subdomains on abscissas and intervals $[k, \mathrm{pred}(k+1)]$ on ordinates. Then, the task is almost the same: system of linear equations with unknown coefficients $c_0, \ldots, c_N$. Except of the numbers $x_i$, $y_i$ we operate intervals in system (4.5).

$$\begin{pmatrix} 1 & \mathbf{x_0} & \cdots & \mathbf{x_1^N} \\ 1 & \mathbf{x_1} & \cdots & \mathbf{x_1^N} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \mathbf{x_N} & \cdots & \mathbf{x_N^N} \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} \mathbf{y_0} \\ \mathbf{y_1} \\ \vdots \\ \mathbf{y_N} \end{pmatrix} \tag{4.5}$$

The difference from the classical solution of the interval system is that we need only one vector $(c_0, c_1, \ldots, c_N)$, but not the set of all suitable coefficients. We may find the tolerance solution set of the system (4.5) in polynomial time, but it can be empty. In this case the united solution set may be found, but this problem is NP-hard [235] . Anyway, we have connected coefficients in the system matrix, and the existing methods do not take into account this type of connection. We leave this transition to *a priori* conditions for the future work.

**Several examples of code generation**

The two first examples correspond to $f = \exp$ with the target accuracy $\bar{\varepsilon} = 2^{-53}$, polynomial degree is bounded by $d_{max} = 9$. Then we test a toy composite function and a sigmoid function as used in neural networks.

1) For $f = \exp$ on the small domain $I = [0, 0.3]$, the first level is enough. The generated code only consists of polynomial coefficients and polynomial evaluation function. Function

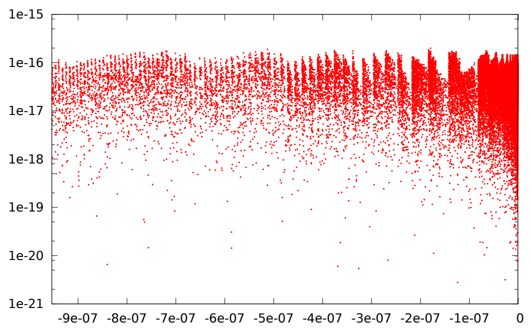

Figure 4.10 – Logarithmic relative error of generated code on domain $[-2^{-20}; -2^{-30}]$ for $\sin(\cos(x/3) - 1)$

generator does not handle special cases for the moment (infinities, NaNs, overflows, etc.), so for larger domains this filtering has to be added manually. This is addressed in Section 4.4.2. This function flavor is about 1.5 times faster than the standard `exp` function from the `glibc libm`.

2) For exponential flavors on larger domains, the table-driven argument reduction mentioned in Section 4.2.2 is used. We enlarge the domain from the previous example to $I = [0, 5]$ and set $w = 4$ for table (the table size is $2^w$).

On the third level, the family of exponential functions is detected. The domain is reduced to $[-\log(2)/2^{w+1}, \log(2)/2^{w+1}]$, where the first level generates polynomial code. The obtained code for this flavor executes in 10 to 60 machine cycles, with most inputs, requiring less than 25 cycles. For comparison, `libm` code requires 15 to 35 cycles.

3) For the composite function $f(x) = \sin(\cos(x/3) - 1)$ on the domain $I = [-1, 1]$ with 48 bits of accuracy, using the standard `libm` will involve two function calls (both with special cases handling) and a division. In the generated code, the composite function is directly approximated by polynomials. Even though the target error is less than the mantissa length for double precision, the generated code wins both in accuracy and performance. Due to cancellation at zero, the error for composed `libm` functions explodes (Figure 4.11), while the error of generated function stays bounded (Figure 4.10). Execution of the generated code takes between 8 and 470 machine cycles, while execution of the `libm`'s analog of this flavor takes more than 150 cycles and may reach 650 cycles in the worst case.

4) Another example is the sigmoid function $f(x) = \frac{1}{1+e^{-x}}$ on the domain $I = [-2; 2]$ with 52 correct bits. No algebraic property is detected, so the generation is done on the second level. The generated code and the `libm`'s code are both of comparable accuracy and performance: execution takes between 25 and 250 cycles with most cases done within 50 cycles. The polynomial degree for the generation is bounded by $d_{max} = 9$, the domain was split to 22 subintervals.

## Open-ended code generation: wrap up and outlook

As shown with the previous examples, and with its own 10300 lines of code, open-ended code generation has reached a certain level of maturity. It is able to quickly generate code for functions defined by various means, including black-box definitions. The produced codes do not yet reach the performance which manual implementation would eventually reach but

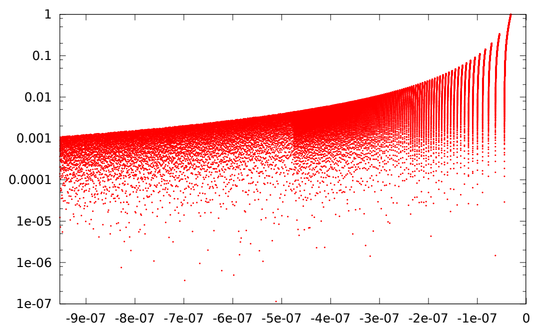

Figure 4.11 – Logarithmic relative error of the `libm` call on domain $[-2^{-20}; -2^{-30}]$ for $\sin(\cos(x/3) - 1)$

they are available quickly, at reduced cost. Final accuracy is bounded by construction and in the case of composite functions, it is better than for plain `libm` function composition.

However, the current prototype has some limitations that we intend to lift. First of all, unless an argument reduction can be detected, code generation is currently limited to small domains. Domain splitting can offset this limitation, but only up to some point. For instance, splitting domains like $[1; \infty]$ or just $[1; 2^{1024}]$ is hopeless. A possible way to address this limitation is to transform $f$ over a large domain $I = [a; b], a \geqslant 1$ into $f(1/x)$ over $[1/b; 1/a]$. However, black-box static error analysis for this case remains to be studied.

Another room for improvement is the back end code generation, currently limited to generic C code. This may be addressed by the framework that we present now.

### 4.2.4 A code generation framework assisting `libm` developers

We now describe a development framework that enables `libm` developers to address the productivity issue. This framework could be used to implement the previous techniques and manage the back end code generation. However, we mostly present it from the point of view of a developer of classical `libm`. As already mentioned, there is no clear boundary between the two approaches depicted on Figure 4.1.

Due to space restrictions, we essentially present and motivate various technical choices. A reader interested in more details is invited to look at (and experiment with) the open-source code available from `www.metalibm.org`.

#### General overview

All the framework is implemented in Python, a language chosen to ensure that the framework is fully scriptable. More importantly, an evaluation scheme is itself described in Python. Technically, one first defines a Python variable as being the output of the generated code, then all computations that eventually affect this variable are considered as belonging to the evaluation scheme.

A similar mechanism enables the designer to embed, in the same Python code, the analysis and proof of numerical properties of the evaluation scheme. First, intervals may be described and manipulated directly in Python using interval arithmetic. This is useful to script range and error analysis. The fact that interval computations are scripted in Python is a non-

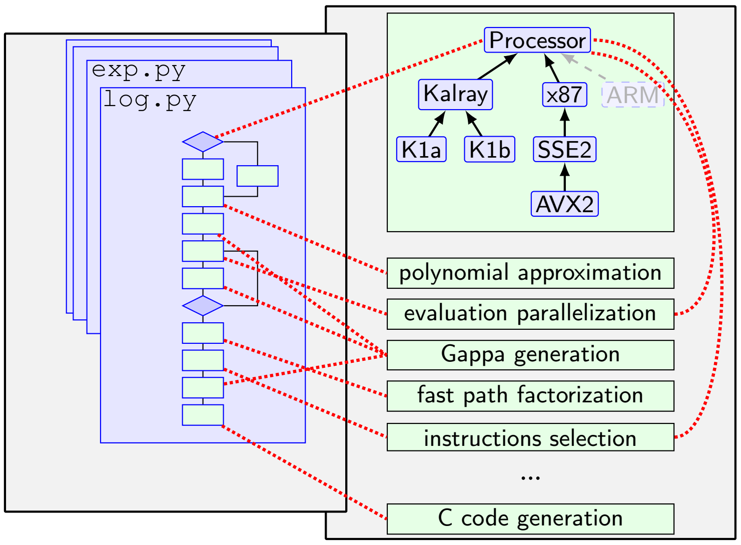

Figure 4.12 – Zoom on the back end C code generator

automatic, but practical way out of the correlation issues that plague naive interval arithmetic. Second, pure mathematical expression may also be described, still in the same classical Python syntax. They can be used for describing what the code is supposed to compute, so that Sollya may compute $\varepsilon_{\text{approx}}$, and Gappa may bound the accumulation of rounding errors [64].

This may seem a lot of overloading for the Python syntax. Below all this, we still also keep standard Python to script the code generation, in several steps (Figure 4.12).

First, the execution of the Python code describing the evaluation scheme builds a corresponding abstract syntax tree (AST). Variables and operations in the AST may be annotated with all sorts of information, either explicitly by the programmer, or automatically by the framework. Examples of annotations include: desired precision, intervals for ranges and errors, probability of taking a branch, exactness of operations (no rounding errors), dependency to FP environment (rounding mode, exception visibility).

Then, the AST may be manipulated from within the same Python code. The framework provides many optimization steps for this. Some of them are similar to those implemented in generic compilers (for instance if-conversion or instruction selection). Some others are specific to the `libm` context (for instance labeling the tree with range and error intervals, or reparenthesizing of arithmetic expressions to express parallelism, similarly to the CGPE approach [194]). It is also possible to write function-specific optimization steps.

An important feature is that these optimization steps are invoked from the same Python code where the evaluation scheme is described. The `libm` designer may therefore choose to apply them or not (depending on the context), and decide on which part of the AST an optimization step is applied. This provides much finer control than the optimization switches of a compiler, while remaining easy to manage because elementary function codes are small. It is therefore a practical way out of the issue of conflicting compiler optimizations.

Among these optimization steps, instruction selection transforms the AST into a form closer to the generated code. It is dependent on the target hardware, as depicted in next section. The generated C is often very close to assembly: each line of C roughly matches one machine instruction, sometimes explicitly thanks to intrinsics if the target decided so.

Python generic code:

```
k = NearestInteger(unround_k, precision = self.precision)
```

NearestInteger is a method of the generic Processor class, and generates the following code for binary32:

```
k = rintf(unround_k);
```

When overloaded in the Kalray processor class, it generates the following code:

```
t = __builtin_k1_fixed(_K1_FPU_NEAREST_EVEN, unround_k, 0);
k = __builtin_k1_float(_K1_FPU_NEAREST_EVEN, t, 0);
```

The same Python also generates binary64 versions, here for for x86 with SSE2:

```
t = _mm_set_sd(unround_k);
t1 = _mm_round_sd(t, t, _MM_FROUND_TO_NEAREST_INT);
k = _mm_cvtsd_f64(t1);
```

Figure 4.13 – Examples of processor-specific code generation

However we still leave a lot to the compiler: the detailed scheduling of the instructions, the register allocation, and the autovectorization.

It is in general a challenge to avoid reinventing the compiler. We want to rely on existing compilers for what they are good at. As compilers improve, some of the technical choices made here will be reevaluated.

### The processor class hierarchy

Modern processors, even within ISA families such as IA32 or ARM, exhibit a lot of variety in their instruction sets. They differ in the basic arithmetic support (presence or not of a fused multiply-and-add, of a divide instruction, of various int/float conversion instructions). They also differ in the model of parallelism and in the capabilities of the hardware to extract this parallelism (some are VLIW, some are superscalar, some support vector parallelism; all are pipelined but the depth and width of the pipeline vary greatly). Finally, they are all increasingly memory-starved, and may offer different strategies to address this issue (caches, memory prefetching, memory access coalescing, etc). Generating optimized elementary function code for such a variety of processors is very challenging. Actually, we want to optimize the code not for a processor, but for a context that includes the processor. For instance, for the same processor, throughput-oriented or latency-oriented code may be very different.

In order to do so, for each processor, we define a class that provides information to optimization steps, as well as code generation services. As recent processors tend to inherit the instruction set of their ancestors, the inheritance mechanism of object-oriented languages works well for overriding old techniques with newer ones when new instructions appear.

The current prototype generates code for four targets from two very different families. The first is a recent x86 processor, an out-of-order superscalar architecture. We consider an SSE2-enabled version (using the modern SIMD FPU that complements the legacy x87 unit), and a more recent AVX2-enabled one (adding hardware support for the FMA). The second family is the K1 core, developed by Kalray for its MPPA manycore processor. It implements a 5-issue VLIW in-order architecture, and comes in two versions: the first, K1A, offers a mixed single precision / double precision FP unit. The second version, K1B, adds binary64 FMA capability and two-way SIMD capabilities for binary32.

There is also a default processor target where no assumption is made on the hardware support, except for IEEE-754 compliance. Support for the ARM family will be added soon.

Figure 4.13 shows one example of code generation service provided by the processor class: the rounding of a FP number to an int.

### Some optimizations performed on the abstract syntax tree

**Instruction selection** Instruction selection is a code generation service provided by the processor class. We could, in principle, delegate it to the compiler that does it well. To illustrate why we want to keep control on it, consider a simple example: fusing one addition and one multiplication into an FMA. This is desirable most of the time, as it improves both accuracy and latency. However, the very fact that it is more accurate impacts the error computation. The code generator therefore needs to know what FMA fusion will be performed. The impact may be deep.

Consider for instance the classical Cody and Waite argument reduction [196], that computes $r = (x - kc_h) - kc_l$ where $k$ is an integer chosen such that the first subtraction cancels. Without an FMA, we need to keep $k$ a small integer, on a few bits, and define $c_h$ as having so many trailing zeroes in its mantissa, so that $kc_h$ may be computed exactly. The subtraction is then Sterbenz-exact. With an FMA, this computation remains exact for much larger $k$ (as long as $k$ may be represented exactly as FP), and there is no need for zero bits in $c_h$, which provides more accuracy in $c_h + c_l$. Here the FMA has an impact not only on the accuracy of the result, but on the relevance domain of the range reduction.

Table 4.1 – Speedups obtained with respect to default libm.

| processor | function | speedup | default `libm` |
|---|---|---|---|
| K1a | expf (binary32) | 4.0 | newlib |
| | logf (binary32) | 2.7 | |
| | exp (binary64) | 5.8 | |
| | log (binary64) | 5.8 | |
| K1b | expf (binary32) | 4.0 | |
| | logf (binary32) | 2.7 | |
| | exp (binary64) | 1.8 | |
| | log (binary64) | 2.2 | |
| core i7, SSE2 | expf (binary32) | 1.7 | glibc |
| | logf (binary32) | 1.02 | |
| | exp (binary64) | 1.7 | |
| | log (binary64) | 1.6 | |
| core i7, AVX2 | expf (binary32) | 1.1 | |
| | logf (binary32) | 0.96 | |
| | exp (binary64) | 1.9 | |
| | log (binary64) | 1.6 | |

All the code variants tested in this table are C11-compliant and optimized for latency. They are generated from the same two files `exp.py` and `log.py`.

Similar arguments apply to other instruction selection situations, such as float/int conversions.

Another interesting example is the fast reciprocal approximation. This instruction, offered by some processors to bootstrap FMA-based division, can also be used in an efficient range reduction for the logarithm [52]. In this case, it must be emulated on processors that do not offer it.

In the near future, we intend to explore using SIMD parallelism to speed up polynomial evaluation. This requires specific instruction to move data within a SIMD vector that are extremely processor-dependent, and actually constrain the evaluation scheme. This is another case for embedding instruction selection in our code generator framework.

**Control path generation and vectorization** Modern compilers are more and more able to autovectorize long, computationally intensive loops. When there are `libm` calls in such loops, these functions must be vectorized themselves.

One approach is to define vector types (which may be exposed to the programmer, but also inferred by the compiler), and to provide vector versions of each `libm` function. This solution is offered for instance by Intel and AMD, at considerable development cost. Note that no standard exists (yet) for functions with vector arguments.

Another approach is to write the `libm` code in such a way that it will autovectorize well. Then, the vectorization itself (i.e. the use of vector instructions) is delegated to the compiler. For this to work, the code must obey certain rules, essentially regarding the tests: tests that translate to branches in the code should be avoided, and replaced with tests that translate to data selection [76]. This entails speculative execution, which typically increases the latency, as both branches of the test must be executed before the selection. However vectorization makes up for this by improving the throughput. Experiments with the Cephes library [209] showed this approach to be extremely effective, both in terms of computational efficiency and portability, with recent versions of the GCC compiler.

This second approach is therefore the one chosen here. For this purpose, there are optimization steps that specifically rework the control path of the function, depending on its intended use. For high-throughput, vectorizable code with speculative execution must be generated. For low latency code, the main objective is to make the common case fast [7], while speculative execution may still be used to exploit the ILP offered by the processor.

Initial experiments with our generator show that autovectorized code on 4-way SSE2 SIMD achieves speedups above 3 over the scalar version. This is consistent with [209].

### A class for `libm` function

If the function is not a black box, but one of the C11 `libm` functions, it comes with a more complete specification, for instance including the management of exceptional numbers in input or output, a significant amount of the function code in a classical `libm`. This is not only a constraint, as it also enables us to use well-known techniques for a specific function.

Another advantage of having an explicit reference to the function concerns its testing. It is possible to generate random tests out of a black-box function, but we can do better if the function is known. It is possible, for instance, to design corner case or regression tests (for instance to check that a function overflows exactly when it should or handle subnormals properly – the Gappa-based proof generation does not cover special values). Also, for random testing, the default random generator can be overridden with a function-specific one that will stress the function on the parts of its domain where it is most useful. This applies to functional testing, but even more to performance testing. For instance, what matters to users of `exp` is its average performance on the small domain where it is defined, not on the full FP range where it mostly returns $0$ or $+\infty$. Similarly, performance tests for `log` should use a pseudo-random number generator strongly imbalanced towards positive numbers. For `sin` it should concentrate on the first periods, but avoid over-testing the very small numbers (which represent almost half the FP numbers) where $\sin x \approx x$.

The proposed framework therefore defines a class for a C11 function with its test infrastructure. Instances of this class (one per function) may overload some methods of this class, such as the generator of regression tests with some default values and one or few generators of random input.

**Wrapping up: current state of the project**

The back end code generator (processor classes, generic optimization steps and proof generators) currently consists of 3800 lines of Python code (counted by the CLOC utility) for the open-source part covering the generic and 86-optimized processor classes. In addition there are also a few proprietary files related to the Kalray processor.

The C11 function generators currently amount to 2482 lines of code. This includes exp, log, log1p, log2, and sin/cos at various degrees of completion (they generate working code, but not always of performance comparable to handwritten code). There are also inverse, division, inverse square root, and square root implementations, essentially intended for processors without a hardware divider, such as the IA64 or Kalray. Inverse approximation is actually of more general use and can replace standard division in situations where it is needed with accuracy lower or higher than the FP precision.

A function generator typically consists of 200-300 lines of code. Considering that the generated code is about 100-200 lines, the Gappa proof about the same size, and that one generator produces many code flavors, the code generator approach should be much easier to maintain.

Near-term future work include adding ARM processors and more functions. Medium-term future work include an OpenCL back end to target GPUs (increasingly used as FP accelerators), more evaluation scheme optimization using CGPE [194], and the generation of correctly-rounded function flavors. An interface with the PeachPy framework [75] should also be investigated.

### 4.2.5 Conclusions and future work

This Section discusses two approaches to addressing the challenges faced by `libm` developers. The first is automated generation of evaluation schemes, to address the large number of functions of interest. The second is a specific development framework, to address the large number of hardware targets of interest. These two approaches were developed independently, and current work focuses on integrating them more. A good case study for working on such an integration is the correct rounding of elementary functions. It presents several challenges, such as function evaluation in larger-than-standard precisions or less common formats, or vectorization of a technique centered on a run-time test [264].

Ideally, we wish we could have a generator where we have a clear separation (as on Figure 4.1) between a front-end building an approximation scheme, and a back end implementing it on a given target technology. A challenge is that the front-end itself must often be directed by the target context. A good example of this is table-based range reduction techniques. Formally capturing this complexity is currently out of reach. Happily, it is always possible, as in tools like ATLAS, to perform an empirical exploration of the parameter space: generate several variants, compile and time then, and pick up the best. This technique could also help with out-of-order processor architectures whose behavior is too hard to model anyway.

Finally, as the codes produced by the current generators are already hardened through the use of Gappa as a formal proof tool, integration of code generation into certified code development suites such as Why, or even model checkers, could be investigated.

## 4.3 Open-Ended Function Generation

In the last Section, we have presented an overview on the different use-cases for code generators for mathematical functions. While we have already hinted at the fact that our open-ended code generator metalibm-lutetia can be applied to functions which are not defined algorithmicly but by some kind of mathematical equation, such as a differential equation, we have not explained in detail how such code generation *ab initio* would work. We shall do so in this Section, which is based on [163]. The combination of software that resulted from this experiment is highly experimental; for this reason we refer to our code generator as Frankenstein.

### 4.3.1 Implementation of Special Functions

Fixed-precision floating-point ("FP") operations come with varying performance and accuracy guarantees. Basic operations such as multiplication are typically implemented in hardware and, being correctly rounded, provide perfect accuracy. So-called elementary functions like $\exp$ and $\log$ are provided in software through general-purpose mathematical libraries (libms). They are well-optimized for performance and either correctly rounded, as recommended by the IEEE754 Standard [117], or provided with maximum error not exceeding one unit in the last place.

This work is concerned with special functions, i.e., "functions which are widely used in scientific and technical applications, and of which many useful properties are known" [94]. Some, like Bessel functions or the Gaussian error function, are present in some libms. Most are provided by specialized libraries such as GSL or Root. Others are only available in computer algebra systems like Maple and Pari and have no efficient fixed-precision implementation. The NIST *Digital Library of Mathematical Functions* [1] contains a good overview of existing implementations.

There is a huge gap in both performance and rigor between these implementations and state-of-the-art elementary functions. However, implementing or reimplementing special functions manually in the classical way is tedious and it requires considerable expertise. And indeed, in many cases, publicly available implementations of a given special function all derive from a single source, typically Cephes [193].

This Section presents the first results of a project exploring the implementation of special functions through automatic *code generation*. In our view, this is the only feasible way to implement wider classes of special functions while bridging the quality gap with respect to elementary functions. Additionally, code generation allows for production of specialized implementations tailored to specific applications, in terms of accuracy, supported domain or code properties.

Our focus is on special functions satisfying linear ordinary differential equations of the form

$$p_r(x)f^{(r)}(x) + \cdots + p_1(x)f'(x) + p_0(x)f(x) = 0, \quad p_i \in \mathbb{Q}[x]. \tag{4.6}$$

Such functions are called *D-finite* or *holonomic*. A key idea in the field of D-finite functions is that an equation such as (4.6) along with initial values makes up a concise exact representation of its solution that can be used in computations.

Many common special functions can be defined this way. Among the better known special functions of mathematical physics, univariate D-finite functions include in particular the error function (and related functions such as the Dawson functions or the Voigt profile), the Airy functions (as well as Scorer functions, generalized Airy functions...), the Bessel and Hankel functions, the Struve functions of integer order, the hypergeometric and generalized hypergeometric functions, the spheroidal wave functions, and the Heun functions.

Under the term *code generator*, we understand a software tool which takes as input a description of a function $f$, a set of floating-point inputs $X$ and a target accuracy $\varepsilon > 0$, and which generates source code (in our case, in C) that provides a function $\widetilde{f}$ satisfying

$$\forall\, x \in X, \qquad \left| \frac{\widetilde{f}(x) - f(x)}{f(x)} \right| \leqslant \varepsilon. \tag{4.7}$$

In this work, we consider inputs in IEEE754 double precision (binary64) format taken in a relatively small interval $[a, b]$, say of width $b - a < 100$, and accuracies compatible with that format, i.e., $2^{-53} \leqslant \varepsilon \leqslant 1$. These restrictions can be loosened with some manual work, so that implementations on the whole set of double precision numbers become feasible.

Code generation for mathematical functions is not entirely a new idea. A powerful toolbox, including in particular Gappa [64] and Sollya [41], has been developed in recent years to help human developers with the most tedious steps of the implementation process. Building on these advances, several authors have considered code generation both for "flavors" of a fixed function and for "black-box" functions given as executable code [30,32,62,148,165]. Independently, Beebe [14] produced a new, very extensive mathematical library with the help of semi-automatic code generation. We base our work on the black-box approach of [165], where the starting point is executable code, able to evaluate $f$ and its first few derivatives on intervals, up to any required precision. Providing such an evaluator is easy when $f$ is a composition of basic elementary functions, thanks to multiple-precision interval libraries, but it is a major limitation for more general functions.

Nevertheless, if $f$ satisfies some kind of functional equation, any rigorous arbitrary-precision numerical solver could in principle play the role of the black box. In the case of D-finite functions, such solvers can be built in practice [198,245]. Moreover, the behavior of D-finite functions in the neighborhood of their singularities is well-understood, opening the door to further extensions of the method. These observations were, in some sense, the starting point of the present work. We soon noticed that the interval black-box model is not well suited to our case, as the number of queries to the function tends to be very large. This led us to favor a different interface, based on polynomial approximations, as described in Section 4.3.2.

This Section is further organized as follows. Section 4.3.2 describes the general architecture of our function generation pipeline and introduces our prototype implementation. Section 4.3.3 focuses on the frontend part, that is, the rigorous ODE solver. Section 4.3.4 discusses the backend, which is the part in charge of floating-point implementation choices. The remaining Section presents experimental results and applications.

### 4.3.2 From Differential Equations to C-Code

This text describes both an approach to code generation for D-finite functions and experiments performed with a prototype implementation of this approach. Our prototype, named Frankenstein, is available from `https://gforge.inria.fr/git/metalibm/frankenstein.git`. It is based on modified versions of two existing software packages, Metalibm [148], written in Sollya [41], and NumGfun [189], written in Maple. The name should give a pretty accurate idea of how the combination is realized. Thus, not all design choices have the same significance: while some would survive a cleaner rewrite, others are only justified by the goal of producing a usable prototype out of two research-quality codes that had never been intended to work with each other. Here we focus on the former category, with brief mentions of more peculiar features of Frankenstein as necessary.

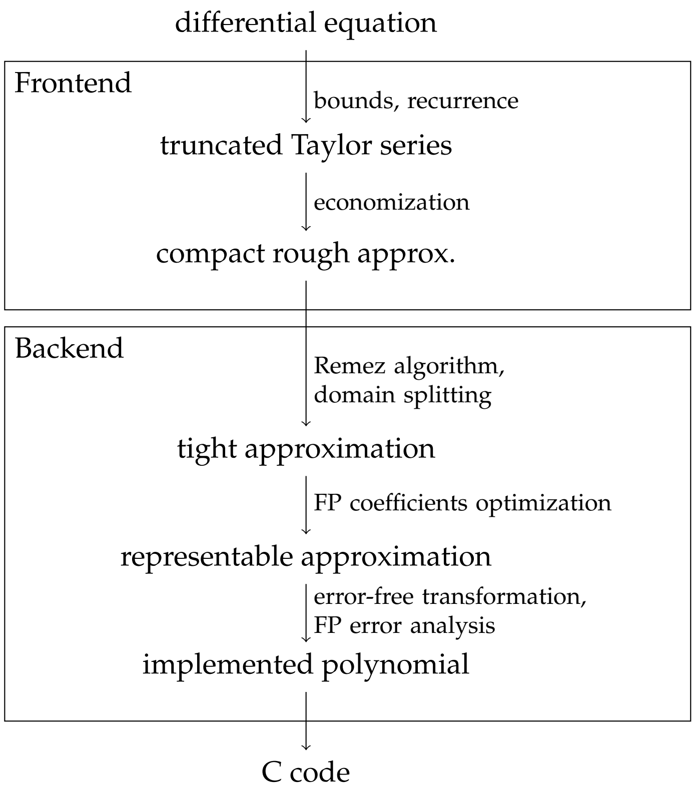

Figure 4.14 – A piecewise polynomial approximation pipeline.

Consider a sufficiently small interval $I = [a, b] \subset \mathbb{R}$ and a solution $f$ of (4.6) defined on $I$. Assume for simplicity that $0 \in I$ and $p_r(x) \neq 0$ for $x \in I$. The function $f$ is then analytic on $I$, and the vector of initial values $F(0) = (f(0), \dots, f^{(r-1)}(0))$ characterizes $f$ among the solutions of (4.6). Assume additionally that $|f(x)| \leqslant 2^{1024}(1 - 2^{-53})$ for $x \in I$. Then, special floating-point values (NaN, $\pm\infty$) occur neither on input nor on output. Denoting by $\mathbb{F}$ the set of double-precision numbers, our goal is to produce a program computing a value $\widetilde{f}(x) \in \mathbb{F}$ satisfying (4.7) with $X = I \cap \mathbb{F}$.

Our method takes as input $(p_i)_{i=0}^{r}$ and $F(0)$ (which together define $f$), $I$, $\varepsilon$, and various constraints on allowable implementations. It either succeeds and produces a program satisfying (4.7) by construction [4], or fails if no implementation fitting the constraints is found. Failure does not imply that no feasible solution exists.

One way to view the process leading from the definition of $f$ by (4.6) to an implementation is as a rigorous piecewise polynomial approximation pipeline, as illustrated on Figure 4.14. Each stage receives a set of approximations of $f$ (and possibly its first few derivatives) by polynomials on subintervals of $I$ and produces approximations with different properties that are passed on to subsequent stages. The subintervals vary from stage to stage, and typically overlap even within a single stage. For example, the domain splitting procedure can use both a rough approximation on the whole domain to get a general picture of the behavior of $f$, and tight approximations on tiny intervals to compute precise values.

The complete pipeline can be approximately divided into a frontend and a backend. Roughly speaking, the frontend is the part where decisions are driven by the analytic properties of $f$. The backend is the part where they are driven by the floating-point environment and other implementation constraints. To work with another class of mathematical functions, one would replace the frontend; to target a different evaluation environment (fixed-point arithmetic, say), one would swap out the backend. There is no formal abstraction of how the backend can query the function, though, and both parts have full knowledge of the implementation problem.

In Frankenstein, the frontend and the backend respectively correspond to NumGfun and Metalibm. For convenience reasons, the process is driven by the backend, which can ask the frontend for approximations of variable quality on various subintervals. As we reuse code written for the black-box function model, the backend mostly uses these approximations to perform interval evaluations, but we expect to make more direct use of the polynomial representation in future developments.

Our general implementation strategy fits the standard pattern of special function implementations and is especially close to that of Harrison [109]. Classical argument reduction algorithms based on algebraic properties of elementary functions typically do not apply. Accordingly, the only feature of $f$ that we consider to reduce the implementation domain is its parity. Parity properties can be detected either by numerical comparison of $f(x)$ and $f(-x)$ (as Frankenstein currently does), or based on the differential equation. Periodic functions could be handled in a similar way but are very uncommon.

We then single out a small number of "interesting points", the neighborhood of which need to be handled in a special way. Under our working hypotheses, interesting points include $x = 0$, where the floating-point grid is denser than usual, and the points $x$ with $f(x) \simeq 0$, where obtaining a relative error bound requires special care. In a more general setting, one would add at least $\pm\infty$ and the singularities of $f$ to the list. In a small interval around each interesting point, approximations of $f$ are computed and implemented in a way that takes into

4. This is not entirely true of Frankenstein yet. Although the overwhelming part of the code uses rigorous error bounds, some heuristic estimates remain.

account the special constraints. The remaining subintervals (if any) are further subdivided until approximation by small-degree polynomials becomes feasible.

Let us now consider in more detail the main ingredients of the process, starting with the frontend.

### 4.3.3 From a differential equation to rigorous approximations

The frontend's role is to provide the backend with rigorous polynomial approximations of $f$ of the form

$$\forall x \in J, \quad |f(x) - p(x)| \leqslant \eta \tag{4.8}$$

for various subintervals $J \subseteq I$. As Equation (4.8) suggests, in our architecture, the approximations come with *absolute* error bounds. Indeed, these are often easier to obtain and this choice does not hinder the use of relative error bounds in later stages.

Values of $J$ above can range from point intervals $J = \{x\}$ to $J = I$. It is not necessary that the frontend be able to find approximations on arbitrarily wide intervals, as the backend has the necessary logic to split $J$, if the returned error bound is infinite (or too large). The only requirement is that sufficiently precise queries on sufficiently thin intervals eventually produce satisfying approximations.

The smallest useful value of $\eta$ in our setting is slightly less than the smallest subnormal number, $2^{-1074}$. In principle, it would be possible to replace $f$ once and for all by a set of polynomial approximations achieving such an accuracy. The frontend of Frankenstein is more flexible and it can compute arbitrarily good approximations. Precise polynomial approximations on wide intervals quickly become large and costly to compute. For historical reasons, our frontend provides a separate interface dedicated to points and very thin intervals that always returns "polynomial approximations" reduced to constants, which can reach absolute precisions $\log \eta^{-1}$ in the hundred of thousands if necessary.

The computation of the approximations (4.8) is based on a combination of classical techniques that we now summarize. We refer the reader to [189,245] for details.

#### Transition matrices

Recall that $f$ is specified by the differential equation (4.6) and the vector of initial values $F(0)$, or, to be precise, rigorous arbitrary precision approximations of $F(0)$. Let us also extend the notation $F(x) = (f(x), \ldots, f^{(r-1)}(x))$ to arbitrary $x$. Polynomial approximations on an interval $J \subseteq I$ will be derived from the Taylor expansion of $f$ around the center $c$ of $J$. This expansion is easily computed from the differential equation and the "initial value" $F(c)$. To deduce $F(c)$ from $F(0)$, we use Taylor expansions at intermediate points $x_0 = 0, x_1, \ldots, x_n = c$ chosen in such a way that $x_{k+1}$ lies within the disk of convergence of the Taylor expansion of $F$ at $x_k$. In other words, our frontend is essentially a rigorous ODE solver based on the so-called method of Taylor series [187].

More precisely, we proceed as follows [47,245]. By linearity of (4.6), for all $x, y \in I$, there exists a matrix $\Delta_x(y) \in \mathbb{R}^{r\times r}$ depending only on (4.6) (not on the particular solution $f$) and such that $F(y) = \Delta_x(y)\, F(x)$. We have $\Delta_x(z) = \Delta_y(z)\, \Delta_x(y)$ for all $x, y, z$, and hence

$$F(c) = \Delta_0(c)F(0) = \Delta_{x_{n-1}}(c) \ldots \Delta_{x_1}(x_2)\, \Delta_0(x_1)\, F(0). \tag{4.9}$$

The entries of $\Delta_x(y)$ are values at $y$ of solutions of (4.6) corresponding to unit initial values at $x$ (or derivatives thereof). Denoting by $\rho(x)$ the distance from $x$ to the nearest complex root of the leading coefficient $p_r$ of (4.6), the Taylor expansion at $x$ of any solution of (4.6) has radius of convergence at least $\rho(x)$. It is a classical fact that the coefficients of these expansions obey linear recurrences with polynomial coefficients, making it easy to compute as many terms as needed. As we assumed that $p_r$ does not vanish on $I$, computing $F(c)$ reduces to forming the product (4.9), where each factor $\Delta_x(y)$ can be evaluated by summing convergent power series, whose coefficients are easy to compute. Binary splitting can be used to compute the partial sums efficiently to very high precisions [47].

Our assumption that the initial values are provided at a point of $I$, is artificial: the above argument still works with $0 \notin I$ as long as the leading coefficient $p_r(x)$ of (4.6) does not vanish between $0$ and $I$. In addition, (4.6) actually defines $f$ for complex values of $x$, and nothing prevents the path $(x_0, \dots, x_n)$ from going through the complex plane. If $p_r(s) = 0$ for some $s$ between $0$ and $I$, analytic continuation along a path avoiding $s$ still defines $f$ on $I$ (in general in a way that depends on the path). Furthermore, when $0 \notin I$, we can relax the assumption that $p_r(0) \neq 0$: the theory extends to the case where $0$ is a so-called regular singular point of (4.6) [189,246]. Both of these extensions are supported in Frankenstein, albeit with specific tool setup and subject to heuristic error estimates in some cases. The second one is useful because many classical special functions are best characterized by their behavior at regular singular points of their defining equation. A typical example is the family of Bessel functions (see Example 3 below).

**Error bounds**

It is crucial that the frontend provides rigorous error bounds on the approximations it computes, as these approximations are the backend's only access to the function $f$. Here, we recall a simple bound computation technique based on the theory of majorizing series [114, Chap. 2]. Frankenstein (via NumGfun) actually uses a more sophisticated variant of the same technique [191], but the present version conveys the main ideas and it would probably suffice for our purposes.

Consider a power series with matrix coefficients (or, equivalently, a matrix of power series) $Y(x) = \sum_{n=0}^{\infty} Y_n x^n \in \mathbb{R}^{r\times r}[[x]]$, and let $\|\cdot\|$ denote a matrix norm. We write $Y \preccurlyeq w$ if $w(x) = \sum_{n=0}^{\infty} w_n x^n \in \mathbb{R}_+[[x]]$ is a power series that bounds $Y$ coefficient-wise, i.e., $\|Y_n\| \leqslant w_n$ for all $n \in \mathbb{N}$. The method transfers such bounds on the coefficients of differential equations to similar bounds on the solutions.

Our goal is to control the error committed by truncating the Taylor expansions of a solution of (4.6). Without loss of generality, we restrict ourselves to Taylor expansions at the origin. Rewrite (4.6) in matrix form, as

$$Y'(x) = P(x)Y(x), \tag{4.10}$$

where $P \in \mathbb{Q}(x)^{r\times r}$ is a companion matrix with entries of the form $p(x)/p_r(x)$, and for notational simplicity $Y$ is also taken to be a matrix. The matrix function $P$ admits a convergent power series expansion at 0, whose coefficients $P_n$ satisfy $||P_n|| = \mathrm{O}(\alpha^n)$ for all $\alpha > \rho(0)^{-1}$. Given such an $\alpha$, it is not too hard to compute $M > 0$ such that

$$P(x) \preccurlyeq q(x) := \frac{\alpha M}{(1-\alpha x)}.$$

Expanding both sides of (4.10) in power series and collecting the matching powers of $x$, we obtain

$$(n+1)Y_n = \sum_{j=0}^{n} P_n Y_{n-j}. \tag{4.11}$$

The same reasoning applied to the equation $w'(x) = q(x)w(x)$ yields

$$(n+1)w_n = \sum_{j=0}^{n} q_n w_{n-j}. \tag{4.12}$$

Now assume $w_0 \geqslant \|Y_0\|$. Comparing (4.11) with (4.12), we see by induction that $Y(x) \preccurlyeq w(x)$. But the second equation is solvable in closed form: we have

$$w(x) = \frac{w_0}{(1-\alpha x)^M}.$$

This explicit expression makes it easy to bound the tails $w_n x^n + w_{n+1}x^{n+1} + \cdots$, and hence also $Y_n x^n + Y_{n+1}x^{n+1} + \cdots$, which yields truncation orders that guarantee a certain accuracy.

**Polynomial approximations**

We are now ready to combine the results of the previous sections to obtain rigorous polynomial approximations on a given interval $J$. Write $J = [c-\delta, c+\delta]$, and assume that $\delta < \rho(c)$. With the notation of Section 4.3.3, we have

$$F(c+\xi) = \Delta_c(c+\xi)\, \Delta_{x_{n-1}}(c) \ldots \Delta_{x_1}(x_2)\, \Delta_0(x_1)\, F(0) \tag{4.13}$$

for $|\xi| \leqslant \delta$. Each factor (except $F(0)$) is given by power series that we can truncate so as to guarantee a prescribed accuracy. Error bounds on the individual factors are combined by repeated use of the inequality

$$\|\widetilde{A}\widetilde{B} - AB\| \leqslant \|\widetilde{A}\|\, \|\widetilde{B} - B\| + \|\widetilde{A} - A\|\, \|B\|.$$

Once we replace each series by a truncation (and $F(0)$ by an approximation with rational entries), the entries of (4.13) become polynomials in $\xi$ with rational coefficients.

Thus, we compute the entries of $\Delta_c(c+\xi)$ truncated to a suitable order as polynomials in $\xi$ and multiply the resulting matrix by an approximation of $\Delta_0(c)\, F(0)$. The entries of the result readily provide the first $r$ derivatives of $f$. Higher-order derivatives, if needed, can be obtained by multiplying $F(c+\xi)$ on the left by a row vector of rational functions (or polynomial approximations of rational functions) deduced from (4.6). In Frankenstein, most of these steps are performed exactly, using multiple precision rational arithmetic. Roundoff errors in the remaining steps are taken into account in the result.

Taylor series typically do not provide good approximations on intervals. Additionally, the bounds of Section 4.3.3 can be quite pessimistic. For these reasons, polynomials, computed as outlined above, tend to have very high degree. However, due to the way they are constructed, the rescaled coefficients $c_n\delta^n$ quickly decrease to zero, and typically $|c_n\, \xi^n| \ll \eta$ for large $n$. Before handing it to the backend, the frontend hence reduces the degree of the computed polynomial $p$ by *economization* [87, §4.6]: while the leading term of $p$ is small compared to $\eta$, a multiple of the Chebyshev polynomial $T_n$ (rescaled to the interval $J$) chosen so that the leading terms cancel is subtracted from $p$. The error bound is updated accordingly. (The choice of Chebyshev polynomials makes it possible to take advantage of the fact that the error bound only needs to hold for $x \in J$, not for complex $x$ with $|x - c| < \delta$.) This procedure is very effective at producing polynomials of reasonable size that can easily be manipulated by the backend.

### 4.3.4 From rigorous approximations to evaluation code

It is the backend's job to transform the high-degree, rough approximation produced by the frontend into fine-tuned approximations with suitable FP properties and eventually, into source code. In absence of classical argument reduction techniques, implementation of the function is reduced to piecewise polynomial approximation. The backend hence needs to compute four pieces of information, as discussed in the next Sections.

1. The implementation interval $I$ needs to be split into subintervals $I_k$, together covering $I$, such that an approximation polynomial of small degree is possible over each $I_k$.
2. These small degree polynomials need to be computed, initially with real coefficients.
3. Connected with the approximation step is the problem of choosing an appropriate translation $f(t_k + \cdot)$ on a new domain $I_k - t_k$. Indeed, the evaluation of the approximation behaves better when $I_k - t_k$ is a small interval around $0$.
4. The backend needs to transform the small-degree approximation polynomial on each subdomain into a polynomial with FP coefficients, suitable for evaluation in FP arithmetic, and generate code for that FP-based polynomial evaluation.

#### Domain splitting

Given a target accuracy $\varepsilon$ and a maximum degree $d$, the backend starts with computing a list of intervals $I_k$ touching each other in so-called split-points $s_k$, i.e. such that $I_{k-1} \cap I_k = \{s_k\}$, covering the whole interval $I = \bigcup_k I_k$ and such that for each $k$ there exists a polynomial $p_k \in \mathbb{R}[x]$ of degree no larger than $d$ approximating $f$ on $I_k$ with a relative error at most $\varepsilon$:

$$\forall\, k,\, \forall\, x \in I_k, \quad \left| \frac{p_k(x) - f(x)}{f(x)} \right| \leqslant \varepsilon. \tag{4.14}$$

Such a splitting can be computed using an algorithm [148] based on bisection, interpolation of $f$ in Chebyshev nodes and application of de la Vallée-Poussin's theorem [38]. In Frankenstein, we leverage the existing Sollya procedure `guessdegree` [41] to implement this step.

#### Small degree polynomial approximation

On each subdomain $I_k$, the backend computes a polynomial approximation $p_k \in \mathbb{R}[x]$ of degree at most $d$ for the function $f$. The $p_k$ are computed as *minimax* approximations with relative error, using a modified Remez-Stiefel algorithm [38]. However, these polynomials with real coefficients [5] are not immediately suitable to IEEE754 FP evaluation. Several problems arise.

5. As the underlying Sollya environment is FP-based, any polynomial we can exhibit has of course FP coefficients. However, as precision may be increased at will at this generation stage, the polynomials appear to have real coefficients.

When $0 \notin I_k$, in particular when $I_k$ contains values $x \in I_k$ that are significantly larger than $1$, Horner evaluation (or any other Estrin-like evaluation) may behave badly. Roughly speaking, at each step $q_{i+1}(x) = c_i + x \times q_i(x)$, the evaluation error accumulated on $q_i(x)$ will be amplified by multiplication with $x$ when $|x| \gg 1$, not attenuated [165]. After splitting $I_k$, if its radius is too large, we translate both the function $f$ and the interval $I_k$ by $t_k$ to always be in the case when all $x \in I_k - t_k$ stay small. In most cases, taking (a rounded value of) the midpoint of $I_k$ is appropriate, while ensuring that the translated argument $\xi = x - t_k$ can be computed exactly in FP arithmetic thanks to Sterbenz' lemma [165,238]. Special care is used when $f$ has a zero in $I_k$ (see Section 4.3.4). Further, $t_k$ may be optimized to take into account FP effects on the coefficients of the polynomial [165].

When $I_k$ is small and the function $f$ is symmetrical around some point in $I_k$, the minimax approximation polynomial $p_k$ tends to reflect the symmetry: some monomials have very small coefficients compared to others. This effect may hinder FP Horner evaluation due to catastrophic cancellation but it can also be exploited to reduce the number of non-zero coefficients of the polynomial [160].

**Achieving relative error bounds for functions with zeros**

Clearly, the ratio $(p_k(x) - f(x))/f(x)$ can stay bounded only if $f$ has no zero in the domain for $x \in I_k$, or if $p_k$ has a zero wherever $f$ has one. Classical polynomial approximation theory either considers absolute error bounds or excludes functions with zeros. It can be extended to cover approximation with relative error of functions $f$ with $f(0) = 0$ as follows: when $f$ is known to have a zero of multiplicity $m$ at $x = 0$, one computes a polynomial approximation $q_k$ of $x^{-m} f(x)$ and takes $p_k(x) = x^m \cdot q_k(x)$. The only requirement is not to use Remez' original minimax algorithm but an enhanced version by Stiefel [38].

Since the backend anyway uses approximation domains $I_k - t_k$ containing $0$, the technique above already makes it possible to handle functions with a zero at some FP number. It suffices to take $I_k$ small enough to contain a single zero of $f$ and $t_k$ equal to that zero in order to obtain $f(t_k + \xi) = 0$ for $\xi = 0$. The value of $t_k$, can easily be computed with any numerical solver, such as Newton's iteration. No rigor is required in this step; if the zero is not correctly determined, relative error polynomial approximation will simply fail.

The technique no longer applies when $f$ has a zero $c \notin \mathbb{F}$. However, in this case, $f$ is actually never zero on the FP numbers it is to be evaluated at. This means that even if the ratio $(p_k(x) - f(x))/f(x)$ is unbounded for $x \in I_k$, it stays bounded over $I_k \cap \mathbb{F}$. Provided that the procedure we use to bound the relative error only takes into account FP numbers, approximation of a function $f$ with non-FP-representable zeros boils down to two subproblems: computing a minimax polynomial for $f$ and translating the original function such that FP-based polynomial evaluation will exhibit bounded evaluation error.

The minimax polynomial can be computed as follows: the function $g(x) = f(c + x)$ has an exact (FP-representable) zero at $0$, and can be evaluated at any desired precision by computing $c$ with a rigorous Newton algorithm. In the Sollya framework, $g$ is implemented by an expression tree containing a constant function $c$ whose evaluation algorithm searches the original domain $I_k$ for a zero of $f$ whenever and at whatever precision needed. This is enough for the Remez-Stiefel minimax algorithm to be applicable to $g$.

Once $c$ itself and a polynomial $p(x)$ approximating $g(x)$ over $I_k - c$ are known, we set $t_k$ to $c$ rounded to the nearest FP number. The polynomial $r(x) = p(t_k - c + x)$ then approximates $f(t_k + x)$ over $I_k - t_k \ni 0$. At $x = t_k$ (the FP value nearest to $c$), the translated argument

$\xi = x - t_k$ is exactly zero, hence FP Horner evaluation of $r$ will return the constant coefficient of $r$, which is equal to (a rounded version of) $f(c)$. For the FP $x$ just after the zero, $\xi$ will fit on a small number of bits due to cancellation in $\xi = x - t_k$, hence FP Horner evaluation will behave correctly, too. The accuracy of evaluation can be checked statically using Gappa as explained below.

Note, though, that the polynomial $r$ passed to the code generation step then contains coefficients involving the Newton-iteration based representation of $c$. In contrast, in the usual case (when $f$ has no zero in the domain), the coefficients directly come from the Remez-Stiefel minimax algorithm.

**FP code generation for polynomials**

Finally, the backend needs to generate a code sequence, based on IEEE754 FP arithmetic, for each subdomain $I_k$, and to connect these code sequences through a branching sequence determining the appropriate subdomain given an input $x \in I$. Generating the subdomain determination sequence is easy.

Generating the polynomial evaluation is harder. The backend takes the following four sub-steps.

1. It determines the minimal precisions needed for the coefficients of the approximation polynomial $p_k$ on each subdomain, following the approach described in [165].
2. It replaces the polynomial $p_k \in \mathbb{R}[\xi]$ with a polynomial $\widetilde{p}_k \in \mathbb{F}[\xi]$ with FP coefficients. This could be done by rounding the real coefficients, however, equi-oscillation properties of the minimax polynomial and therefore accuracy might be affected too much. We therefore use a technique [38] based on lattice reduction which globally searches for a FP-valued-coefficients polynomial "close" to the original real-coefficients polynomial.
3. Starting with the target accuracy $\varepsilon$, the backend estimates suitable accuracies for the intermediate steps of a Horner evaluation of $\widetilde{p}_k$. This choice of accuracies determines the FP precisions—and, possibly, double-double or triple-double expansions—to be used in the different steps. The technique is described in [160,165]. The backend then outputs C code for polynomial evaluation along with a Gappa [64] proof script.
4. Finally, the backend runs the proof script using Gappa, hence verifying the correctness of the accuracy estimates established in the previous step.

**Evaluation-based vs. polynomial-based interfaces**

In a cleanly designed special function code generator, the backend would take the rough polynomial approximations produced by the frontend and proceed with refining them without ever returning back to the original function $f$. However, this would require the frontend to take into account the relative error due to replacing $f$ by a polynomial, which requires some understanding of FP properties of the eventual implementation, when $f$ has zeros in the domain.

For that reason and for reasons due to the reuse of existing code, Frankenstein works in a slightly different way. The backend binds a Sollya function object, corresponding to $f$, to the evaluator dedicated to tiny intervals provided by the frontend. This would in principle be enough for the backend to run, since the backend—in all steps, even its polynomial approximation ones—is purely based on (interval or point) evaluation. In practice, this

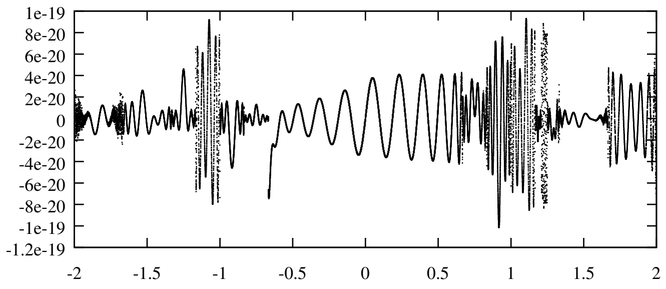

Figure 4.15 – Relative overall error of generated implementation of $\mathrm{erfc}$ on $[-2; 2]$.

simple interface is not enough. The number of function evaluations is too large to allow for reasonable code generation performance. In addition, an evaluation-based interface makes it hard to capture non-local properties of the function $f$, such as monotonicity.

We therefore enhanced Sollya to allow a function object, such as the one bound to the frontend's representation for $f$, to be annotated with an alternate approximate representation along with a bound on the approximation error. The approximation polynomials generated by the frontend provide such an approximate representation. When it needs to evaluate the function $f$ at some point or over some interval, Sollya first tries to use the annotation. If this first try is inconclusive, for instance if the approximation error is too large in view of the evaluation precision, it falls back to the original evaluator bound to the frontend. Interval evaluation using the annotation exploits global properties of the polynomial. In particular, monotonicity over the whole annotation interval is detected once and for all and later used to reduce interval evaluations on subintervals to evaluations at their endpoints.

That combination has given Frankenstein the necessary performance. We should mention however that making the annotation mechanism usable for our purposes required some fine-tuning in the Sollya architecture. In particular, as we have seen, the backend manipulates not only $f$ itself, but also expression trees containing compositions of $f$ with other functions, which need to be reflected on any polynomial annotation. In addition, code written in the evaluation-based model tends to perform high-precision evaluations with no real need (that is, to request information on $f$ that cannot have any influence on the correctness of the generated code). The whole benefit of the polynomial annotations can be lost if too many evaluations fall back to a slow arbitrary-precision evaluator that will try to satisfy these requests.

### 4.3.5 Numerical Results

#### Bounded intervals

As already mentioned, there is no complete guarantee that Frankenstein succeeds to implement a given D-finite function. The following experiments show how it performs on classical special functions. All these examples were handled automatically, with minimal manual setup. The reported timings were measured on a typical desktop computer.

We start with a simple example where the functions stays away from zero on the whole implementation domain.

**Example 1.** *The complementary error function* $\mathrm{erfc}(x) = 1 - 2\pi^{-1/2} \int_0^x e^{-t^2} \mathrm{d}t$ *can be defined* *as the solution of*

$$f''(x) + 2xf'(x) = 0, \quad f(0) = 1, \quad f'(0) = -2/\sqrt{\pi}.$$

*We implemented* $\operatorname{erfc}$ *in the range* $I = [-2; 2]$*, with a target accuracy* $\varepsilon = 2^{-62}$*, under the constraint that no generated polynomial have more than* $14$ *non-zero coefficients. Such a target accuracy slightly beyond double precision is typical for first phases of correctly rounded implementations [165,197]. Though we excluded them above, our code generator is able to handle such accuracies in simple cases, automatically introducing double-double expansions [165,236], as necessary.*

*Figure 4.15 shows a plot of the relative error* $\widetilde{f}(x)/f(x) - 1$ *of the implementation. Code generation took around* $780$ *seconds. The code generator split* $I$ *into* $16$ *subdomains of varying length. In the subdomain* $I_7 = [-2/3; 2/3]$*, the only nonzero coefficients of the generated polynomial are those of odd degree and that of degree* $22$*, taking advantage [160] of an approximate symmetry around* $0$*. Accordingly, the Horner evaluation code for* $x \in I_7$ *performs evaluations in* $x^2$ *in all but one steps. When executed, our implementation takes between* $110$ *and* $350$ *machine cycles per call, with most calls completing in* $260$ *cycles. For comparison, a call to a typical libm exponential takes around* $80$ *cycles.*

We then consider an example with several zeros, none of which is representable in floating-point.

**Example 2.** *The Airy function* $\operatorname{Ai}$ *satisfies*

$$\begin{cases} \operatorname{Ai}''(x) - x \operatorname{Ai}(x) = 0, \\ \operatorname{Ai}(0) = 3^{-2/3}\Gamma(2/3)^{-1}, \quad \operatorname{Ai}'(0) = -3^{-1/3}\Gamma(1/3)^{-1}. \end{cases}$$

*We implemented* $\operatorname{Ai}$ *on* $I = [-4.5; 0]$*, asking for an accuracy* $\varepsilon = 2^{-45}$*. Frankenstein takes* $280$ *seconds to generate an implementation. It splits the domain into* $10$ *subdomains. In two subdomains polynomials with some zero coefficients are used; the generated code precomputes* $x^2$ *accordingly. It can be checked that all subdomains containing zeros of* $\operatorname{Ai}(x)$ *are translated by an amount equal to the double-precision number nearest to the zero, and an error plot confirms that the target accuracy is met even in the neighborhood of the zeros. Evaluation of the generated function takes* $60$ *to* $90$ *cycles, with most calls completing in* $72$ *cycles.*

Our last example directly parallels Harrison's implementation of Bessel functions for "small" arguments [109, Section 3].

**Example 3.** *The Bessel functions* $J_0$ *and* $Y_0$ *are defined as solutions of Bessel's equation*

$$xf''(x) + f'(x) + xf(x) = 0. \tag{4.15}$$

*We consider the function* $J_0$ *on the interval* $I = [0.5; 42]$ *with a target accuracy of* $\varepsilon = 2^{-45}$*. The immediate neighborhood of* $0$ *is excluded because* (4.15) *is singular at* $x = 0$ *(its leading coefficient vanishes). For the same reason, initial values* $f(0)$*,* $f'(0)$ *do not make sense for an arbitrary solution of* (4.15)*. However,* $J_0$ *can still be defined as the unique solution whose value tends to* $1$ *as* $x \to 0$*. As mentioned in Section 4.3.3, Frankenstein's frontend supports such generalized initial conditions.*

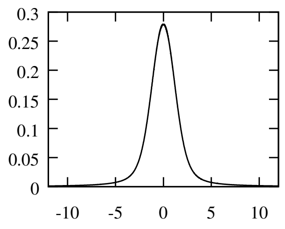

Figure 4.16 – $V(x)$ for $\sigma = 1$, $\lambda = \frac{1}{2}$.

*Code generation starting from this specification took about* 33 *minutes. The code generator splits the domain $I$ into* 18 *subdomains. The approximation polynomials in all subdomains have non-zero coefficients. For some subdomains containing zeros of $J_0'(x)$, the implementer chooses to store the coefficients as double-double expansions, even though it rounds the final result to double precision. Evaluation of the generated implementation takes between* 60 *and* 500 *machine cycles, with most calls completing in* 75 *cycles.*

**Implementation on the whole real line: an example**

In some cases at least, automatically generated implementations on bounded intervals can be combined to implement a special function on the whole set of double-precision floating-point numbers. We illustrate how this can be done on a simple example. The manual steps we take are really an instance of a method of some generality, but we leave it to future work to handle such cases without human intervention.

The Voigt profile $V(x)$ (Figure 4.16) is a probability distribution used in particular in spectrography, and whose computation has been the subject of abundant literature [234, 243]. It depends on two parameters $\lambda, \sigma > 0$ and it is defined for $x \in \mathbb{R}$ as a convolution of a Gaussian distribution and a Cauchy distribution,

$$V(x) = \frac{1}{\sigma\sqrt{2\pi}} \frac{\lambda}{\pi} \int_{-\infty}^{+\infty} \frac{\exp \frac{-x^2}{2\sigma^2}}{(x-t)^2 + \lambda^2} \mathrm{d}t.$$

A change of variable yields the alternative expression

$$V(x) = \frac{1}{\sigma\sqrt{2\pi}} \frac{\lambda}{\pi} \int_{-\infty}^{+\infty} \frac{\exp \frac{(u-x)^2}{2\sigma^2}}{u^2 + \lambda^2} \mathrm{d}u, \tag{4.16}$$

and it is not hard to see that (4.16) satisfies

$$\begin{cases} \sigma^4 V''(x) + 2\sigma^2 V'(x) + (x^2 + \lambda^2 + \sigma^2) V(x) = \dfrac{\lambda}{\pi}, \\ V(0) = \dfrac{1}{\sigma\sqrt{2\pi}} \exp\left(\dfrac{\lambda^2}{2\sigma^2}\right) \operatorname{erfc}\left(\dfrac{\lambda}{\sigma\sqrt{2}}\right), \quad V'(0) = 0. \end{cases} \tag{4.17}$$

(To remain in our general setting, a homogeneous ODE can be derived by differentiating one more time.)

The Voigt profile with arbitrary parameters can be expressed in terms of the Faddeeva function, itself a renormalization of the complex error function. A general implementation of

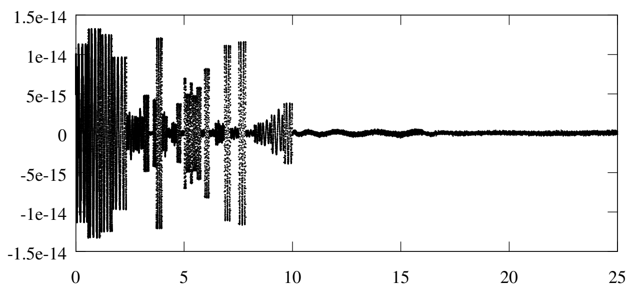

Figure 4.17 – Relative overall error of our implementation for $V(x)$.

these functions, due to Johnson [129], is used for instance in the standard library of the Julia programming language. Here we are interested in implementing the function $V(x)$ for fixed $\lambda$ and $\sigma$, with an overall relative accuracy $\varepsilon = 2^{-45}$. In our experiment, we take $\sigma = 1$ and $\lambda = \frac{1}{2}$, but the method applies verbatim to any moderate rational values.

We start with generating an implementation for $x \in [0; 10]$. With approximations having at most 11 nonzero coefficients, $[0; 10]$ is split into 33 subdomains of width varying from about $0.1$ around $x = 5$ to about $0.6$ near the endpoints.

For large $x$, however, the computation of polynomial approximations starting with the initial values at 0 as described in Section 4.3.3 is no longer feasible. Also, it would not make any sense to split $[10; 2^{1024}]$ into small subintervals. An obvious remedy is to consider $f(\xi) = V(1/\xi)$ on $(0, 0.1)$. A change of variable in (4.17) shows that

$$\sigma^4\xi^6 f''(\xi) + 2\sigma^2\xi^3(\sigma^2\xi^2 - 1)f'(\xi) + (\lambda^2\xi^2 + \sigma^2\xi^2 + 1)f(\xi) = \frac{\lambda\xi^2}{\pi}.$$

This equation has an *ir*regular singular point at $0$, beyond the scope of the polynomial approximation method we discussed. Nevertheless, looking for solutions as formal power series yields a unique divergent series (compare to [105])

$$\xi^2 L(\xi) = \xi^2 + (2\sigma^2 - \lambda^2)\,\xi^4 + (1\sigma^4 - 10\sigma^2\lambda^2 + \lambda^4)\,\xi^6 + \cdots \tag{4.18}$$

whose coefficients again satisfy a simple linear recurrence. It is well-known that such formal solutions at infinity are asymptotic expansions of "true" analytic solutions and provide good approximations for large $x$ [202]. In the present case, the solutions of the homogeneous part of the equation decrease exponentially as $\xi \to 0$, $\xi > 0$, so all solutions of the homogeneous equation share the same asymptotic expansion (4.18).

With our values of $\sigma$ and $\lambda$, we estimated that the polynomial $\widetilde{L}$ obtained by truncating $L(\xi)$ to the order $90$ satisfies $|\widetilde{L}(1/x) - x^2V(x)| < 2^{-65}$ for all $x > 10$. For the purposes of this experiment, this truncation order was found numerically. Rigorous bounds could likely be derived by the method of Olver [202, Chap. 7]. NumGfun does not support evaluations near irregular singular points, but as $\widetilde{L}(\xi)$ stays away from zero on $[0, 0.1]$, we could directly pass it to the Frankenstein backend. The resulting code uses two polynomials of degree 11 with floating-point coefficients.

We then manually combined the two automatically generated codes into implementation that works correctly on the whole of $\mathbb{F}$. The Voigt function is even, so the final code starts by dropping the sign of the argument. Based on a simple comparison, it then calls either the code for $V(x)$ around zero or computes $\xi = 1/x$ (in double precision) and calls the code for $V(1/x)$ around zero.

Figure 4.17 shows a plot of the relative error of the implementation of $V$. The plot covers all subdomains, both around zero and around infinity. It confirms that the accuracy target $\varepsilon = 2^{-45}$ is satisfied. Evaluations take $48$ to $180$ cycles, with most calls in the domain around zero completing in $60$ cycles and most calls around infinity completing in $96$ cycles.

### 4.3.6 Conclusion and Future Work

Our prototype code generator is a step forward towards easier, more adaptive and widespread implementation of special functions. However, it currently comes with several limitations.

First, though we expect it to succeed for all typical examples satisfying the assumptions laid out in Section 4.3.2, code generation might fail on some functions. Making it work for a given function may also require some deal of human intervention.

Second, our code generator handles double precision and double/triple-double expansions, but it does not work for any other FP format, let alone, say, fixed-point arithmetic. It would be interesting to extend it in that direction, perhaps with the help of other existing code generation backends.

Finally, and most importantly, automatic code generation is currently limited to bounded domains. A major perspective for future research is to provide automatic support for evaluation near infinity and singularities of the function, similar to what we outlined in the case of the Voigt profile. This also means extending our framework to handle infinite families of quasi-regularly spaced zeros in the style of [109].

## 4.4 Other Metalibm Use-Cases

In Section 4.2, we have presented the general ideas of our open-ended metalibm-lutetia code generator. In the previous Section 4.3, we have seen how this open-ended code generator can be used to generate code for functions which are not defined algorithmicly but with recurrence relations, making them D-finite. In the next Section 4.4.1, we shall first show that the use of metalibm-lutetia is not reduced to open-ended functions: to this end, we consider the implementation of vectorized `libm` functions, i.e. code evaluation a function on each of the elements of a (small) vector. Then, in Section 4.4.2, we will go one step further and mix up open-ended metalibm-lutetia and domain-specific metalibm-lugdunum even more, by combining them into one piece of code where metalibm-lutetia leverages metalibm-lugdunum for the code generation step required for polynomials.

### 4.4.1 A Metalibm Experiment: A Vectorized `libm`

This Section 4.4.1 on the usage of metalibm-lutetia for the implementation of vectorized functions is based on [157]. Contrary to our expectations, this experiment has led to pretty high interest from researchers and software engineers, working e.g. on FreeBSD. This interest has made us disseminate our code with an Open-Source license at `https://gitlab.com/cquirin/vector-libm`.

### Introduction

Operation Systems provide support for mathematical functions such as $\exp$, $\sin$, $\cos$, $x^y$ through mathematical libraries, commonly called libm. As hardware becomes more and more parallel, in particular through integration of hardware for floating-point [117] SIMD instructions, a need for vector libms arises. These parallelized libms work on vectors, applying the respective mathematical functions in an element-by-element (SIMD-like) manner.

The vector lengths vary: systems may choose to provide only functions for fixed length vectors, typically for the vectors sizes supported by the underlying SIMD hardware, or they may provide APIs for arbitrary vector lengths, cutting the vectors into chunks then mapped onto the SIMD units.

A first advantage of such vectorized libms is to give the user an interface to leverage the full potential of their SIMD hardware. Second, modern compilers such as gcc [6] have support for autovectorization, where scalar loops are transformed into SIMD operations and calls to SIMD-like libm functions. Vectorized libms are hence required as support libraries of modern compilers.

Vectorized libms are no new idea. Several such libms already exist: Intel's SVML [7], AMD's libm [8], GNU glibc/gcc recent addition libmvec [9] or CERN's VDT [211]. However, all these existing libraries have certain shortcomings: some of them are proprietary, which does not allow them to be used on a wider range of systems. Others are written in assembly, which also hinders portability. The only library source code is available for, CERN's VDT, is a mere adaptation of an existing scalar libm, with no optimization of the underlying algorithms to a SIMD environment.

In this Section, we present a new Open-Source vector libm that was developed to fill the gap left by the existing libraries. It was designed with the following goals in mind: first, our library is written in plain C, making it fully portable and easier to maintain and inspect. However, the C source is written in a manner that allows a modern compiler such as gcc to autovectorize the library code to any SIMD hardware unit with reasonable floating-point support. Second, our library uses newly optimized polynomial approximations [26], chosen with parallelization in mind. Finally, the library is completely freestanding, i.e. it does not require a classical scalar libm to handle corner cases of the functions, like libmvec does for example. This further improves code maintainability and prevents certain issues of floating-point non-reproducibility.

This Section is further organized as follows: in Section 4.4.1, we analyze properties of existing vector math libraries. In Section 4.4.1, we give a short overview on the principal techniques used to implement a mathematical function, such as $\exp(x)$, in floating-point arithmetic. In Section 4.4.1, we describe the autovectorization capabilities of modern C compilers and identify certain issues arising with classical math function implementation techniques with respect to autovectorization. In Section 4.4.1, we describe our approach to implementing vectorized math functions, before presenting experimental results in Section 4.4.1. We conclude on vectorized mathematical libraries with an outlook in Section 4.4.1.

6. `https://gcc.gnu.org/`
7. `https://software.intel.com/en-us/node/583201`
8. `http://developer.amd.com/tools-and-sdks/archive/libm/`
9. `https://sourceware.org/glibc/wiki/libmvec`

**Existing Vector Math Libraries**

As already mentioned, several vectorized libms provided by different parties exist: Intel's SVML and VML libraries, AMDs libm, GNU's libmvec and CERN's VDT. None of these existing libraries fully satisfies all users' needs, in particular in the Open-Source world.

First of all, the libraries provided by Intel and AMD are closed-source and particularly optimized for each vendor's hardware. Although libmvec, recently added to the GNU glibc to support autovectorization in gcc, is Open-Source software, it is not fully portable: as a matter of fact, the library is provided as an assembly source for x86-64 systems only. Porting it to other hardware, such as ARM systems is next to impossible. Maintenance and code-inspection (e.g. to derive bounds on maximum approximation error) are hindered by the assembly nature of libmvec.

CERN's VDT library [211] is Open-Source and fully written in high-level languages (C and C++). It is hence portable across systems and it allows for code maintenance and inspection. However, VDT is not a full redevelopment: for its core, i.e. the polynomial and rational approximations of the mathematical functions it implements, it is based on the legacy Cephes library [10]. This means that its developers could not optimize these approximations (in terms of degree etc.) with respect to vectorization. Finally VDT has a C++ interface, which makes it unsuitable for certain C-based projects and hinders its use as a support library for a C compiler.

All existing vector libraries share one common issue: they all depend on a scalar libm to implement certain special cases of certain functions. For example, most vector libraries implement the trigonometric functions such as $\sin(x)$ only in a limited domain around zero. For larger input values, they call respective function in the scalar math library. While this approach is reasonable with respect to performance, as large input values to trigonometric functions are rare in well-written codes, reproducibility issues may arise for input values around the limit when the vector library switches from internal handling to calling the scalar library: the code-base and hence floating-point properties (such as maximum error) for both implementations of the same mathematical functions are not the same. Reproducibility issues, such as monotonicity problems, may hence arise. These issues are even more widespread for non-freestanding vector libms that choose to call the scalar math library for all elements of a vector, if at least one of the vector elements requires special case handling.

**Implementation of Mathematical Functions**

The implementation of a mathematical function, such as $\exp(x), \sin(x)$ or $\operatorname{atan}(x)$, in the floating-point environment provided by IEEE754-2008 has been extensively studied and described in the literature [32, 52, 65, 90, 165, 197, 239]. Classically, a mathematical function on a floating-point argument is computed in four or five steps.

In the first step, the inputs are filtered for special values such as infinities or Not-A-Number data, as well as for numeric input values that lie outside of the function's definition domain or surely provoke overflow or underflow. These special inputs are handled appropriately.

All unhandled numeric floating-point inputs proceed to the second step, called argument reduction step. In this step, algebraic properties of the implemented function, such as $e^{a+b} = e^a \cdot e^b$ or $\sin(x + 2k\pi) = \sin(x)$, $k \in \mathbb{Z}$, are used to reduce the domain the function needs to be actually computed on to some small interval, mostly around zero. In cases when

10. `http://www.netlib.org/cephes/`

the function does not allow for algebraic reduction of its input domain or when the domain obtained after algebraic argument reduction is still too large, the domain is simply split into subdomains. Such subdomain splitting gets reflected in the function's implementation by a sequence of memory reads and conditional branches.

In the third step, called polynomial approximation step, the function's value gets approximated by evaluation of an approximation polynomial. In some cases, a rational approximation may also be used. This part of the code is fully straight-line but may be optimized in terms of degree of the polynomial used and of amount of memory to be read in for the polynomial's coefficients [26].

In a fourth and final step, the function's value on its original argument is recovered by "inverting" the effects of the argument reduction step. This final code sequence commonly is pretty short.

In cases when addressing a slightly larger (about $8$ to $24$ kbytes) amount of memory with indirect addressing is possible, an additional step, executed in parallel with the polynomial approximation step, called table lookup step, may be used [239]. The use of a lookup table with precomputed values allows the reduced argument's domain to be yet smaller, which decreases the degree of the approximation polynomial and hence evaluation latency.

**Leveraging Compiler Support for Autovectorization**

In Section 4.4.1, we have expressed our concern with the use of assembly in different vector libms on the one hand. On the other hand, if we wish to increase portability with the use of a high-level language to implement a vector libm, we need to find a way to express the SIMD parallelism in that high-level language. A good compiler can then perform SIMD instruction selection, hence may avoid us the use of assembly.

With modern C compilers, such as gcc, the SIMD parallelism actually does not need to be expressed explicitly: when enabling so called autovectorization, the compiler is able to map scalar code sequences that get executed repeatedly with in a loop onto SIMD instructions. As a matter of course, both the loop with its length and memory access pattern and the scalar code inside that loop need to satisfy certain conditions in order to allow the compiler to perform the autovectorization.

In our use-case, the implementation of mathematical functions, the loop around the scalar floating-point code is of statically known length (most commonly a small multiple of the underlying hardware's native SIMD vector length). The access pattern to the input and output vectors is strictly linear. Satisfying the compiler's constraints is easy toward this respect.

The constraints on the scalar code inside the loop are more restricting for our purpose. We were unable to find any (formal) specification of the actual constraints required by an autovectorizing C compiler such as gcc but we were able, by experiment, to guess the following constraints:

— The scalar code inside the loop needs to be free of conditional branches. Conditional expression evaluation, using the C operator `cond ? a : b`, is possible in very easy cases: both expressions are integer constants. This constraint typically has the effect that argument reduction by domain splitting cannot be used for the implementation of vector math functions.

— All memory accesses need to be either to a constant address (e.g. to load literal constants) or to consecutive addresses, linearly depending on the loop variable. Hence it is of course possible to read a function's argument in a vector and to write the function's value back to an output vector. The constraint however prohibits the use of lookup tables for argument reduction, as the index to the lookup table does not merely depend on the loop variable but on the floating-point values given in the argument vector.

We are going to describe in the next Section an approach we propose for the implementation of mathematical functions whilst still satisfying these constraints on the code, set out by the capabilities of autovectorizing compilers.

**A Freestanding Vector Math Library**

Our goal is to propose a freestanding vector math library, fully written in C, with code that can be autovectorized with a modern compiler, such as gcc. We strive at providing the most common functions, namely $\exp, \log, \sin, \cos, \tan, \mathrm{asin}, \mathrm{acos}, \mathrm{atan}, \sqrt[3]{}$ in both the IEEE754-2008 binary32 and binary64 format. In this Section, we concentrate on the IEEE754-2008 binary64 format, leaving out the binary32 format. We do this for two reasons: one an autovectorizable algorithm is known for a function in the binary64 format, transcribing it to the binary32 format is a pure matter of software development. Second, as we are going to explain in more detail below, a scalar implementation of each function is required as a fallback in certain cases. While this doubles development work for the binary64 format, performance is not of an issue for that fallback. The binary32 format can hence reuse the same fallback functions, converting hence and forth from binary32 to binary64.

**A Bird's Eyes View on the Proposed Algorithms** The algorithms in our proposed vector libm keep the principal structure of an implementation of a mathematical function: special case handling, argument reduction, polynomial approximation and reconstruction. As set out in more detail below, no conditional branches are possible inside the loop to be vectorized. We therefore cut the loop on the vector into two loops.

First, a loop passes over all vector elements and it determines if the corresponding inputs lie inside the main definition domain of the function. This test is implemented using integer operations that work on the memory representation of IEEE754-2008 binary floating-point numbers. The outputs of each vector element's boolean test result are considered integer numbers that are summed. Autovectorizing compilers are able to translate this construction into a vector reduction operation. If at least one of the elements lies outside the main definition domain of the function, a fallback function is called.

If all vector inputs lie inside the main definition domain of the function, a second loop is started that will compute the mathematical function in a SIMD-parallel manner. Manual analysis of the generated assembly shows that no memory accesses to the input vector are needed for this loop, as the vector elements have already been loaded for the first, domain-check loop. Inside the function computation loop an argument reduction will first be performed. That argument reduction is always based on an algebraic property of the implemented function, at least for the $9$ functions currently implemented. No subdomain splitting is performed, no tables are used. We are going to give more details below.

Polynomial approximation follows argument reduction. Its SIMD autovectorization is trivial to achieve. The degrees of the used polynomials are pretty high, as the use of no tables for argument reduction requires the polynomials to be accurate on larger domains. For certain functions, in particular the inverse trigonometric functions ($\mathrm{asin}, \mathrm{acos}$ and $\mathrm{atan}$)

that have either infinite derivatives at one point or derivatives that tend toward zero, we use more than one approximation polynomial. Their results are then combined in the last, reconstruction step. We have optimized all our polynomials coefficients using the techniques described in [26]. None of our polynomials uses coefficients with higher precision than the working precision, whilst other libraries tend to use floating-point expansions at least for the coefficients of lower degree [32, 165].

Function reconstruction typically requires no branching nor table access and is hence also easy to autovectorize. We always restrict the main path of the function to a domain where the function's value surely does not underflow. This way, we avoid any handling of subnormals on the main path, which is to be autovectorized.

We shall signal a particularity of our implementation of the cubic root function $\sqrt[3]{\cdot}$. We implement this function as $\sqrt[3]{x} = 2^{\log_8(x)}$ with additional handling for the sign of $x$. The particularity is that we use the sequence of argument reduction, polynomial approximation and reconstruction twice: once for $z = \log_8 x$, once for $2^z$.

**Special Case Handling** As mentioned, we call a fallback function to handle special and other particular cases, such as subnormal inputs, whenever at least one element in the given input vector lies outside the main definition domain of the function. This general approach gives convincing results in terms of average performance but requires two parameters to be determined: first, what the main definition domain of a given function is, and, second, which SIMD vector length is appropriate for our approach.

The first parameter is pretty easy to determine: for most functions, it is given by their mathematical definition domain and the properties of the floating-point format –with its exponent width defined by IEEE754-2008. Only for some functions, typically the trigonometric functions $\sin, \cos$ and $\tan$, a choice must be made by design. Essentially, their argument reduction $r = x - k \cdot \pi$ with $k$ the integer nearest to $x/\pi$ becomes increasingly harder for $x$ becoming larger in terms of required working precision. Specialized argument reduction schemes, such as Payne-Hanek argument reduction [207], exist for large input values $x$. Their SIMD parallelization is hard to achieve. In contrast, large input values to trigonometric functions are not quite common in scientific codes. It is hence reasonable to handle only a certain main definition domain around zero in a SIMD vectorized manner and to handle all other (larger) inputs only in the scalar fallback code. We chose to handle inputs $x$ smaller in magnitude than $\left(2^{14} - 1\right) \cdot \pi$ on the main path and to perform Payne-Hanek argument reduction only in the scalar fallback function. This choice is motivated by error analysis but it stays a somewhat arbitrary design decision.

The second parameter to determine with respect to our approach to special case handling is the vector length. Our approach is to call a fallback function as soon as at least one of the input vector elements does not lie in the main evaluation domain. That fallback code is not SIMD-parallelized and hence significantly slower. The average performance of our vector functions is hence determined by two parameters: the probability $p$ to encounter an out-of-main-domain input (among a supposedly large number of inputs) and, of course, the vector length $n$. With $n$ increasing, the probability for the main-domain SIMD-parallelized code to be used becomes lower, typically $(1 - p)^n$. Further, when the fallback function gets called, with increasing vector length $n$, its evaluation time increases. In contrast, when no fallback is necessary, the SIMD-parallelized function typically gets more efficient with increasing vector length, as most compilers are able to interleave several SIMD evaluations, to make them run simultaneously. As the probability $p$ of out-of-domain inputs is typically unknown, it is hard to answer that second question of determining the vector length $n$. We performed some experiments with some assumptions on the distribution of floating-point numbers, in particular equidistribution of the exponents. We could determine that a vector length of $n = 8$ seems to be most efficient on average. Anyway, as our code is written in a high-level language, adapting it to other vector lengths is pretty easy.

**Avoiding the Use of Lookup Tables** The constraint that code to be autovectorized must not use any indirect memory accesses, where the addresses depend on the input vector, implies that we may not use any lookup tables in our implementations. This has several consequences:

Most approaches for argument reduction found in the literature for the implementation of mathematical functions do use tables [65, 239]. We have to revise the argument reduction techniques. For certain functions, this may involve nothing but dropping the table lookup step, increasing the interval the reduced argument may fall in and hence increasing the degree of the approximation polynomial. The functions $\exp(x)$ and $\log(x)$ are examples for this situation.

For other functions, like $\cos(x)$, not having access to lookup tables –combined with the required avoiding of conditional branches– requires overcoming issues of computing and writing evaluation code for polynomials approximating a function that has a zero elsewhere than at zero. However, approaches for this issue do exist [163].

Finally, some other functions, like $\operatorname{asin}(x)$, $\operatorname{acos}(x)$ or $\operatorname{atan}(x)$, have derivatives that become infinite at some point or that tend towards zero for the input argument going to infinity. In these cases, using polynomials for approximation is most inappropriate. Classical argument reduction techniques for these functions with lookup tables try to hide the singularities of the derivatives in the tables. For our work, we had to find other argument reduction schemes. The guiding idea is that an accurate evaluation of the inverses of these tricky functions is easy to obtain with polynomial approximation and evaluation. Argument reduction may hence proceed as follows: first evaluate a low-degree polynomial approximation of the direct function, yielding a pretty inaccurate result. Then compute an accurate approximation of the inverse function at the point found by the inaccurate approximation. Last compute a reduced argument out of the original evaluation point and the result of the inverse function evaluation. Finally accurately evaluate the function of the reduced argument, which will be bounded in magnitude by some upper bound related to the accuracy of the initial inaccurate approximation.

**Example: Implementation of the Logarithm Function** Let us now detail the implementation of the logarithm function as an example for the functions in our vector libm. The function $\log x$ is defined for all inputs $x > 0$. On output, it may not produce any underflow nor overflow [65].

We remove the handling of subnormal input from the main path: the test loop checking whether special case handling is required tests whether all IEEE754-2008 binary64 inputs $x$ are between $2^{-1021}$ and $2^{1023} \cdot \left(2 - 2^{-52}\right)$ (bounds included). Since we implement this test with two integer comparisons on the IEEE754-2008 memory representation, the same test eliminates all zeros, infinities and Not-A-Numbers as well.

All inputs $x$ in that range are then split into an exponent $E$, stored on an integer variable,

| **Function** | **Vector libm** max error in ulps | **System libm** cycles per element | **Vector libm** cycles per element | **Speed-up** |
|---|---|---|---|---|
| exp | 2.6695 | 37.6832 | 10.7050 | 252% |
| log | 1.0825 | 51.9903 | 29.5193 | 76.1% |
| sin | 2.3300 | 236.8824 | 164.1783 | 44.2% |
| cos | 2.4221 | 232.6561 | 198.5054 | 17.2% |
| tan | 3.0352 | 262.3415 | 218.0489 | 20.3% |
| asin | 4.9627 | 21.0550 | 13.8167 | 52.3% |
| acos | 2.4031 | 19.3828 | 15.2912 | 26.7% |
| atan | 2.1013 | 22.0204 | 7.8097 | 181.9% |
| cbrt | 1.1638 | 47.8003 | 27.3462 | 74.7% |

Table 4.2 – Measured maximum error of vector libm and measured performance of system and vector libm

and a significand $m$, stored on an IEEE754-2008 binary64 variable, such that $x = 2^E \cdot m$ and $0.75 \leqslant m < 1.5$. The uncommon range for the significand $m$ between $0.75$ and $1.5$ requires extra work but allows a catastrophic cancellation to be avoided [65]. It is possible to perform that split of the input $x$ into the exponent $E$ and the significand $m$ without using branches, merely with integer operations on the IEEE754-2008 memory representation, even when targeting the uncommon significand range between $0.75$ and $1.5$.

A constant value of $1$ is then subtracted from the significand $m$, yielding $r = m - 1$. This subtraction does not provoke any rounding, as per Sterbenz' lemma. The argument of the logarithm function is hence reduced as follows:

$$\log x = E \, \log 2 + \log\left(1 + r\right),$$

where $r$ varies in the interval $r \in [-0.25; 0.5]$.

On this domain, the function $\log\left(1+r\right)$ can be replaced by a polynomial $p(r)$ of degree $20$ to achieve the required accuracy for IEEE754-2008 binary64.

Reconstruction involves converting the exponent $E$, originally represented on an integer variable, to a IEEE754-2008 variable and multiplying it by the constant $\log 2$. The conversion is performed using the appropriate machine instruction, available in most SIMD instruction sets. It is expressed as a cast in C and it runs in parallel with polynomial evaluation. The constant $\log 2$ is stored with some extra precision as an unevaluated sum of two IEEE754-2008 numbers with some zero trailing bits in their significand to make the multiplication by the exponent $E$ exact.

**Experimental results**

The vector libm we propose was developed targeting a maximum error of the functions of at most $8\mathrm{ulp}$ with respect to the corresponding precision of the function. While this target means that our functions are clearly less accurate than standard scalar libms, which commonly provide correct rounding [66,117,165], it corresponds well with the accuracy requirements of High-Performance Computing e.g. for Physics [211]. In any case, providing correct rounding for vectorized transcendental functions is extremely hard, due to the divergence of the execution paths in the different SIMD slots created by the rounding test [66]. Non-exhaustive measurements show that our library fulfills that accuracy target. Most functions actually have a maximum error below $5\text{ulp}$ (see Table 4.2).

Concerning performance, the speed-up (in terms of cycles per element computed) provided by our library, compiled with gcc 6.3.0, compared with respect to a scalar libm, for instance the GNU glibc 2.24-11 running Debian 4.9.0-3-amd64 on a x86-64 Intel i7-5500U at 2.4GHz, is pretty convincing: for certain functions, such as $\exp(x)$ we obtain a speed-up of up to $252\%$. For certain functions, like $\cos(x)$, which we optimized less, we obtain a speed-up of at least $17.2\%$. See Table 4.2 for details. Additional experiments showed that about half of the speed-up is due to a better utilization of the hardware (SIMD processing, decreasing function call overhead etc.); the other half stemming from optimization of the approximation polynomials.

**Conclusion and Future Work**

We presented a vectorized mathematical library (libm) that is fully implemented in modern C, allowing for good code maintenance and portability. The library is Open-Source, which is not commonly the case for its competitors. It provides reasonable speed-up with respect to a scalar libm, attaining up to $252\%$ of speed-up per vector element. It is fully free-standing as it does not require any scalar libm as a support library. The maximum error of the functions provides stays well below $8\text{ulp}$, corresponding to accuracy requirements of High Performance Computing.

As it stands now, the library is not fully completed with respect to the functions provided by a classical scalar libm. We currently concentrated our efforts on the IEEE754-2008 binary64 format. We still need to derive an implementation of the functions for IEEE754-2008 binary32 format from the algorithms we proposed. Research and design work are also ongoing for certain functions that are hard to implement in a vectorized manner, such as the power function $x^y$, the bi-variate arctangent $\text{atan}\left(\frac{y}{x}\right)$ or the Gaussian error function $\text{erf}(x)$.

### 4.4.2 Uniting with other Code Generators for Enhanced Performance

In the previous Section 4.4.1, we have seen how metalibm-lutetia can be leveraged for the practical implementation of classical, mathematical functions in a vectorized manner. As explained in Section 4.2, code generation for classical functions is more application specific and hence the predilect field of application of metalibm-lugdunum, a "competitor" to our metalibm-lutetia. In this Section 4.4.2, we are going to blur the borders between the open-ended metalibm-lutetia and the domain specific metalibm-lugdunum even more: we will show how we can unite with metalibm-lugdunum –and yet another tool, CGPE– for improved code generation for polynomials, replacing metalibm-lutetia's Horner-scheme based polynomial code generator, which ten-year old legacy code now [165], by a more powerful tool. This Section is based on an article we have recently come up with. We intend to publish it soon.

**Introduction**

Nowadays, scientific computing is more and more important, and achieving high performance requires the development of programs using at best the specificities of modern architectures, especially vector architectures, and manipulating data and operations whose formats have been adapted just right. Concerning the implementation of mathematical functions, a critical part is the polynomial evaluation which is the heart of such programs, whose precision must be controlled to ensure a given output accuracy, and whose efficiency greatly impacts the efficiency of the whole implementation. In this Section, we address the automated development of polynomial evaluation programs, where (1) an evaluation scheme is selected so that its level of parallelism fits at best the capabilities of the underlying architecture, and (2) where operation formats are chosen to guarantee the overall error is no greater than a user-defined error bound, while maintaining high performance.

Many projects have been carried out to assist with the efficient and accurate implementation of mathematical functions. On the first hand, regarding parallelism exploitation, let us cite CGPE [194, 218]. This tool performs automatic generation of parallel codes for evaluating polynomials, sums, and dot-products with sufficient accuracy to ensure that evaluation error is no greater than a given bound. But it does not provide a format adaptation step, and the generation process may fail if the bound to reach is too tight. Another relevant tool is Sardana [121], whose goal is to rewrite expressions, in order to improve accuracy of the output, but without any constraint on the parallelism of the resulting expression. Note, as shown in Section 4.4.2, CGPE's strength is its ability to explore part of the scheme space, returning quickly the more parallel ones. On the other hand, regarding accuracy handling, relevant tools are Sollya [41] and Gappa [188]. Indeed Sollya allows polynomial approximation to be performed with floating-point coefficients and rigorous approximation error bounds, while Gappa uses rewriting rules and interval arithmetic to bound the roundoff errors entailed by the evaluation of a floating-point program.

More generally, tools already exist that have been designed to directly tackle the whole function implementation process. This is the case of Metalibm-lutetia [145, 165], presented in Section 4.4.2. This tool implements various range reduction techniques (symmetry and periodicity detections, domain splitting, ...) and relies on the classical Horner scheme to produce code for evaluating a given function using only binary64 (single or multiple word) floating-point arithmetic [117]. The formats of all the operations have been chosen to ensure a given accuracy, that may be checked *a posteriori* using Gappa. Another paradigm is followed by the Metalibm-lugdunum project [32], an advanced code generator presented in Section 4.4.2. The range reduction is described in a Python-like syntax, and this framework provides back-ends for various (micro-)architectures. It includes automatic code transformation to generate high performance vector codes from scalar description. Note it has already been used in the context of fast mathematical function implementation [33].

Our goal is to extend the Metalibm-lutetia approach, and to provide a framework to implement mathematical functions at a given accuracy, and exploiting at best the underlying architecture parallelism. Therefore the main contributions of the work presented in this Section are:

— The use of schemes exposing more parallelism than the one offered by the Horner scheme, and obtained using CGPE.

— A new and more rigorous way to determine operation formats and to detect cancellations, considering parallel expressions, and allowing to produce code accurate by construction, for which Gappa certificate is not required anymore.

— The use of an "opaque" data type, thus improving the genericity of the approach, and leaving the possibility to extend this work to any other IEEE floating-point formats.

— The integration of this new approach in the Metalibm-lugdunum framework, allowing for high performance implementations.

### Uniting with Other Tools

In this Section, we will shortly review the Metalibm-lutetia approach, and describe other tools, striving at identifying interfaces for possible enhancements.

**Metalibm-lutetia: range reduced polynomial approximation** Metalibm-lutetia is a software tool for automatic code generation for mathematical functions [32]. Its strengths are in the manner the functions to be implemented are represented by the tool: Metalibm-lutetia considers them as black-box objects with a very limited interface, typically simple evaluation over small intervals [32, 145]. It works under the premise that the user does not need to supply any indications on the algorithmic scheme to be used to implement the function. This allows code generation with Metalibm-lutetia for functions defined in an non-algorithmic way, e.g. by a differential equation or as an inverse of another function [163].

The Metalibm-lutetia code generator achieves its goal by an analysis of the given function's properties, geared towards exploitation with floating-point arithmetic, typically by reducing the range of the function's input based on these properties. It is able to detect and produce range reduction code for symmetrical functions (such as $\sinh$), periodic functions (such as $\sin$), functions with additive range reduction (such as $\exp$), functions with multiplicative range reduction (such as $\log$) as well as functions with certain patterns of asymptotic behavior (such as $\sinh$, eventually behaving like $e^x$) [32, 145]. If no other range reduction method can be found with this detection process, it resumes to splitting the input interval into sufficiently small subintervals, allowing for polynomial approximation on each subinterval [147].

For the core function to be implemented on each small range-reduced interval, Metalibm-lutetia is able to perform polynomial approximation, optimized with respect to the use of floating-point numbers as the coefficients of the approximation polynomials. For this step, it leverages the basic building bricks integrated into the Sollya tool, such as an implementation of the Remez algorithm [38, 41, 214] and a variant for approximation floating-point coefficients, commonly called fpminimax [26, 38]. Metalibm-lutetia enhances these basic bricks by a floating-point format optimization, trying to find an approximation polynomial with a minimum number of floating-point values in expansions used for the polynomial's coefficients [165].

The existing Metalibm-lutetia tool, as described in [145], is already able to generate code for the approximation polynomials it generates. For this step, it uses legacy code integrated in the Sollya tool, able to generate code for the Horner evaluation scheme, using only the three formats binary64, double-binary64 and triple-binary64 [164, 165]. This legacy, hard-to-maintain, format-specialized polynomial code generator is the part of Metalibm-lutetia that the work presented in this Section strives at replacing with an optimized, generic code generation algorithm, suitable not only for Metalibm-lutetia but also other tools.

**CGPE: polynomial evaluation scheme generation** In Metalibm-lutetia, for each subinterval, a polynomial is built and an evaluation program is designed where the polynomial is evaluated using Horner scheme. Indeed this scheme is classically used when implementing mathematical functions [57]. The interest of using such evaluation scheme is twofold: (1) this has good numerical property especially when the polynomial is evaluated not too close to a zero of the polynomial [19], and (2) it minimizes the number of additions and multiplications [138, p. 519]. However, Horner scheme is fully sequential, and for achieving high performance, we prefer schemes exposing more parallelism, like Estrin scheme.

In this sense, CGPE has been designed to explore the space of evaluation schemes, and especially those for evaluating polynomials. Particularly, CGPE (standing for Code Generation for Polynomial Evaluation) is a software tool to generate fast and certified programs for evaluating polynomial [194, 218]. Initially designed to produce codes in fixed-point arithmetic, it has been widely used in the development of the FLIP library, and is now capable of handling floating-point arithmetic.

Internally, CGPE behaves like a compiler with an architecture including (1) a front-end, computing DAG representing polynomial evaluation schemes, (2) a middle-end, performing analyses (like fixed-point format determination) and error bound computations, and (3) a back-end in charge of producing codes in different languages or architecture description. The work presented here relies on its front-end only. Indeed its middle-end does not embrace precision adaptation process, which is the heart of our work, since it simply checks if a given target evaluation error is satisfied. Moreover its back-end does not provide capabilities for targeting vector architectures, and we prefer leaving this step to Metalibm-lugdunum presented in Section 4.4.2 which provides such capabilities to take benefit from different modern vector architectures.

The main issue of using CGPE comes from the explosion of the search space of evaluation schemes [194]. In order to reduce these combinatorics, CGPE uses heuristics for degree greater than $6$. However these heuristics might be inefficient when the polynomial degree grows, and we switch to classical schemes like Estrin as soon as polynomial degree gets greater than $12$.

**MetaLibm-lugdunum: back-end and code generation** Metalibm-lugdunum (MLLug) is a code generator dedicated to the generation of mathematical function. It consists of:

— a domain specific language: the Metalibm Description Language (MDL),

— a set of middle-end optimization passes,

— a set of back-ends able to emit machine-specific code.

Based on Python, the MDL is expressive enough to allow for the precise description of the implementation of an elementary function. This description is processed by a configurable pipeline of optimization passes which is architecture agnostic. Those passes translate the MDL description into a reduced form which is then processed by one of the available back-ends (which include different flavors of x86, Kalray's K1). This back-end translates the MDL intermediate representation (IR) into a compilable source code while selecting architecture specific implementations (e.g. intrinsics). The versatility of this IR permits static code transformation such as vectorization or inline (if-conversion, . . . ) very easily.

The strength of MLLug resides in its expressive domain specific language, its parametrizable optimization pipeline, and the fact that it splits middle-end consideration from target specific code generation, allowing for several range of independent optimization and implementation selections.

### Generic implementpoly: theoretical aspects

This Section presents the theoretical aspects of our work, and particularly the way used to compute target error bounds and to handle cancellations.

**Problem statement** Let us now precisely state the problem we consider in this Section: given a univariate polynomial $p \in \mathbb{R}[x]$ with real or floating-point coefficients, an interval $I = [a, b]$ with finite bounds $a, b \in \mathbb{R}$, as well as a so-called positive target error $\bar{\varepsilon} \in \mathbb{R}^+$, our objectives are:

— We want to generate floating-point code for a function $P$, such that for every floating-point input $x \in I$, there exists $\varepsilon \in \mathbb{R}$ such that

$$P(x) = p(x)\ (1 + \varepsilon)\,, \text{ with } |\varepsilon| \leqslant \bar{\varepsilon}. \tag{4.19}$$

In other words, we want the relative error of the generated code to be bounded by $\bar{\varepsilon}$.

— We want the generated code be optimized in terms of performance, for a given computer architecture and the floating-point formats sensibly usable on that architecture, leveraging both the power of CGPE and Metalibm-lugdunum. Typically, we want the CGPE-provided polynomial evaluation scheme to exhibit sufficient opportunities for parallelism to possibly fully exploit the instruction-level parallelism of the considered architecture. A model of the architecture and the usable floating-point formats becomes an additional input of our code generation algorithm, typically captured as a code generation context. We will model the optimization of the generated code typically by an optimization of the number of operations possibly able to run in parallel for different polynomial evaluation schemes.

— We want the code generator to be rigorous in the sense that when it reports a code generation success, the code must rigorously and provably satisfy the constraint given by Eq. (4.19). In order to allow this property to be proven formally, we want our generator to produce, along with the code, a Gappa proof script [188] able to exhibit the fact that Eq. (4.19) is satisfied by the generated code.

— In cases when the code generator needs to round certain coefficients of the polynomial $p$ to allow for actual implementation on a certain floating-point architecture, for instance when the coefficients of $p$ are real numbers that are not representable as floating-point numbers, we want the code generator to report the actual polynomial $\widetilde{p}$ implemented. This allows higher-level code generators like Metalibm-lutetia to take the difference between $p$ and the actual polynomial $\widetilde{p}$ into account.

**Target error bound computation** Once an evaluation scheme is fixed –on output of CGPE– for a certain polynomial $p$, we need to find a way to distribute the error "budget" which the target error $\bar{\varepsilon}$ gives us for the rounding errors in its evaluation, over the different operations. Technically, in this phase, the polynomial $p$, together with its evaluation scheme, is represented as a tree; our task is to distribute the target error $\bar{\varepsilon}$ over the operations at the nodes –and leaves, as we shall see– of that tree.

Under a different angle, the various elementary rounding errors due to the operations in the tree combine to form an overall error. This combination is of course not linear; certain sub-errors may get amplified or attenuated. When distributing the overall target error in a recursive manner over the tree, our task is to predict the different amplification factors in such a way that those operations' elementary errors which get amplified most get most out of the target error budget, or, similarly, that all elementary errors, after amplification, appear for the same amount in the overall, combined error term. This error term needs to enclose also the second and high-order error terms in order to allow for a fully rigorous approach.

A first step in distributing the target error $\bar{\varepsilon} \in \mathbb{R}^+$ over $k$ elementary errors $\bar{\varepsilon}_i \in \mathbb{R}^+$, each of which gets amplified resp. attenuated by a factor $\bar{\alpha}_i \in \mathbb{R}^+$ in such a way that

$$\sum_{i=1}^{k} \bar{\alpha}_i \, \bar{\varepsilon}_i \leqslant \bar{\varepsilon}$$

is to determine the number $k$ of elementary rounding errors. In order to do so, we need to identify the possible sources of rounding error in a polynomial's evaluation: as matter of course, the addition and multiplication operations form sources of error. As we consider multi-word arithmetic with floating-point expansions [164, 165], the elementary rounding error bound for each addition and multiplication is not always the same; it is rather given by the format of the operation's operands and the format of the output of that operation. We shall detail this point below. But once we look at multi-word arithmetic, we must also consider the point $x$ the polynomial $p$ is evaluated at as a multi-word floating-point datum. Similarly, we must consider the coefficients of the polynomial to be stored on such multi-word floating-point data. Doing so, we discover two additional sources of error: whenever they occur in the polynomial evaluation scheme's associated tree, the point $x$ and the coefficients might be considered to be exact but they may also be truncated to a shorter multi-word datum in order to allow for more performance in evaluation. This truncation yields to an error; we shall hence also take into account the elementary rounding error due to the operations accessing the variable $x$ and the coefficients of the polynomial. Finally, considering a polynomial with its scheme as a tree is a slight oversimplification: the schemes exhibiting the best chances for parallel evaluation often work on sub-polynomials expressed in terms of powers of the variable $x$, such as $x^2$ or $x^4$. These powers $x^t$ of the variable $x$ may appear several times in the polynomial's evaluation "tree", which is technically speaking a directed acyclic graph (DAG). For target error distribution purposes with multi-word arithmetic, we must hence consider a source of error for these powers of $x$, too.

Thus, we are given a tree with leaves formed by constants for coefficients, accesses to the variable $x$, and accesses to powers $x^t$ of the variable $x$, while inner nodes are formed by additions and multiplications. We strive at distributing the target error budget evenly over the amplified elementary errors. We therefore perform a first traversal of the tree and allocate what we call *rounding error units* to each of the nodes of the tree: the leaves are annotated with

— $k = 1$ rounding error unit for the coefficients,

— $k = 1$ rounding error unit for the variable $x$,

— $k = \lceil \log_2 t \rceil$ rounding error units for the power $x^t$, $t \geqslant 2$.

The inner nodes are annotated with $k = k_l + 1 + k_r$ rounding error units for an addition or a multiplication of a left and a right sub-tree that get annotated with $k_l$ resp. $k_r$ rounding error units.

In a second step, we can then distribute the target error $\bar{\varepsilon}$ over the different amplified elementary errors in the tree, guided by the already available rounding error units, which indicate the fraction the amplified elementary error should take out of the budget. A second tree traversal is performed; for the leaf nodes, the recursively propagated target error $\bar{\varepsilon}$ is taken as is:

— a coefficient is rounded with the elementary error bound $\bar{\varepsilon}$,

— an instance of an access to the variable $x$ is performed with an elementary error bound $\bar{\varepsilon}$ and, still similarly,

— an access to a power $x^t$ of the variable $x$ is performed with the full target error $\bar{\varepsilon}$.

For the inner nodes, on which an addition or a multiplication is considered, some more work is needed: suppose we work with relative error bounds for the evaluation of a polynomial $p$ at $x$.

— If an inner node is of multiplication type and $k = k_Q + 1 + k_R$ rounding error units have been allocated at this step, we have to analyze the error of evaluating a polynomial $p(x) = q(x) \times r(x)$ by first evaluating $q(x)$ to a floating-point approximation $Q(x) = q(x)\ (1 + \varepsilon_Q(x))$ and $r(x)$ to a floating-point approximation $R(x) = r(x)\ (1 + \varepsilon_R(x))$ and then multiplying $Q(x)$ and $R(x)$ with a rounding error to obtain $P(x)$ as an approximation of $p(x)$: $P(x) = Q(x) \otimes R(x) = Q(x) \times R(x)\ (1 + \varepsilon_M(x))$. Again, we know that $k_Q$ rounding error units have been allocated to the evaluation of $q$ and $k_R$ for $r$. We get

$$\begin{aligned} P(x) &= Q(x) \times R(x)\ (1 + \varepsilon_M(x)) \\ &= q(x) \times r(x) \\ &\qquad (1 + \varepsilon_Q(x))\,(1 + \varepsilon_R(x))\,(1 + \varepsilon_M(x)) \\ &= p(x)\ (1 \\ &\qquad + (1 + \varepsilon_R(x))\,(1 + \varepsilon_M(x))\ \varepsilon_Q(x) \\ &\qquad + (1 + \varepsilon_M(x))\ \varepsilon_R(x) \\ &\qquad + \varepsilon_M(x)) \\ &\triangleq p(x)\ (1 + \varepsilon_P(x))\,. \end{aligned}$$

By allocating, out of the target error $\bar{\varepsilon}$ for this step, $\bar{\varepsilon}_M \triangleq \frac{1}{k}\,\bar{\varepsilon}$ as a target error for the multiplication operation and

$$\begin{aligned} \bar{\varepsilon}_R &\triangleq \frac{k_R}{k}\,\frac{1}{1 + \bar{\varepsilon}_M}\,\bar{\varepsilon} \\ \text{and}\quad \bar{\varepsilon}_Q &\triangleq \frac{k_Q}{k}\,\frac{1}{(1 + \bar{\varepsilon}_R)\,(1 + \bar{\varepsilon}_M)}\,\bar{\varepsilon} \end{aligned}$$

for the two recursive calls for the target error distribution on the trees for $q$ and $r$, we ensure that $|\varepsilon_Q(x)| \leqslant \bar{\varepsilon}_Q$ and $|\varepsilon_R(x)| \leqslant \bar{\varepsilon}_R$ for all $x$ in the considered domain $I$. We therefore obtain the desired bound for $\varepsilon_P(x)$:

$$\begin{array}{rcl}
|\varepsilon_P(x)| & = & |(1+\varepsilon_R(x))\,(1+\varepsilon_M(x))\;\varepsilon_Q(x) \\
& & \quad +\,(1+\varepsilon_M(x))\;\varepsilon_R(x) \\
& & \quad +\varepsilon_M(x)| \\
& \leqslant & \dfrac{k_Q}{k_Q+1+k_R}\,\dfrac{|(1+\varepsilon_R(x))\,(1+\varepsilon_M(x))|}{(1+\bar{\varepsilon}_R)\,(1+\bar{\varepsilon}_M)}\,\bar{\varepsilon} \\
& & +\dfrac{k_R}{k_Q+1+k_R}\,\dfrac{|1+\varepsilon_M(x)|}{1+\bar{\varepsilon}_M}\,\bar{\varepsilon} \\
& & +\dfrac{1}{k_Q+1+k_R}\,\bar{\varepsilon} \\
& \leqslant & \bar{\varepsilon}.
\end{array}$$

— If the inner-node is of addition type, i.e. when $p(x) = q(x) + r(x)$, the target error distribution process is quite similar but the error on $Q(x) = q(x)\;(1+\varepsilon_Q(x))$ and $R(x) = r(x)\;(1+\varepsilon_R(x))$ get amplified each by a factor, which we need to bound. With $P(x) = Q(x) \oplus R(x) = (Q(x)+R(x))\;(1+\varepsilon_A(x)) \triangleq p(x)\;(1+\varepsilon(x))$, we obtain

$$\begin{array}{rcl}
\varepsilon(x) & = & \varepsilon_A(x) \\
& & \quad +\,(1+\varepsilon_A(x))\;\dfrac{q(x)}{p(x)}\,\varepsilon_Q(x) \\
& & \quad +\,(1+\varepsilon_A(x))\;\dfrac{r(x)}{p(x)}\,\varepsilon_R(x) \\
& \triangleq & \varepsilon_A(x)\, + \\
& & \quad (1+\varepsilon_A(x))\;\alpha_Q(x)\,\varepsilon_Q(x) \\
& & \quad +\,(1+\varepsilon_A(x))\;\alpha_R(x)\,\varepsilon_R(x).
\end{array}$$

Let us assume we can compute finite bounds $\bar{\alpha}_Q \in \mathbb{R}^+$ and $\bar{\alpha}_R \in \mathbb{R}^+$ on the maximally possible amplification factors $\alpha_Q(x)$ and $\alpha_R(x)$, i.e. bounds such that

$$\|\alpha_Q\|_\infty^I \leqslant \bar{\alpha}_Q \quad \text{and} \quad \|\alpha_R\|_\infty^I \leqslant \bar{\alpha}_R.$$

It is then easy to show that the following distribution of the target error $\bar{\varepsilon}$ over the addition and the two sub-polynomials yields to a valid bound for $|\varepsilon(x)| \leqslant \bar{\varepsilon}$ for all $x$ in the considered domain $I$:

$$\begin{array}{rrcl}
& \bar{\varepsilon}_A & \triangleq & \dfrac{1}{k}\,\bar{\varepsilon}, \\
& \bar{\varepsilon}_Q & \triangleq & \dfrac{k_Q}{k}\,\dfrac{1}{1+\bar{\varepsilon}_A}\,\dfrac{1}{\bar{\alpha}_Q}\,\bar{\varepsilon}, \\
\text{and} & \bar{\varepsilon}_R & \triangleq & \dfrac{k_R}{k}\,\dfrac{1}{1+\bar{\varepsilon}_A}\,\dfrac{1}{\bar{\alpha}_R}\,\bar{\varepsilon}.
\end{array}$$

Computing finite and tight but rigorous bounds for $\alpha_Q(x) = \frac{q(x)}{q(x)+r(x)}$ and $\alpha_R(x) = \frac{r(x)}{q(x)+r(x)}$ is possible and pretty easy when $p(x) = q(x) + r(x)$ has no zero in the domain $I$ other than where $q(x)$ or $r(x)$ have a zero. In other words, finite bounds exist and are readily computed when no catastrophic cancellation can occur on the floating-point addition between $Q(x)$ and $R(x)$.

Technically, rigorous bounds on the rational functions $\alpha_Q(x)$ and $\alpha_R(x)$ can be obtained using Sollya's facilities to manipulate and analyze polynomials: similarly to what has been used in [40,253], we bound a rational function $f/g$, $f, g \in \mathbb{R}[x]$, over an interval $I = [a, b]$ by first reducing the rational function to its least terms, leveraging Sollya's support to compute $\gcd(f, g)$, then exclude possible poles why determining that $g$ has no zero over $I$, using implementation of Sturm's theorem [40,41], tightly enclose all the zeros of $f'g - fg'$ with thin intervals using a zero search and again Sollya's implementation of Sturm's theorem and finally evaluate $f/g$ over all these intervals as well as at $a$ and $b$.

**Cancellation detection and absolute error bounds** In the case when a polynomial $p$ is of the form $p(x) = q(x) + r(x)$ and it has a zero at some point $x_0$ where $q(x)$ and $r(x)$ have no zero, the amplification factors $\alpha_Q(x) = q(x)/p(x)$ and $\alpha_R(x) = r(x)/p(x)$ have no finite bound, due to the pole at the zero of the polynomial $p$. This situation describes the case of a catastrophic cancellation on the addition of $q(x)$ and $r(x)$. Since $q(x)$ and $r(x)$ cannot generally be computed exactly, this infinite error amplification means that no accuracy would be left after the cancellation. We hence need to detect these cases and address them separately.

Detection of the case is actually pretty simple: when trying to bound $\alpha_Q$ and $\alpha_R$, we anyhow check that their denominator polynomial $q$ has no zero in the considered interval $I$. We can then stop the recursive relative target error distribution process described in the previous Section 4.4.2.

Sometimes an easy way to address cases of cancellation is to just consider another polynomial evaluation scheme which might not offer as much possible parallelism but, in contrast, might be free of cancellation cases. We implemented this fallback as one of the solutions our code generator considers.

However, for certain polynomials, even the Horner scheme, fully sequential and offering no opportunity for parallelism, contains cancellation cases. In order to allow at least one evaluation scheme to be used in implementation, we need to find other ways to support cancellation.

Even if no general solution exists, sometimes it helps to consider absolute error bounds for the polynomial and its subpolynomials. Suppose we can discover, sufficiently close to the root of the polynomial's evaluation scheme tree, a (sub-)polynomial $s$ that has no zero in the considered interval $I$. Suppose the recursive relative target error distribution algorithm presented in the previous Section requires $s$ to be implemented with a relative target error $\bar{\varepsilon}_S > 0$. As $s$ has no zero in the interval $I$, we can compute a non-zero lower bound $\underline{s}$ on $|s(x)|$, valid for all $x \in I$:

$$0 < \underline{s} < |s(x)|\,, \forall x \in I.$$

If we now find a way to implement $s$ with an absolute target error bound $\bar{\delta} \triangleq \bar{\varepsilon}_S\, \underline{s} > 0$, we obtain

$$S(x) = s(x) + \delta(x)$$

where $|\delta(x)| \leqslant \bar{\delta} = \bar{\varepsilon}_s\, \underline{s}$, $\forall x \in I$, and hence

$$S(x) = s(x)\, \left(1 + \frac{\delta(x)}{s(x)}\right)$$

where $\frac{\delta(x)}{s(x)}$ is upper bounded in magnitude (for all $x \in I$) by

$$\left|\frac{\delta(x)}{s(x)}\right| \leqslant \frac{|\delta(x)|}{\underline{s}} \leqslant \frac{\bar{\varepsilon}_S\,\underline{s}}{\underline{s}} = \bar{\varepsilon}_s$$

as desired. Computing a lower bound $\underline{s}$ is pretty easy, using again Sollya's numerical zero finders on $s'$ and ensuring that all roots of $s'$ have be enclosed using Sturm's theorem.

This way we reduce the implementation of $s$ with a relative target error of $\bar{\varepsilon}$ to an implementation of $s$ with an absolute target error of $\bar{\delta}$. With an absolute target error, catastrophic cancellation is no longer an issue: we can modify the recursive target error distribution procedure as follows:

— For an addition $p(x) = q(x) + r(x)$, an absolute target error $\bar{\delta}$ can be distributed in the following way:

$$\bar{\delta}_A \triangleq \frac{1}{k}\,\bar{\delta}, \quad \bar{\delta}_Q \triangleq \frac{k_Q}{k}\,\bar{\delta}, \quad \text{and} \quad \bar{\delta}_R \triangleq \frac{k_R}{k}\,\bar{\delta}.$$

Inhere, $k_Q$ and $k_R$ are the rounding error unit counts for the subpolynomials $q$ and $r$, $k = k_Q + k_R + 1$ the rounding error count for $p$, $\bar{\delta}_A$ an absolute error bound for the floating-point addition of $Q(x)$ and $R(x)$ and $\bar{\delta}_Q$ and $\bar{\delta}_R$ the absolute target errors for $q$ and $r$, recursively passed on to the subpolynomials.

— For a multiplication $p(x) = q(x) \times r(x)$, distributing an absolute target error $\bar{\delta}$ is slightly more complicated: we need to first compute tight and finite upper bounds $\bar{\alpha}_Q \geqslant 0$ and $\bar{\alpha}_R \geqslant 0$ such that

$$\|q\|_\infty^I \leqslant \bar{\alpha}_Q \quad \text{and} \quad \|r\|_\infty^I \leqslant \bar{\alpha}_R.$$

In contrast to the amplification factors considered for relative target error propagation, finite bounds $\bar{\alpha}_Q$ and $\bar{\alpha}_R$ always exist in this absolute error multiplication case, $q$ and $r$ being polynomials considered on an interval $I$ with finite bounds.

Knowing these bounds $\bar{\alpha}_Q$ and $\bar{\alpha}_R$ we can then distribute the absolute target error $\bar{\delta}$ as follows:

$$\bar{\delta}_M \triangleq \frac{1}{k}\,\bar{\delta},\ \bar{\delta}_Q \triangleq \frac{k_Q}{k}\,\frac{1}{\bar{\alpha}_R}\,\bar{\delta}\ \text{and}\ \bar{\delta}_R \triangleq \frac{k_R}{k}\,\frac{1}{\bar{\alpha}_Q + \bar{\delta}_Q}\,\bar{\delta}.$$

Inhere, $k_Q$ and $k_R$ are the rounding error unit counts for the subpolynomials $q$ and $r$, $k = k_Q + k_R + 1$ the rounding error count for $p$, $\bar{\delta}_M$ an absolute error bound for the floating-point multiplication of $Q(x)$ and $R(x)$ and $\bar{\delta}_Q$ and $\bar{\delta}_R$ the absolute target errors for $q$ and $r$, recursively passed on to the subpolynomials.

As a matter of course, performing target error distribution for an absolute error bound comes at the cost that some of the accuracy bounds get estimated in a pessimistic way. Further, it is only possible if we actually find a (sub-)polynomial $s$ that has no zero in $I$. Due to this observation, we implemented a mix of relative and absolute target error distribution, using absolute error bounds only if cancellation cases are detected for the relative error bounds.

**Tightness of the obtained bounds** Up to this point, we have only sought to find ways to distribute a target error on the different operations in a polynomial's evaluation scheme tree under the premise of correctness: the generated code's overall error must rigorously be less than the target error bound. However, an additional, important question for maximum code performance is the tightness of the bound, as higher accuracy always affects floating-point performance.

A general observation is that the previous analysis is in certain cases quite tight, in others quite pessimistic and hence not tight. This lack of tightness stems from several factors:

— We allocated parts of the error "budget" of the target error for the rounded access to the polynomial's free variable $x$ and for the rounding of the polynomial's coefficients. For actual code generation examples, it can be observed that both the variable $x$ and the coefficients are already stored on the smallest floating-point format that is considered and hence no rounding will ever actually occur on them.

— Our target error distribution algorithm determines real-valued, positive bounds for the relative or absolute error of each operation in the tree. These bounds cannot generally be achieved tightly with a given discrete set of floating-point formats and corresponding operations. Given an error bound, we will hence choose a floating-point operation that is actually often quite a bit more accurate than the bound.

Seen from another viewpoint, the actual, final error not tightly achieving the initial target error bound can have positive effects: if a sufficient number of operations is over-accurate, other operations might actually use up the additional headroom and use less accurate implementations with better performance.

It is pretty easy to modify the target error distribution algorithm presented above, so that it also computes and reports a bound on the actual error due to the floating-point operations eventually chosen out of the discrete set of possible operations. If the final reported bound on the actual error is deemed too small with respect to the initial target error, the target error distribution can be rerun with a higher target error, hoping for the actual error bound being still lower than the original target error bound.

An elementary step in the modified target error distribution step hence reads as follows: For each node, execute the following steps:

1. Distribute the target error $\bar{\varepsilon}$ onto the operation's target error $\bar{\varepsilon}_O$ and possibly target errors $\bar{\varepsilon}_Q$ and $\bar{\varepsilon}_R$ for the sub-polynomials $q$ and $r$.
2. On an inner addition or multiplication node, perform recursive calls for $q$ and $r$.
3. Based on $\bar{\varepsilon}_O$ determine the actual formats used for the floating-point operation for this step.
4. Compute an actual bound for the chosen operation.
5. Using the actual bounds reported by the possibly performed recursive steps and the actual bounds for the floating-point operation, compute and report an actual error bound.

### Steps toward code generation

Figure 4.4.2 shows the various stages of the inner part of the generation process. It is embedded into Metalibm-lutetia which iterates over it to generate a complete function approximation. This Section deals with the work done at an implementation-level to prepare trees for code and Gappa certificate generation.

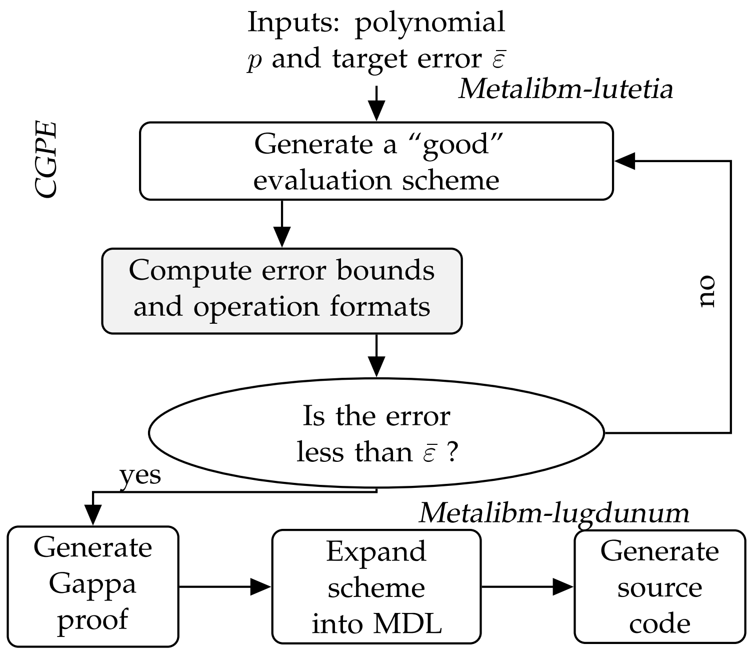
Inputs: polynomial $p$ and target error $\bar{\varepsilon}$
Metalibm-lutetia
CGPE
Generate a "good" evaluation scheme
Compute error bounds and operation formats
no
Is the error less than $\bar{\varepsilon}$ ?
yes
Metalibm-lugdunum
Generate Gappa proof
Expand scheme into MDL
Generate source code

Figure 4.18 – Adaptive implementation process scheme.

**Genericity through an "opaque" data type** The modified target error distribution and actual error bound determination algorithm presented in Section 4.4.2 require a way to determine, for a given operation target error bound, the floating-point format to be used on that operation, as well as a way to compute, for the chosen format and operation type, an actual error bound. These two selection and error bound methods provided, the algorithm is however generic with respect to the actual floating-point formats used in the final code generation phase. These formats might be IEEE binary32 or binary64, or floating-point expansions of such numbers. (In our case, we consider only binary64 based arithmetic).

In our implementation of the polynomial code generation algorithm, we took advantage of this type of genericity by using an "opaque" data type on the interface between the target error distribution algorithm and the back-end that chooses the actual floating-point operations for a given target error and reports the actual bounds for these operations. For the target error distribution algorithm, the formats to be used in a given code generation instance are captured in a code generation context object that contains methods for format selection and error bounding.

From the standpoint of the implementer of these methods, the opaque data type has the advantage that it allows other, non-IEEE intrinsic parameters to be captured, for example overlap bounds for multi-word floating-point expansions, as discussed in Section 4.4.2.

**Multi-word arithmetic overlap handling** We implemented an instantiation of the "opaque" data type which bridges the new method described in Section 4.4.2 with Metalibm-lugdunum meta-blocks.

One of the challenges of large multi-word numbers is to manage overlap and renormalization. Indeed while every double-double operation we use produces results with no overlap between limbs, this format is not sufficient when targeting higher accuracies, and triple-double format becomes mandatory. Basic block for triple-double are really expensive, in term of computation, compared to their double-double counterparts. This cost can be partially limited by using non-normalizing operations. Those operations inspired by the algorithms published in [164] only enforce normalization between the middle and low part of their triple-double results, they let overlap between high and middle part grows with each operation. This overlap, although required for performance reason, must be kept contained.

There are two basic families of methods to manage overlaps. The first one called *early renormalization* insert a renormalization as soon as the accuracy of a triple-word output appears to be less than the one which could have been obtained with a non-overlapping double-word results. This method was used in the legacy implementpoly [165]. The second method called *lazy just-in-time renormalization* waits until renormalization is required. Only at the point where there is no known algorithm to implement an operation, we try to renormalize one or both of the inputs until we find an applicable algorithm for the operation at hand. We tolerate increasing overlap until its grows beyond what was acceptable.

**Work on Metalibm-lugdunum basic blocks** Metalibm-lugdunum has been extended with multi-word basic blocks. Those basic blocks, based on algorithms from [164], describe addition, multiplication and normalization primitives from various combination of single-word and multi-word inputs and outputs. MLLug allows us to go beyond the source description: MLLug's *meta-blocks* are a generic implementation of basic algorithms for multi-word arithmetic. They can easily be instantiated for various support precisions. A support precision is the basic binary32 or binary64 format used to store a word in a multi-word value.

The meta-blocks manipulate format descriptors that record various information (value format, range, relative error, error target). They are used to choose which meta-block can be applied and to compute the output descriptors which will be used as input descriptors for the next stage of meta-blocks.

Each MLLug's block consists of: an application predicate, an expansion function, and an output format generator. The application predicate encodes the required conditions for a meta-block to be valid (e.g. order between operands, overlap conditions, ...). Only if this predicate in valid on a set of inputs descriptors the corresponding meta-block can be applied to those inputs. The expansion function generates the MDL sequence to implement the meta-block. This function is parameterized by the support word format (e.g. binary64). Finally, the output format generator is a function which generates the output format descriptor based on the input format descriptors.

**Power tree handling** As explained in Section 4.4.2, in this work we manipulate a tree, where leaves may be formed by accesses to power $x^t$ of the variable $x$. These particular nodes may appear at different place in the tree, and for performance purpose, they must be handle separately. This is done in two steps.

First when deciding the polynomial evaluation tree, we compute, at the same time and using CGPE, a set of trees, one for each $x^t$ leaf appearing in the whole evaluation. These small trees are formed by accesses to the variable $x$ and multiplication nodes, and are chosen so as the number of multiplications used for evaluating all the $x^t$ is the smallest.

Second, once the target bound computation and format determination are performed, for each power $x^t$ node appearing in the polynomial evaluation tree, we obtain a couple (target error, format). The remaining part consists in distributing the error "budget" in these small power trees, and in deciding if this format can be used while ensuring an actual error less than this target error. For doing this, for each value $t$ and each format we need for this $x^t$, we consider only the $x^t$ power node requiring the smallest target error, the other ones can be implemented in the same way while satisfying accuracy constraint. And finally we launch the target bound computation on the tree computing $x^t$ with this particular smallest target error, and we determine the corresponding output format, which might come out different from the format previously chosen. This whole process results in eventually several "budget" distributions for a given $x^t$ power, and thus different computation trees with different output accuracies and formats. At generation time, the most appropriate trees will finally be chosen among the ones computed according to target error bounds and output formats.

**Generation of a Gappa correctness proof** The pieces of information, such as the error bounds of the operations, manipulated by the modified target error distribution algorithm presented above, are easily captured as annotations on the polynomial's evaluation scheme tree. Using these annotations, it is pretty straightforward to generate a Gappa script. This script, when fed into the Gappa proof tool, allows the error bound computations to be checked and formally verified, when needed. Metalibm-lutetia for example does not rely on the polynomial code generation algorithm in a way that it needs it to report a rigorous result. It rather uses the error bound computed and proven by Gappa.

One challenge in the Gappa proof script generation is to find the appropriate amount of Gappa proof hints. These hints are used by Gappa as an indication on which is the best way to express the overall roundoff error of the implemented polynomial with respect to the exact polynomial in terms of the elementary error bounds for each operation. On the one hand, a Gappa script with too few proof hints may result in Gappa's failure to prove the overall error bound while the bound is still valid. On the other hand, Gappa may take too much time to process a proof script with too many proof hints, making the validation infeasible in practice. Our heuristic to solve this issue is to emit all possible hints we are aware of given the annotations in the polynomial evaluation scheme tree but to make Gappa use them only if a first Gappa run without hints was unable to prove the implemented polynomial's overall error bound.

**Numerical experiments**

We have implemented this new approach in a Python environment, using PythonSollya and PythonCGPE bindings, and plugged it with the range reduction process of Metalibm-lutetia. Numerical experiments were carried out to compare the new method against Sollya's legacy implementpoly function [41, 165] in terms of performance. Those experiments were made on four problem definitions with target errors varying from $2^{-10}$ up to $2^{-85}$:

— **exponential**: approximation of $\exp(x)$ on $[-\frac{1}{2}, \frac{1}{2}]$,

— **sine**: approximation $\sin(x)$ on $[-\frac{\pi}{4}, \frac{\pi}{4}]$,

— **hyperbolic sine**: approximation of $\sinh(x)$ on $[-1, 1]$,

— **erf**: approximation of $\mathrm{erf}(x) - \frac{1}{2}$ on $[\frac{1}{4}; \frac{1}{4}]$.

The **exponential**, **sine** and **hyperbolic sine** problems only use polynomial approximations after argument reduction, while the **erf** problem authorizes a table with up to 8-bit indices.

Table 4.3 – Speedup of new vs. reference implementpoly.

| Target accuracy: $-\log_2 \bar{\varepsilon}$ (bits) | **sine** | **erf** | **exponential** | **hyperbolic sine** |
|---|---|---|---|---|
| 10 | 0.70 | 0.73 | 1.07 | 1.08 |
| 20 | 0.86 | 1.07 | 1.32 | 1.13 |
| 30 | 0.81 | 1.09 | 1.46 | 1.07 |
| 40 | 1.24 | 1.24 | 1.69 | 1.26 |
| 45 | 1.51 | 1.34 | 1.53 | 1.00 |
| 50 | 0.81 | 0.87 | 1.69 | 0.33 |
| 55 | ? | ? | 1.42 | 0.98 |
| 65 | ? | ? | 3.13 | 1.53 |
| 75 | ? | ? | 2.54 | 1.82 |
| 85 | ? | ? | 2.73 | 3.48 |

The latter is interesting because the function has zeros on the interval, but those zeros are not floating-point values. Thus it is possible to implement it with a bounded relative error.

For each experiments, the output code was compiled using GCC 6.3.0 under GNU/Linux environment on an Intel® core i5 (7th gen, 2.5 GHz), with AVX2 flags activated (-mavx2). Results are shown in Table 4.3, where "?" means that either the reference or the new version of the process failed.

Using our new implementation process, we can observe a speedup up to 3.48 (for the **exponential** problem), which shows the interest of our approach.

An interesting point is that the speedup grows when the required target error decreases, and thus our solution efficiency increases with the polynomial degree and the complexity of the selected meta-blocks. Using multi-word floating-point arithmetics make programs more complex, and much slower. By using parallel schemes our method suffers less from this slow-down, since this parallelism allows for better exploitation of modern architecture capabilities.

However our new method was not able to generate a proper implementation for all target errors, in part due to the smaller meta-block diversity compared to the legacy implementpoly. It constitutes one of the current limitations of our method.

On small accuracy targets, our approach is slower or close to a speedup of 1. This is due to the fact that there is not as much parallelism to exploit for those error targets: the polynomial degree is generally smaller, requiring less operations.

**Conclusion**

This Section addresses the problem of generating fast and accurate programs for polynomial evaluation, in the context of the automated implementation of mathematical functions. The proposed approach consists in extending Metalibm-lutetia by considering parallel evaluation schemes and by plugging this tool into the Metalibm-lugdunum framework to benefit from its advanced features and the ability of its back-end in targeting modern vector architectures.

The work presented here led to the development of a first version of the tool. As shown in Section 4.4.2, this allows us to generate programs for a significant set of benchmarks in a few minutes, for which a speedup of up to 3.48 can be observed in practice on a AVX2 micro-architecture.

Our research direction is twofold: first our current implementation considers only binary64 (single or multi-word) floating-point arithmetic, thus limiting the scope of problems that can be tackled. A direct extension is to consider other IEEE floating-point formats, such as binary32. The genericity of our tool through the use of an "opaque" data type makes this treatable in the near future. However a more challenging extension is to adapt that work for other non-standard arithmetics, like fixed-point arithmetic. Second, as detailed in Section 4.4.2, programs are built so as their relative error satisfies the error constraint. It makes sense to adapt our tool to work with absolute error, instead, as in [33].

### 4.4.3 Metalibm Usage: Future Work

In Section 4.3 and in this Section 4.4, we have seen how our Metalibm-Lutetia tool can be combined with other tools, such as NumGfun [189], Metalibm-Lugdunum [32], CGPE [194, 218], in order to leverage various aspects of it: open-endedness by producing code for D-finite functions, performance on modern super-scalar processors as well as vectorized use on SIMD vector units. As a matter of course, this list of cooperation and code linkage has not yet been exhausted; many opportunities are left: for future work, we may consider combining metalibm-lutetia, e.g. with static code analysis tools such as Daisy [58], which we have tempted with prototypes already, or accuracy enhancement tools, such as Precimonious [222], or even Compilers, such as clang based on LLVM. We see a technical hindrance that is becoming pressingly important. It is the interpreted nature of the Sollya [41] language, used for Metalibm-Lutetia. Another hindrance is the absence of success rate guarantees for functions generation instances, in the sense that Metalibm guarantees the produced code to be correct when it succeeds to generate code. However it is currently unable to guarantee that code can be generated for a given function implementation problem input. We shall strive at addressing these issues in future work.

## 4.5 Tool-Chain Similarities with LTI Filter Implementation

In this Chapter 4, we have opened the floating-point environment with respect to code that is not written by humans and that stays static but that is obtained through a code generation process. We have done this work at the example of mathematical functions. As a matter of course, the use of mathematical functions is widespread in scientific computing [211] and even computer gaming makes heavy use of certain elementary functions [233]. However, another type of numerical software is tremendously more widespread: Linear Time Invariant (LTI) Filters. As we shall see in more detail in Chapter 5, these filters are devices used in Digital Signal Processing to attenuate or amplify certain spectral components of a digitized signal, such as an audio or machine control signal. They are used in common application devices like telephones, telecommunication equipment, cars, trains, airplanes, industrial machinery, hearing aids, etc. [250].

On the one hand, as an object that needs numerical implementation, LTI filters are quite similar to the object of the functions we have considered so far: numerical input is transformed into numerical output, according to rules that are given as analytical statements. An implementation using some kind of arithmetic is nothing but an approximation to the real, mathematical object. The resulting error needs to be kept as low as possible, while ensuring good performance. On the other hand, LTI filters are different from mathematical functions. First of all, functions do not have internal state, filters do [250]. Second, functions are often implemented in floating-point environments, whereas filter implementations on embedded, small processors often prefer fixed-point arithmetic. Third, while approximations to functions are most reasonably evaluated in terms of the maximum error [43], filter approximations need to be evaluated in other types of error, for example in terms of absolute noise energy induced by the filter [250]. Finally, it should be noted that while mathematical libraries (`libm`s) of standard functions make sense in scientific computing, filters are naturally open-ended: each implementation is a one-time use-case of the general concept of LTI filters. It is adapted in the design phase to the precise functionality it needs to deliver in this one only utilization [250].

Amazingly, LTI filters and mathematical functions also share commonalities in their implementation and in the way this implementation is derived from its specification:

— Functions are eventually approximated using polynomials, i.e. (repeated) sums of products of input values and precomputed constants. As we will see in more detail in Chapter 5, LTI filter implementations also use sums of products with precomputed constants. Sensitivity and accuracy analysis techniques for polynomials as well as approaches to map them onto superscalar hardware units can hence be reused for LTI filters, too.

— The approximation polynomials used for implementation of functions are computed using variants of the Remez algorithm. LTI filters coefficients are obtained for a given specification using the Parks-McClellan algorithm [205] which is nothing but a variant of the Remez algorithm [214, 215].

— The structure of a code generation algorithm, seen from a birdview perspective, for LTI filters is confronted with just the same issues a code generator for functions is confronted with: errors that stem from arithmetical evaluation mix with errors due to approximation. Code generation hence often contains optimization loops, which are not fully understood algorithmicly [250].

Due to these commonalities, it would hence make sense to share software between code generators for filters and functions. Unfortunately, we are not there, yet. At the time of writing, software reuse between metalibm and code generators for LTI filters, such as fixif[11] is reduced to a common base in the form of Sollya. However, in future work, we strive at making the code base even more shared, which is a kind of work that may start with enhancements to Sollya directly aimed towards LTI filters. We shall present the start of this work in Chapter 5.

## 4.6 Code Generators: Lessons Learned and Open Questions

With our work on code generators for mathematical functions we have strove at extending the floating-point environment, as it is offered e.g. by IEEE754-2008. Producing an implementation of a mathematical function has become really easy for the numerical developer, so its cost has been considerably reduced. For instance, as shown in Section 4.4.1, we could leverage our metalibm tool for the design of a vectorized mathematical library. The overall

11. `https://github.com/fixif/fixif`

effort for all the 9 functions in the library was less than the effort we had put into the development of one of the functions we implemented in CRLibm in the past [57,165]. It was also possible to leverage metalibm to provide implementations of functions which no other implementation was known for in the literature; we showed these approaches in Section 4.3.

We have started working on other subjects, such as LTI filters, for which code generation might also make sense; we discussed this point in Section 4.5 and we will look at it again in Chapter 5. In the future, we would like to leverage the opportunities offered by code generation techniques also for other domains, such as Kalman filters or code used in machine learning. We shall explain the details of these projects in more detail in Chapter 6, Section 6.2.3.

One of the lessons that we have learned with our work on code generators is that even though –and because– there is a change at which level algorithmic development takes place, the design of a code generator for a specific domain needs very precise knowledge of the algorithms used in this domain first. Simply put, it is hard to write a code generator for a class of say exponential functions $a^x$ without having written such functions manually a couple of times. While this seems natural, it might become a major hindrance for the further democratization of code generation techniques for numerical code: we would rather like to be able to abstract from the domain-specific knowledge. For other code transformation tools, such as compilers for languages like C, Fortran, Java etc. this is already the case; it suffices to look at classical books on compiler design to see that compilation technology is both language- and architecture-agnostic [104]. A compiler-designer for a certain language can build on the general, established principles for the design of a compiler, while the designer of a numerical code generator has no such guidance, yet.

Open questions in numerical code generation also include finding the right place for static and dynamic code analysis. Even though we would like to start code generation from the most basic, mathematical description of a problem, software engineers are often confronted with the issue that the only description that does exist is legacy code. Even though we would like to start code generation from an easy-to-analyze mathematical description, technologically it might also make sense to start code generation with existing code. For these applications, static and also dynamic code analysis might help. A possible scheme aimed at increased performance is to replace existing code only for very heavily used input values, keeping the legacy code as a fall-back for the rarer "outlier" inputs. Using approaches like the one described by Piparo [210] may be helpful for this purpose. Work on integration with static code analysis has already started [58]; we shall continue on this path.

CHAPTER 5

# A priori Error Control in Multiprecision Floating-Point Arithmetic

*Errare humanum est, sed in errore perseverare diabolicum.*

Cicero, *Orationes Philippicae*

## 5.1 Introduction

As we have already seen in the previous Chapter 4, Section 4.5, code generation for Linear Time Invariant (LTI) Filters, which mostly use fixed-point arithmetic internally, is quite similar to code generation for elementary functions. This aspect has encouraged us to invest ourselves into the research field of LTI Filters, in particular into analyzing the behavior of LTI Filters, as they undergo the different steps of a code generation process.

Code generation is always concerned with producing code that is optimal with respect to certain criteria, as we have seen already in Chapter 4. For LTI Filters, the most important criteria are the noise introduced by the filter during the filtering of a signal as well as power consumption and cost of evaluation, be it measured in area on an FPGA or ASIC or in terms of basic operations on a DSP [250].

When starting to speak about "optimal" implementations, one of course easily gets onto thin ice, in particular when the respective criterion can only be measured with a certain error. The question is whether the "optimal" implementation has really reached an extremum point or whether we simply can see clear around that extremum point, due to the error. Interval arithmetic can often help in these circumstances, allowing a statement about some measure to be made rigorous, with both a lower and an upper bound. We have used some *a posteriori* certificates in the past, for instance when we computed the maximum error of a polynomial with respect to a function it was supposed to approximate with interval arithmetic techniques [40].

*A posteriori* techniques however have the disadvantage of often requiring human insight into problems: it is up to a human to understand that the compute precision at a certain step in a generation process had not been set to an appropriate level and that the fact that the final result does not fulfill this or that criterion is due to this one step. When we strive for full automation, we should rather have the computer adapt the compute precision to an appropriate level in the first place.

*A priori* error control means exactly this: we wish to design numerical algorithms that solve a certain computational problem rigorously, viz. by providing an output interval, but we want at the same time these algorithms to adapt their compute precision automatically, so that we are sure that the interval eventually output by the algorithm will satisfy a certain *a priori* error bound.

It is easy to see that most interval arithmetic-based algorithms can be forced to satisfy such an *a priori* error specification: with compute precision set to some initial value heuristically, the algorithm is run, which yields an interval. If that interval is accurate enough, it is output as the interval fulfilling the given *a priori* error bound. If not, the compute precision is increased and the algorithm is re-run. This process continues until the error bound is met. This scheme is easy to implement but it is not always very efficient.

We hence tried to study other techniques in computer arithmetic that would allow for the computation of measures on LTI filters in one single pass, adapting precision as needed to meet the *a priori* error bound at once. We succeeded to design such an algorithm for the so-called Worst-Case Peak Gain (WCPG) measure; we are going to present this algorithm in Section 5.2. For other LTI implementation problems, we go still one level up, utilizing our WCPG algorithm as a basic brick, in order to statically determine the compute precisions –in terms of fixed-point MSB and LSB positions– needed for LTI filter evaluation with a given noise error bound. We will present this work in Section 5.3. Unfortunately, for certain very particular instances, we will need to fall back to a precision increasing scheme again.

## 5.2 Precision Adaptation in LTI Filter Analysis

This Section on precision adaptation in LTI filter analysis is based on an article we published in [252]. It was our first step into rigorous multiple precision computer arithmetic that adapts precision automatically with a scheme more complex than a compute-check-increase-precision loop.

### 5.2.1 Introduction

The majority of control and digital signal processing algorithms is dedicated to linear time-invariant (LTI) systems with finite or infinite impulse response. Most of them are implemented for application in embedded systems, which use finite-precision arithmetic. Unfortunately, the quantification of coefficients and further roundoff errors lead to degradation of the algorithms. Therefore, an accurate error analysis of implementation of such algorithms is required.

However, this analysis is complicated by the non-linear propagation of errors through the filter as they are amplified on each step by internal state of the system. A solution is proposed in [113], based on a property of bounded-input bounded output systems [13,35] where the largest possible peak value of the output is determined by the use of the Worst-Case Peak Gain (WCPG) matrix. Error propagation analysis in LTI systems is directly dependent on the reliable evaluation of the WCPG.

This measure is computed with an infinite sum and it has matrix powers in each summand. These problems are both known to be non-trivial. In this Section we propose a detailed algorithm for the reliable evaluation of the WCPG matrix with multiple precision. This algorithm ensures that the WCPG is computed with an absolute error rigorously bounded by an *a priori* given value $\varepsilon$. For these purposes several multiprecision algorithms for complex entries with rigorous bounds were developed. This is achieved by adapting the precision of intermediate computations and correct rounding. Therefore, we present not only the error analysis of the approximations made on each step of the WCPG computation, but we also deduce the required accuracy for our kernel multiprecision algorithms such that the overall error bound is satisfied.

We analyze the error induced by truncating the infinite sum and a direct formula for the computation of a lower bound on truncation order for a desired absolute error. The truncation order algorithm involves Interval Arithmetic computations and uses Theory of Verified Inclusions.

Some preliminary definitions about LTI systems are recalled in Section 5.2.2. Section 5.2.3 describes the global algorithm used to reliably evaluate the WCPG matrix. The truncation order and the truncation error are analyzed in Section 5.2.4. Section 5.2.5 is focused on the different steps used for the summation and the associated error analysis, whereas Section 5.2.6 details some basic bricks in multiple precision. Finally, numerical examples are presented in Section 5.2.7 before conclusion.

**Notation:** Throughout this Section matrices are in uppercase boldface (for example $\boldsymbol{A}$), vectors are in lowercase boldface (for example $\boldsymbol{v}$), scalars are in lowercase (for example $\alpha$). Operators $\otimes$, and $\oplus$ denote floating-point (FP) multiplication and addition respectively, $\mathbb{F}$ the set of FP numbers. $[x]$ corresponds to an interval. $\boldsymbol{A}^*$ denotes the conjugate transpose of the matrix $\boldsymbol{A}$. All matrix absolute values and inequalities are considered to be element-by-element, for example $|\boldsymbol{A}| < |\boldsymbol{B}|$ denotes $|\boldsymbol{A}_{ij}| < |\boldsymbol{B}_{ij}|\ \forall i,j$. In addition, $\boldsymbol{A} < \varepsilon$ denotes $\boldsymbol{A}_{ij} < \varepsilon\ \forall i,j$. $\boldsymbol{I}_n$ denotes the identity matrix of size $n \times n$ and $\rho(\boldsymbol{A})$ the spectral radius of $\boldsymbol{A}$.

In further discussions the error matrices are bounded in respect to their Frobenius norm. The Frobenius norm is sub-multiplicative and it has several good properties used in numerical analysis. Let $\boldsymbol{K}$ be some matrix, $\boldsymbol{K} \in \mathbb{C}^{n\times m}$, then

$$|\boldsymbol{K}_{ij}| \leqslant \|\boldsymbol{K}\|_F\ \forall i,j \tag{5.1}$$

$$\|\boldsymbol{K}\|_2 \leqslant \|\boldsymbol{K}\|_F \leqslant \sqrt{\min(m,n)}\,\|\boldsymbol{K}\|_2\,, \tag{5.2}$$

where $\|\boldsymbol{K}\|_2$ is the spectral-norm, i.e. equal to the largest singular value of $\boldsymbol{K}$.

Moreover, if $\boldsymbol{K}$ is a square $n \times n$ matrix such that $\|\boldsymbol{K}\|_2 \leqslant 1$, then for all $k$, $\left\|\boldsymbol{K}^k\right\|_2 \leqslant 1$ and $\left\|\boldsymbol{K}^k\right\|_F \leqslant \sqrt{n}$.

### 5.2.2 The Worst-Peak Gain Measure

A Linear Time Invariant (LTI) filter is a system used in signal processing, image processing, control theory, etc. It is defined by an input-output relationship in time-domain or equivalently in frequency-domain. Linear controllers, Finite Impulse Response (FIR) filters, Infinite Impulse Response (IIR) are classical examples of LTI systems. We focus here only on discrete-time systems: a discrete-time LTI system (filter) is a numerical application that transforms an input signal $\{\boldsymbol{u}(k)\}_{k\geqslant 0}$ into an output signal $\{\boldsymbol{y}(k)\}_{k\geqslant 0}$ ($\boldsymbol{u}(k)$ and $\boldsymbol{y}(k)$ may be vectors or scalars), where $k \in \mathbb{N}$ is the step time.

A common input-output relationship is the state-space representation [133]. It describes the evolution of the state vector $\boldsymbol{x}(k)$ from the previous step and the input:

$$\mathcal{H}\left\{\begin{array}{rcl} \boldsymbol{x}(k+1) &=& \boldsymbol{A}\boldsymbol{x}(k)+\boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &=& \boldsymbol{C}\boldsymbol{x}(k)+\boldsymbol{D}\boldsymbol{u}(k) \end{array}\right. \tag{5.3}$$

where $\boldsymbol{u}(k) \in \mathbb{R}^{q\times 1}$ is the input vector, $\boldsymbol{y}(k) \in \mathbb{R}^{p\times 1}$ the output vector, $\boldsymbol{x}(k) \in \mathbb{R}^{n\times 1}$ the state vector and $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{B} \in \mathbb{R}^{n\times q}$, $\boldsymbol{C} \in \mathbb{R}^{p\times n}$ and $\boldsymbol{D} \in \mathbb{R}^{p\times q}$ are the state-space matrices of the system. Unlike a mathematical function, the output at time $k$ depends not only on in the input at time $k$ but also on the internal state of the filter (generally determined from the previous inputs and outputs).

**Proposition 5.2.1** (Bounded Input Bounded Output systems). *Let $\mathcal{H}$ be a state-space system. If an input $\{\boldsymbol{u}(k)\}_{k\geqslant 0}$ is known to be bounded by $\bar{\boldsymbol{u}}$ ($\forall k \geqslant 0, \quad |\boldsymbol{u}_i(k)| \leqslant \bar{\boldsymbol{u}}_i, \quad 1 \leqslant i \leqslant q$), then the output $\{\boldsymbol{y}(k)\}_{k\geqslant 0}$ will be bounded iff the spectral radius $\rho(\boldsymbol{A})$ is strictly less than 1. This property is known as the Bounded Input Bounded Output (BIBO) stability [133].*

*Moreover, in that case, the output is (component-wise) bounded by $\bar{\boldsymbol{y}}$ with $\bar{\boldsymbol{y}} = \boldsymbol{W}\bar{\boldsymbol{u}}$ where $\boldsymbol{W} \in \mathbb{R}^{p\times q}$ is the Worst-Case Peak Gain matrix [13] of the system, defined by*

$$\boldsymbol{W} := |\boldsymbol{D}| + \sum_{k=0}^{\infty} \left|\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}\right| \tag{5.4}$$

*Proof.* Let $\{\boldsymbol{J}(k)\}_{k\geqslant 0}$ be the impulse response matrix of the system, i.e. $\boldsymbol{J}_{ij}(k)$ is the response on the $i^{\text{th}}$ output to the Dirac impulse at time $k=0$ (i.e. $\delta(0)=1$ and $\delta(k)=0, \quad \forall k \neq 0$) on the $j^{\text{th}}$ input. With (5.53), we have

$$\boldsymbol{J}(k) = \begin{cases} \boldsymbol{D} & \text{if } k=0 \\ \boldsymbol{C}\boldsymbol{A}^{k-1}\boldsymbol{B} & \text{if } k>0. \end{cases} \tag{5.5}$$

Since the input $\{\boldsymbol{u}(k)\}_{k\geqslant 0}$ can be seen as a weighted sum of Dirac impulses (shifted in time), and thanks to the linearity and time invariance property of LTI systems [133], we get

$$\boldsymbol{y}(k) = \sum_{l=0}^{k} \boldsymbol{J}(l)\boldsymbol{u}(k-l). \tag{5.6}$$

($\{\boldsymbol{y}\}_{k\geqslant 0}$ is the result of the convolution of $\{\boldsymbol{J}\}_{k\geqslant 0}$ by $\{\boldsymbol{u}\}_{k\geqslant 0}$). Then the output is (component-wise) bounded by

$$\boldsymbol{y}(k) \leqslant \left(\sum_{l=0}^{k} |\boldsymbol{J}(l)|\right)\bar{\boldsymbol{u}}, \quad \forall k \geqslant 0. \tag{5.7}$$

We have equality for the $i^{\text{th}}$ output if the input is such that $\boldsymbol{u}_j(l) = \bar{\boldsymbol{u}} \cdot \operatorname{sign}\big(\boldsymbol{J}_{ij}(k-l)\big)$, $\forall 0 \leqslant l \leqslant k$, where $\operatorname{sign}(x)$ returns $\pm 1$ or $0$ depending on the sign of $x$. Finally

$$\forall k \geqslant 0, \quad \boldsymbol{y}(k) < \left(\sum_{l=0}^{\infty} |\boldsymbol{J}(k)|\right)\bar{\boldsymbol{u}}. \tag{5.8}$$

□

**Remark 1.** *$\boldsymbol{W}\bar{\boldsymbol{u}}$ is the supremum of the output $\{\boldsymbol{y}\}_{k\geqslant 0}$, since it is possible to build a finite input $\{\boldsymbol{u}(k)\}_{0\leqslant k\leqslant K}$ to approach it on any given output at any given distance.*

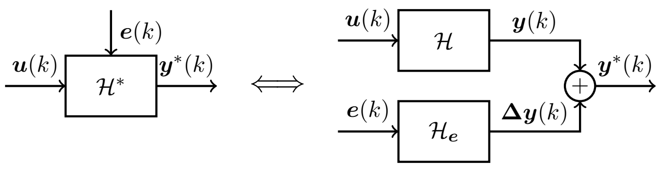

Figure 5.1 – The implemented filter is equivalent to the exact filter where the output is corrupted by the computational errors passing themselves through a filter.

**Remark 2.** *This proposition can be completed when considering intervals for the input, instead of bounds (corresponding to symmetric intervals). In that case, the Worst-Case Peak Gain matrix indicates by how much the radius of the input interval is amplified on the output [113] (although this is not valid for the transient phase, i.e. for the few first steps). However, even in that case,* $\boldsymbol{W}\bar{\boldsymbol{u}}$ *is a supremum we need to compute.*

This proposition can be used to bound outputs, states or intermediate variables in the context of finite precision implementation of algorithms, and, more specifically, in Fixed-Point arithmetic. In [112], an extension of the state-space has been presented, in order to represent and encompass all the possible algorithms for linear filters (i.e. all the input-to-output data flows based on additions, multiplications by constant and delay, such as state-space, direct forms, $\rho$DFIIt [261], etc.), and the same approach was applied.

First, it is used to bound all the variables involved in the algorithm, and then to determine their fixed-point representation (position of the Most Significant Bit and scaling) while preserving by construction from overflow.

Second, it is used to determine the impact on the output of the computational errors. Classical error analysis cannot be used in that context due to the feedback scheme of the computation (Interval Arithmetic or Affine Arithmetic do not provide tight bounds [181]).

Since the filter is linear, the implemented filter $\mathcal{H}^*$ can be seen as the exact filter $\mathcal{H}$ where the output is corrupted by the vector of errors $\boldsymbol{e}(k)$ occurring at each sum of product through a given linear filter $\mathcal{H}_e$ (see Figure 5.1).

A State-space representation of $\mathcal{H}_e$ can be obtained analytically [113] and Proposition 5.2.1 can be used to determine the output error $\boldsymbol{\Delta y}$ due to finite-precision arithmetic.

For all these reasons, the reliable computation of the Worst-Case Peak Gain matrix is a required step for the accurate error analysis of LTI systems in finite precision.

### 5.2.3 An Algorithm with *a priori* Accuracy

Given a BIBO stable LTI filter in state-space realization (5.53) and $\varepsilon$, a desired absolute approximation error, we want to determine the Worst-Case Peak Gain matrix $\boldsymbol{W}$ of this filter, defined in (5.4). While computing such an approximation, various errors, such as truncation and summation errors, are made.

Instead of directly computing the infinite sum $\left|\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}\right|$ for any $k \geqslant 0$, we will use an approximate eigenvalue decomposition of $\boldsymbol{A}$ (i.e. $\boldsymbol{A} \approx \boldsymbol{V}\boldsymbol{T}\boldsymbol{V}^{-1}$) and compute the FP sum $\left|\boldsymbol{C}\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}\boldsymbol{B}\right|$ for $0 \leqslant k \leqslant N$.

Our approach to compute the approximation $\boldsymbol{S}_N$ of $\boldsymbol{W}$ is summarized in algorithm 9 where all the operations ($\otimes$, $\oplus$, inv, abs, etc.) are FP multiple precision operations done at

```
  **Input:** $\boldsymbol{A} \in \mathbb{F}^{n\times n}, \boldsymbol{B} \in \mathbb{F}^{n\times q}, \boldsymbol{C} \in \mathbb{F}^{p\times n}, \boldsymbol{D} \in \mathbb{F}^{p\times q}, \varepsilon > 0$
  **Output:** $\boldsymbol{S}_N \in \mathbb{F}^{p\times q}$
1 Compute $N$
2 Compute $\boldsymbol{V}$ from an eigendecomposition of $\boldsymbol{A}$
  $\boldsymbol{T} \leftarrow \text{inv}(\boldsymbol{V}) \otimes \boldsymbol{A} \otimes \boldsymbol{V}$
  if $\|\boldsymbol{T}\|_2 > 1$ then return $\perp$
3 $\boldsymbol{B}' \leftarrow \text{inv}(\boldsymbol{V}) \otimes \boldsymbol{B}$
  $\boldsymbol{C}' \leftarrow \boldsymbol{C} \otimes \boldsymbol{V}$
  $\boldsymbol{S}_{-1} \leftarrow |\boldsymbol{D}|, \quad \boldsymbol{P}_{-1} \leftarrow \boldsymbol{I}_n$
  for $k$ from $0$ to $N$ do
4     $\boldsymbol{P}_k \leftarrow \boldsymbol{T} \otimes \boldsymbol{P}_{k-1}$
5     $\boldsymbol{L}_k \leftarrow \boldsymbol{C}' \otimes \boldsymbol{P}_k \otimes \boldsymbol{B}'$
6     $\boldsymbol{S}_k \leftarrow \boldsymbol{S}_{k-1} \oplus \text{abs}(\boldsymbol{L}_k)$
  end
  return $\boldsymbol{S}_N$
```

**Algorithm 9:** Floating-point evaluation of the WCPG.

various precisions to be determined such that the overall error is less than $\varepsilon$:

$$|\boldsymbol{W} - \boldsymbol{S}_N| \leqslant \varepsilon. \tag{5.9}$$

The overall error analysis decomposes into 6 steps, where each one expresses the impact of a particular approximation (or truncation), and provides the accuracy requirements for the associated operations such that the result is rigorously bounded by $\varepsilon$. These steps are discussed in detail in Section 5.2.4 and 5.2.5:

**Step 1:** Let $\boldsymbol{W}_N$ be the truncated sum

$$\boldsymbol{W}_N := \sum_{k=0}^{N} \left|\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}\right| + |\boldsymbol{D}|\,. \tag{5.10}$$

We compute a truncation order $N$ of the infinite sum $\boldsymbol{W}$ such that the truncation error is less than $\varepsilon_1 > 0$:

$$|\boldsymbol{W} - \boldsymbol{W}_N| \leqslant \varepsilon_1. \tag{5.11}$$

See Section 5.2.4 for more details.

**Step 2:** Error analysis for computing the powers $\boldsymbol{A}^k$ of a full matrix $\boldsymbol{A}$, when the $k$ reaches several hundreds, is a significant problem, especially when the norm of $\boldsymbol{A}$ is larger than 1 and its eigenvalues are close to 1. However, if $\boldsymbol{A}$ may be represented as $\boldsymbol{A} = \boldsymbol{X}\boldsymbol{E}\boldsymbol{X}^{-1}$ with $\boldsymbol{E} \in \mathbb{C}^{n\times n}$ strictly diagonal and $\boldsymbol{X} \in \mathbb{C}^{n\times n}$, then powering of $\boldsymbol{A}$ reduces to powering the diagonal matrix $\boldsymbol{E}$, which is more convenient.

Suppose we have a matrix $\boldsymbol{V}$ approximating $\boldsymbol{X}$. We require this approximation to be just quite accurate so that we are able to discern the different associated eigenvalues and be sure their absolute values are less than 1.

We may then consider the matrix $\boldsymbol{V}$ to be exact and compute an approximation $\boldsymbol{T}$ to $\boldsymbol{V}^{-1}\,\boldsymbol{A}\,\boldsymbol{V}$ with sufficient accuracy such that the error of computing $\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}$ instead of matrix $\boldsymbol{A}^k$ is less than $\varepsilon_2 > 0$:

$$\left|\boldsymbol{W}_N - \sum_{k=0}^{N}\left|\boldsymbol{C}\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}\boldsymbol{B}\right|\right| \leqslant \varepsilon_2. \tag{5.12}$$

See Section 5.2.5.

**Step 3:** We compute approximations $\boldsymbol{B}'$ and $\boldsymbol{C}'$ of $\boldsymbol{V}^{-1}\boldsymbol{B}$ and $\boldsymbol{C}\boldsymbol{V}$, respectively. We require that the propagated error committed in using $\boldsymbol{B'}$ instead of $\boldsymbol{V}^{-1}\boldsymbol{B}$ and $\boldsymbol{C}'$ instead of $\boldsymbol{C}\boldsymbol{V}$ be less than $\varepsilon_3 > 0$:

$$\left|\sum_{k=0}^{N}\left|\boldsymbol{C}\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}\boldsymbol{B}\right| - \sum_{k=0}^{N}\left|\boldsymbol{C}'\boldsymbol{T}^k\boldsymbol{B}'\right|\right| \leqslant \varepsilon_3. \tag{5.13}$$

See Section 5.2.5.

**Step 4:** We compute in $\boldsymbol{P}_k$ the powers $\boldsymbol{T}^k$ of $\boldsymbol{T}$ with a certain accuracy. It is required that the error be less than $\varepsilon_4 > 0$:

$$\left|\sum_{k=0}^{N}\left|\boldsymbol{C'}\boldsymbol{T}^k\boldsymbol{B'}\right| - \sum_{k=0}^{N}\left|\boldsymbol{C'}\boldsymbol{P}_k\boldsymbol{B'}\right|\right| \leqslant \varepsilon_4. \tag{5.14}$$

See Section 5.2.5.

**Step 5:** We compute in $\boldsymbol{L}_k$ each summand $\boldsymbol{C'}\boldsymbol{P}_k\boldsymbol{B'}$ with a error small enough such that the overall approximation error induced by this step is less than $\varepsilon_5 > 0$:

$$\left|\sum_{k=0}^{N}\left|\boldsymbol{C'}\boldsymbol{P}_k\boldsymbol{B'}\right| - \sum_{k=0}^{N}\left|\boldsymbol{L}_k\right|\right| \leqslant \varepsilon_5. \tag{5.15}$$

See Section 5.2.5.

**Step 6:** Finally, we sum $\boldsymbol{L}_k$ in $\boldsymbol{S}_N$ with enough precision so that the absolute error bound for summation is bounded by $\varepsilon_6 > 0$:

$$\left|\sum_{k=0}^{N}\left|\boldsymbol{L}_k\right| - \boldsymbol{S}_N\right| \leqslant \varepsilon_6. \tag{5.16}$$

See Section 5.2.5.

By ensuring that each step verifies its bound $\varepsilon_i$, and taking $\varepsilon_i = \frac{1}{6}\varepsilon$, we get $\varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_4 + \varepsilon_5 + \varepsilon_6 \leqslant \varepsilon$, hence (5.9) will be satisfied if inequalities (5.11) to (5.16) are.

Our approach hence determines first a truncation order $N$ and then performs summation up to that truncation error, whilst adjusting precision in the different summation steps. A competing approach would be not to start with truncation order determination but to immediately go for summation and to stop when adding more terms does not improve accuracy. However, such an approach would not allow the final error to be bounded in an *a priori* way. As we shall see, the multiple precision FP summation needs to know a bound on the number of terms to be summed, beforehand.

### 5.2.4 Truncation Order and Truncation Error

In [13] Balakrishnan and Boyd propose "simple" lower and upper bounds on $N$. However, they describe their algorithm in terms of exact arithmetic, *i.e.* do not propose any error analysis. This iterative algorithm has several difficulties: first of all, matrix $\boldsymbol{A}$ exponentiation is present, which would require an error analysis such as the one proposed in that article. Secondly, on each iteration (the quantity of which may reach order as high as N) a solution of Lyapunov equations is required, for which there exist no ready to use solution with rigorous error bounds on the result. Therefore, a different approach is indispensable. In this Section we propose a direct formula for the lower bound on $N$ along with a reliable evaluation algorithm.

The goal is to determine a lower bound on the truncation order N of the infinite sum (5.4) such that its tail is smaller than the given $\varepsilon_1$. Obviously, $\boldsymbol{W}_N$ is a lower bound on $\boldsymbol{W}$ and increases monotonically to $\boldsymbol{W}$ with increasing $N$. Hence the truncation error is

$$|\boldsymbol{W} - \boldsymbol{W}_N| = \sum_{k>N} \left|\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}\right|. \tag{5.17}$$

#### A bound on the truncation error

Many simple bounds on (5.17) are possible. For instance, if the eigendecomposition of $\boldsymbol{A}$ is computed

$$\boldsymbol{A} = \boldsymbol{X}\boldsymbol{E}\boldsymbol{X}^{-1} \tag{5.18}$$

where $\boldsymbol{X} \in \mathbb{C}^{n\times n}$ is the right hand eigenvector matrix, and $\boldsymbol{E} \in \mathbb{C}^{n\times n}$ is a diagonal matrix holding the eigenvalues $\boldsymbol{\lambda}_l$, the terms $\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}$ can be written

$$\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B} = \boldsymbol{\Phi}\boldsymbol{E}^k\boldsymbol{\Psi} = \sum_{l=1}^{n} \boldsymbol{R}_l\boldsymbol{\lambda}_l^k \tag{5.19}$$

where $\boldsymbol{\Phi} \in \mathbb{C}^{p\times n}$, $\boldsymbol{\Psi} \in \mathbb{C}^{n\times q}$ and $\boldsymbol{R}_l \in \mathbb{C}^{p\times q}$ are defined by

$$\boldsymbol{\Phi} := \boldsymbol{C}\boldsymbol{X}, \qquad \boldsymbol{\Psi} := \boldsymbol{X}^{-1}\boldsymbol{B}, \qquad (\boldsymbol{R}_l)_{ij} := \boldsymbol{\Phi}_{il}\boldsymbol{\Psi}_{lj}. \tag{5.20}$$

In this setting, we obtain

$$|\boldsymbol{W} - \boldsymbol{W}_N| = \sum_{k>N} \sum_{l=1}^{n} \left|\boldsymbol{R}_l\boldsymbol{\lambda}_l^k\right|. \tag{5.21}$$

As required by Proposition 5.2.1, all eigenvalues $\boldsymbol{\lambda}_l$ of matrix $\boldsymbol{A}$ must be strictly smaller than one in magnitude. We may therefore notice that the outer sum is in geometric progression with a common ratio $|\boldsymbol{\lambda}_l| < 1$. So the following bound is possible (we remind the reader that inequalities and absolute values are considered to be element by element):

$$\begin{aligned} |\boldsymbol{W} - \boldsymbol{W}_{\boldsymbol{N}}| &\leqslant \sum_{k=N+1}^{\infty} \sum_{l=1}^{n} |\boldsymbol{R}_l| \left|\boldsymbol{\lambda}_l^k\right| \leqslant \sum_{l=1}^{n} |\boldsymbol{R}_l| \frac{\left|\boldsymbol{\lambda}_l^{N+1}\right|}{1 - |\boldsymbol{\lambda}_l|} \\ &= \rho(\boldsymbol{A})^{N+1} \sum_{l=1}^{n} \frac{|\boldsymbol{R}_l|}{1 - |\boldsymbol{\lambda}_l|} \left(\frac{|\boldsymbol{\lambda}_l|}{\rho(\boldsymbol{A})}\right)^{N+1}. \end{aligned} \tag{5.22}$$

Since $\frac{|\boldsymbol{\lambda}_l|}{\rho(\boldsymbol{A})} \leqslant 1$ holds for all terms, we may leave out the powers. Notate

$$\boldsymbol{M} := \sum_{l=1}^{n} \frac{|\boldsymbol{R}_l|}{1-|\boldsymbol{\lambda}_l|} \frac{|\boldsymbol{\lambda}_l|}{\rho(\boldsymbol{A})} \in \mathbb{R}^{p \times q}. \tag{5.23}$$

The tail of the infinite sum is hence bounded by

$$|\boldsymbol{W} - \boldsymbol{W}_N| \leqslant \rho(\boldsymbol{A})^{N+1} \boldsymbol{M}. \tag{5.24}$$

**Remark 3.** *Another tighter bound is possible*

$$|\boldsymbol{W} - \boldsymbol{W_N}| \leqslant \rho(\boldsymbol{A})^{N+1-K} \sum_{l=1}^{n} \frac{|\boldsymbol{R}_l|}{1-|\boldsymbol{\lambda}_l|} \left(\frac{|\boldsymbol{\lambda}_l|}{\rho(\boldsymbol{A})}\right)^K, \quad \forall N > K. \tag{5.25}$$

**Deducing a lower bound on the truncation order**

In order to get (5.24) bounded by $\varepsilon_1$, it is required that

$$\rho(\boldsymbol{A})^{N+1} \boldsymbol{M} \leqslant \varepsilon_1.$$

Solving this inequality for $N$ leads us to the following bound:

$$N \geqslant \left\lceil \frac{\log \frac{\varepsilon_1}{m}}{\log \rho(\boldsymbol{A})} \right\rceil \tag{5.26}$$

where $m$ is defined as $m := \min\limits_{i,j} |\boldsymbol{M}_{i,j}|$.

However we cannot compute exact values for all quantities occurring in (5.26) when using finite-precision arithmetic. We only have approximations for them. Thus, in order to reliably determine a lower bound on $N$, we must compute lower bounds on $m$ and $\rho(\boldsymbol{A})$, from which we can deduce an upper bound on $\log \frac{\varepsilon_1}{m}$ and a lower bound on $\log \rho(\boldsymbol{A})$ to eventually obtain a lower bound on $N$.

**A rigorous algorithm to determine truncation order**

Due to the implementation of (5.18) and (5.20) with the finite-precision arithmetic, only approximations on $\boldsymbol{\lambda}, \boldsymbol{X}, \boldsymbol{\Phi}, \boldsymbol{\Psi}, \boldsymbol{R}_l$ can be obtained. There exist many FP libraries, such as LAPACK [1], providing functions for an eigendecomposition as needed for (5.18) and to solve linear systems of equations in (5.20). They usually deliver good and fast approximations to the solution of a given numerical problem but there is neither verification nor guarantee about the accuracy of that approximation.

For these reasons we propose to combine LAPACK FP arithmetic with Interval Arithmetic [59] enhanced with the Theory of Verified Inclusions [223], [224] , [226], [227] in order to obtain trusted intervals on the eigensystem and, eventually, a rigorous bound on $N$.

In Interval Arithmetic real numbers are represented as sets of reals with addition, subtraction, multiplication and division defined [59]. The Theory of Verified Inclusions is a set of algorithms computing guaranteed bounds on solutions of various numerical problems,

1. `http://www.netlib.org/lapack/`

developed by S. Rump [223]. The verification process is performed by means of checking an interval fixed point and yields to a trusted interval for the solution, i.e. it is verified that the result interval contains an exact solution of given numerical problem.

It permits us to quickly obtain trusted error bounds on the truncation order without significant impact on algorithm performance, since this computation is done only once. In addition, if the spectral radius of $\boldsymbol{A}$ cannot be shown less than 1, we stop the algorithm.

Using the ideas proposed by Rump in [227], we obtain trusted intervals for the eigensystem with the following steps:

1. Using the LAPACK eigensolver, we compute FP approximations $\boldsymbol{V}$ for the eigenvectors $\boldsymbol{X}$ and $\boldsymbol{\alpha}$ for the eigenvalues $\boldsymbol{\lambda}$, along with error estimates $\varepsilon_X$ and $\varepsilon_\lambda$. These error estimates are such that $|\boldsymbol{\lambda} - \boldsymbol{\alpha}| \leqslant \varepsilon_\lambda$ and $|\boldsymbol{X} - \boldsymbol{V}| \leqslant \varepsilon_X$ should be not far from the truth.

2. We construct, verify and possibly adjust intervals for $[\boldsymbol{\lambda}] = [\boldsymbol{\alpha} - \varepsilon_\lambda, \boldsymbol{\alpha} + \varepsilon_\lambda]$ and $[\boldsymbol{X}] = [\boldsymbol{V} - \varepsilon_X, \boldsymbol{V} + \varepsilon_X]$ such that for all vectors $\boldsymbol{\lambda}' \in [\boldsymbol{\lambda}]$ there exists a matrix $\boldsymbol{X}' \in [\boldsymbol{X}]$ satisfying $\boldsymbol{A}\boldsymbol{X}' = \boldsymbol{X}' \cdot \mathrm{diag}(\boldsymbol{\lambda}')$ and such that for all matrices $\boldsymbol{X}' \in [\boldsymbol{X}]$ there exists a vector $\boldsymbol{\lambda}' \in [\boldsymbol{\lambda}]$ satisfying $\boldsymbol{A}\boldsymbol{X}' = \boldsymbol{X}' \cdot \mathrm{diag}(\boldsymbol{\lambda}')$. In this process, first intervals for the eigensystem are constructed from the error estimates $\varepsilon_\alpha$ and $\varepsilon_V$ as radii and the approximate solutions $\boldsymbol{V}$ and $\boldsymbol{\alpha}$ as mid-points. Further, these intervals are verified with inclusion algorithms [227]. If the verification does not succeed, the intervals are extended by some small factor and process is repeated until it succeeds or until there exists an eigenvalue interval which contains 1.

For the solution of the linear system of equations (LSE) appearing in (5.20), the algorithm for interval verification is based on [224] and it consists of two steps:

1. Using LAPACK, compute a FP approximation $\boldsymbol{\Omega}$ on the solution of $\boldsymbol{V}\boldsymbol{\Psi} = \boldsymbol{B}$ along with an error estimate $\varepsilon_\Psi$ such that $|\boldsymbol{\Psi} - \boldsymbol{\Omega}| \leqslant \varepsilon_\Psi$ should be not far from the truth.

2. Construct, verify and adjust intervals $[\boldsymbol{\Psi}] = [\boldsymbol{\Omega} - \varepsilon_\Psi, \boldsymbol{\Omega} + \varepsilon_\Psi]$ such that for all matrices $\boldsymbol{X}' \in [\boldsymbol{X}]$ there exists $\boldsymbol{\Psi}' \in [\boldsymbol{\Psi}]$ such that $\boldsymbol{X}'\boldsymbol{\Psi}' = \boldsymbol{B}$ holds.

The intervals for verification are constructed in the same way as for the eigensystem solution. We require the existence of the exact solution of the linear system not for the system $\boldsymbol{V}\boldsymbol{\Psi} = \boldsymbol{B}$ but for $[\boldsymbol{X}]\boldsymbol{\Psi} = \boldsymbol{B}$, i.e. $[\boldsymbol{\Psi}]$ must contain the exact solution for each element of the already verified interval $[\boldsymbol{X}]$.

Finally, the intervals for (5.20), (5.23) and (5.26) are computed with Interval Arithmetic. Our complete algorithm to determine a reliable lower bound on $N$ is given with algorithm 10.

### 5.2.5 Summation

Once the truncation order determined, we need to provide a summation scheme reliable in FP arithmetic, i.e. such that the error of computations is bounded by an *a priori* given value. To do so we propose to perform all operations in multiple precision arithmetic whilst adapting precision dynamically, where needed. Several multiple precision algorithms were therefore developed:

— $\mathrm{multiplyAndAdd}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \delta)$ that computes $\boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C} + \boldsymbol{\Delta}$, where the error matrix $\boldsymbol{\Delta}$ is bounded by $|\boldsymbol{\Delta}| < \delta$, for the given *a priori* bound $\delta$. We shall notate $\boldsymbol{A} \otimes \boldsymbol{B}$ for the output of $\mathrm{multiplyAndAdd}$ when $\boldsymbol{C}$ is the zero matrix.

— $\mathrm{sumAbs}(\boldsymbol{A}, \boldsymbol{B}, \delta)$ that computes $\boldsymbol{A} + |\boldsymbol{B}| + \boldsymbol{\Delta}$, where the error matrix $\boldsymbol{\Delta}$ is bounded by $|\boldsymbol{\Delta}| < \delta$, for the given $\delta$. With a slight notational abuse, we shall also notate $\boldsymbol{A} \oplus \mathrm{abs}(\boldsymbol{B})$ for $\mathrm{sumAbs}$.

```
   Input: $A \in \mathbb{F}^{n\times n}, B \in \mathbb{F}^{n\times q}, C \in \mathbb{F}^{p\times n}, \varepsilon_1 > 0$
   Output: $N \in \mathbb{N}$
1  $\alpha, V, \varepsilon_\alpha, \varepsilon_V \leftarrow$ LAPACK eigendecomposition for $A$;
2  $\Omega, \varepsilon_\Psi \leftarrow$ LAPACK solver for $V\Psi = B$;
3  $[\lambda], [X] \leftarrow$ Eigensystem verification algorithm;
4  $[\Psi] \leftarrow$ LSE solution verification algorithm;
5  $[\Phi] \leftarrow C[X]$;
6  $[R_l]_{i,j} \leftarrow [\Phi_{i,l}][\Psi_{l,j}]$ ;
7  $[\rho] \leftarrow \max_i \left| [\lambda_i] \right|$;
8  $[M] \leftarrow \sum_{i=1}^{n} \frac{\left|[R_i]\right|}{1-\left|[\lambda_i]\right|} \frac{\left|[\lambda_i]\right|}{[\rho]}$ ;
9  $[m] \leftarrow \min_{i,j} \left| [M]_{i,j} \right|$;
10 $N \leftarrow \sup \left( \left\lceil \frac{\log \frac{\varepsilon_1}{[m]}}{\log [\rho]} \right\rceil \right)$;
11 return $N$
```

**Algorithm 10:** Lower bound of truncation order

— $\text{inv}(V, \delta)$ that computes the inverse $V^{-1} + \Delta$, where the error matrix $\Delta$ is bounded by $|\Delta| < \delta$, for the given $\delta$. See Section 5.2.6.

These computation kernels adapt the precision of their intermediate computations where needed. The algorithms we use for these basic bricks will be discussed in Section 5.2.6.

### Step 2: using the Eigendecomposition

**Error propagation** As seen, in each step of the summation, a matrix power, $A^k$, must be computed. In [111] Higham devotes an entire chapter to error analysis of matrix powers but this theory is in most cases inapplicable for state matrices $A$ of linear filters, as the requirement $\rho(|A|) < 1$ does not necessarily hold here. Therefore, despite taking $A$ to just a finite power $k$, the sequence of computed matrices may explode in norm since $k$ may take an order of several hundreds or thousands. Thus, even extending the precision is not a solution, as an enormous number of bits would be required.

However, the state matrices $A$ usually have a good structure. In real life the state matrices are diagonalizable, i.e. there exists a matrix $X \in \mathbb{C}^{n\times n}$ and diagonal $E \in \mathbb{C}^{n\times n}$ such that $A = XEX^{-1}$. Then $A^k = XE^kX^{-1}$. A good choice of $X$ and $E$ are the eigenvector and eigenvalue matrices obtained with eigendecomposition (5.18). However, with LAPACK we can compute only approximations on them and we cannot control their accuracy. Therefore, we propose following method to *almost* diagonalize matrix $A$. The method does not make any assumptions on matrix $V$ except for it being *some* approximation on $X$ . Therefore, for simplicity of further reasoning we treat $V$ as an exact matrix.

Using our multiprecision algorithms for matrix inverse and multiplication we may compute a complex $n \times n$ matrix $T$:

$$T := V^{-1}AV - \Delta_2, \tag{5.27}$$

where $\boldsymbol{V} \in \mathbb{C}^{n\times n}$ is an approximation on $\boldsymbol{X}$, $\boldsymbol{\Delta}_2 \in \mathbb{C}^{n\times n}$ is a matrix representing the element-by-element errors due to the two matrix multiplications and the inversion of matrix $\boldsymbol{V}$.

Although the matrix $\boldsymbol{E}$ is strictly diagonal, $\boldsymbol{V}$ is not exactly the eigenvector matrix and consequently $\boldsymbol{T}$ is a full matrix. However it has prevailing elements on the main diagonal. Thus $\boldsymbol{T}$ is an approximation on $\boldsymbol{E}$.

We require for matrix $\boldsymbol{T}$ to satisfy $\|\boldsymbol{T}\|_2 \leqslant 1$. This condition is stronger than $\rho(\boldsymbol{A}) < 1$, and Section 5.2.5 provides a way to test it. Naturally this condition means that there exist some margin for computational errors between the spectral radius and 1.

Notate $\Xi_k := (\boldsymbol{T} + \boldsymbol{\Delta}_2)^k - \boldsymbol{T}^k$. Hence $\Xi_k \in \mathbb{C}^{n\times n}$ represents an error matrix which captures the propagation of error $\boldsymbol{\Delta}_2$ when powering $\boldsymbol{T}$. Since

$$\boldsymbol{A}^k = \boldsymbol{V}(\boldsymbol{T} + \boldsymbol{\Delta}_2)^k \boldsymbol{V}^{-1}, \tag{5.28}$$

therefore

$$\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B} = \boldsymbol{C}\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}\boldsymbol{B} + \boldsymbol{C}\boldsymbol{V}\Xi_k\boldsymbol{V}^{-1}\boldsymbol{B}. \tag{5.29}$$

Thus the error of computing $\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}$ instead of $\boldsymbol{A}^k$ in (5.10) is bounded by

$$\left| \sum_{k=0}^{N} \left|\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B}\right| - \sum_{k=0}^{N} \left|\boldsymbol{C}\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}\boldsymbol{B}\right| \right| \leqslant \tag{5.30}$$

$$\sum_{k=0}^{N} \left|\boldsymbol{C}\boldsymbol{A}^k\boldsymbol{B} - \boldsymbol{C}\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}\boldsymbol{B}\right| \leqslant \sum_{k=0}^{N} \left|\boldsymbol{C}\boldsymbol{V}\Xi_k\boldsymbol{V}^{-1}\boldsymbol{B}\right| \tag{5.31}$$

Here and further on each step of the algorithm we use inequalities with left side in form (5.31) rather than (5.30), i.e. we will instantly use the triangulation property $\big|\,|a|-|b|\,\big| \leqslant |a-b|$ $\forall a, b$ applied element-by-element to matrices.

In order to determine the accuracy of the computations on this step such that (5.31) is bounded by $\varepsilon_2$, we need to perform detailed analysis of $\Xi_k$, with spectral-norm. Using the definition of $\Xi_k$ the following recurrence can be easily obtained:

$$\|\Xi_k\|_2 \leqslant \|\Xi_{k-1}\|_2 + \|\boldsymbol{\Delta}_2\|_2 \left(\|\Xi_{k-1}\|_2 + 1\right) \tag{5.32}$$

If $\|\Xi_{k-1}\|_2 \leqslant 1$, which must hold in our case since $\Xi_k$ represent an error-matrix, then

$$\|\Xi_k\|_2 \leqslant \|\Xi_{k-1}\|_2 + 2\,\|\boldsymbol{\Delta}_2\|_2\,. \tag{5.33}$$

As $\|\Xi_1\|_2 = \|\boldsymbol{\Delta}_2\|_2$ we can get the desired bound capturing the propagation of $\boldsymbol{\Delta}_2$ with Frobenius norm:

$$\|\Xi_k\|_F \leqslant 2\sqrt{n}(k+1)\,\|\boldsymbol{\Delta}_2\|_F\,. \tag{5.34}$$

Substituting this bound to (5.31) and folding the sum, we obtain

$$\sum_{i=0}^{N} \left|\boldsymbol{C}\boldsymbol{V}\Xi_k\boldsymbol{V}^{-1}\boldsymbol{B}\right| \leqslant \beta\,\|\boldsymbol{\Delta}_2\|_F\,\|\boldsymbol{C}\boldsymbol{V}\|_F\,\left\|\boldsymbol{V}^{-1}\boldsymbol{B}\right\|_F, \tag{5.35}$$

with $\beta = \sqrt{n}(N+1)(N+2)$. Thus, we get a bound on the error of approximation of $\boldsymbol{A}$ by $\boldsymbol{V}\boldsymbol{T}\boldsymbol{V}^{-1}$. Since we require it to be less than $\varepsilon_2$ we obtain a condition for the error on the inversion and two matrix multiplications:

$$\|\boldsymbol{\Delta}_2\|_F \leqslant \frac{1}{\beta}\frac{\varepsilon_2}{\|\boldsymbol{C}\boldsymbol{V}\|_F\,\|\boldsymbol{V}^{-1}\boldsymbol{B}\|_F}. \tag{5.36}$$

Using this bound we can deduce the desired accuracy of our multiprecision algorithms for complex matrix multiplication and inverse as a function of $\varepsilon_2$.

**Checking $\|\boldsymbol{T}\|_2 \leqslant 1$** Since $\|\boldsymbol{T}\|_2^2 = \rho(\boldsymbol{T}^*\boldsymbol{T})$, we study the eigenvalues of $\boldsymbol{T}^*\boldsymbol{T}$. According to Gershgorin's circle theorem [92], each eigenvalue $\mu_i$ of $\boldsymbol{T}^*\boldsymbol{T}$ is in the disk centered in $(\boldsymbol{T}^*\boldsymbol{T})_{ii}$ with radius $\sum_{j\neq i}\left|(\boldsymbol{T}^*\boldsymbol{T})_{ij}\right|$.

Let us decompose $\boldsymbol{T}$ into $\boldsymbol{T} = \boldsymbol{F} + \boldsymbol{G}$, where $\boldsymbol{F}$ is diagonal and $\boldsymbol{G}$ contains all the other terms ($\boldsymbol{F}$ contains the approximate eigenvalues, $\boldsymbol{G}$ contains small terms and is zero on its diagonal). Denote $\boldsymbol{Y} := \boldsymbol{T}^*\boldsymbol{T} - \boldsymbol{F}^*\boldsymbol{F} = \boldsymbol{F}^*\boldsymbol{G} + \boldsymbol{G}^*\boldsymbol{F} + \boldsymbol{G}^*\boldsymbol{G}$. Then

$$\begin{aligned}\sum_{j\neq i}\left|(\boldsymbol{T}^*\boldsymbol{T})_{ij}\right| &= \sum_{j\neq i}|\boldsymbol{Y}_{ij}| \\ &\leqslant (n-1)\,\|\boldsymbol{Y}\|_F \\ &\leqslant (n-1)\left(2\,\|\boldsymbol{F}\|_F\,\|\boldsymbol{G}\|_F + \|\boldsymbol{G}\|_F^2\right) \\ &\leqslant (n-1)\left(2\sqrt{n} + \|\boldsymbol{G}\|_F\right)\|\boldsymbol{G}\|_F\,. \end{aligned} \tag{5.37}$$

Each eigenvalue of $\boldsymbol{T}^*\boldsymbol{T}$ is in the disk centered in $(\boldsymbol{F}^*\boldsymbol{F})_{ii} + (\boldsymbol{Y})_{ii}$ with radius $\gamma$, where $\gamma$ is equal to $(n-1)\,(2\sqrt{n} + \|\boldsymbol{G}\|_F)\,\|\boldsymbol{G}\|_F$, computed in a rounding mode that makes the result become an upper bound (round-up).

As $\boldsymbol{G}$ is zero on its diagonal, the diagonal elements $(\boldsymbol{Y})_{ii}$ of $\boldsymbol{Y}$ are equal to the diagonal elements $(\boldsymbol{G}^*\boldsymbol{G})_{ii}$ of $\boldsymbol{G}^*\boldsymbol{G}$. They can hence be bounded as follows:

$$|(\boldsymbol{Y})_{ii}| = |(\boldsymbol{G}^*\boldsymbol{G})_{ii}| \leqslant \|\boldsymbol{G}\|_F^2\,. \tag{5.38}$$

Then, it is easy to see that the Gershgorin circles enclosing the eigenvalues of $\boldsymbol{F}^*\boldsymbol{F}$ can be increased, meaning that if $(\boldsymbol{F}^*\boldsymbol{F})_{ii}$ is such that

$$\forall i, \quad |(\boldsymbol{F}^*\boldsymbol{F})_{ii}| \leqslant 1 - \|\boldsymbol{G}\|_F^2 - \gamma, \tag{5.39}$$

it holds that $\rho(\boldsymbol{T}^*\boldsymbol{T}) \leqslant 1$ and $\|\boldsymbol{T}\|_2 \leqslant 1$.

This condition can be tested by using FP arithmetic with directed rounding modes (round-up, for instance).

After computing $\boldsymbol{T}$ out of $\boldsymbol{V}$ and $\boldsymbol{A}$ according to (5.27), the condition on $\boldsymbol{T}$ should be tested in order to determine if $\|\boldsymbol{T}\|_2 \leqslant 1$. This test failing means that $\boldsymbol{V}$ is not a sufficient approximate of $\boldsymbol{X}$ or that the error $\boldsymbol{\Delta}_2$ done computing (5.27) is too large, i.e. the accuracy of our multiprecision algorithm for complex matrix multiplication and inverse should be increased. The test is required for rigor only. We do perform the test in the implementation of our WCPG method, and, on the real-world examples we tested, never saw it fail.

#### Step 3: computing $\boldsymbol{CV}$ and $\boldsymbol{V}^{-1}\boldsymbol{B}$

We compute approximations on matrices $\boldsymbol{CV}$ and $\boldsymbol{V}^{-1}\boldsymbol{B}$ with a certain precision and need to determine the required accuracy of these multiplications such that the impact of these approximations is less than $\varepsilon_3$.

Notate $\boldsymbol{C'} := \boldsymbol{CV} + \boldsymbol{\Delta}_{3_C}$ and $\boldsymbol{B'} := \boldsymbol{V}^{-1}\boldsymbol{B} + \boldsymbol{\Delta}_{3_B}$, where $\boldsymbol{\Delta}_{3_C} \in \mathbb{C}^{p\times n}$ and $\boldsymbol{\Delta}_{3_B} \in \mathbb{C}^{n\times q}$ are error-matrices containing the errors of the two matrix multiplications and the inversion.

Using Frobenius norm, we can bound the error in the approximation of $\boldsymbol{CV}$ and $\boldsymbol{V}^{-1}\boldsymbol{B}$ by $\boldsymbol{C'}$ and $\boldsymbol{B'}$ as follows:

$$\sum_{k=0}^{N} \left|\boldsymbol{C}\boldsymbol{V}\boldsymbol{T}^k\boldsymbol{V}^{-1}\boldsymbol{B} - \boldsymbol{C'}\boldsymbol{T}^k\boldsymbol{B'}\right| \leqslant$$
$$\sum_{k=0}^{N} \left\|\boldsymbol{\Delta}_{3_C}\boldsymbol{T}^k\boldsymbol{B'} + \boldsymbol{C'}\boldsymbol{T}^k\boldsymbol{\Delta}_{3_B} + \boldsymbol{\Delta}_{3_C}\boldsymbol{T}^k\boldsymbol{\Delta}_{3_B}\right\|_F. \tag{5.40}$$

Since $\|\boldsymbol{T}\|_2 < 1$ holds we have (using Frobenius norm properties)

$$\left\|\boldsymbol{\Delta}_{3_C}\boldsymbol{T}^k\boldsymbol{B'} + \boldsymbol{C'}\boldsymbol{T}^k\boldsymbol{\Delta}_{3_B} + \boldsymbol{\Delta}_{3_C}\boldsymbol{T}^k\boldsymbol{\Delta}_{3_B}\right\|_F \leqslant \tag{5.41}$$
$$\sqrt{n}\left(\|\boldsymbol{\Delta}_{3_C}\|_F\left(\|\boldsymbol{B'}\|_F + \|\boldsymbol{\Delta}_{3_B}\|_F\right) + \|\boldsymbol{C'}\|_F\|\boldsymbol{\Delta}_{3_B}\|_F\right).$$

This bound represents the impact of our approximations for each $k = 0 \ldots N$. If (5.41) is bounded by $\frac{1}{N+1} \cdot \varepsilon_3$, then the overall error is less than $\varepsilon_3$. Hence, bounds on the two error-matrices are:

$$\|\boldsymbol{\Delta}_{3_C}\|_F \leqslant \frac{1}{3\sqrt{n}} \cdot \frac{1}{N+1}\frac{\varepsilon_3}{\|\boldsymbol{C'}\|_F} \tag{5.42}$$

$$\|\boldsymbol{\Delta}_{3_B}\|_F \leqslant \frac{1}{3\sqrt{n}} \cdot \frac{1}{N+1}\frac{\varepsilon_3}{\|\boldsymbol{B'}\|_F}. \tag{5.43}$$

Therefore, using bounds on $\|\boldsymbol{\Delta}_{3_C}\|_F$ and $\|\boldsymbol{\Delta}_{3_B}\|_F$, we can deduce the required accuracy of our multiprecision matrix multiplication and inversion according to $\varepsilon_3$.

**Step 4: powering $\boldsymbol{T}$**

Given a square complex matrix $\boldsymbol{T}$ with prevailing main diagonal, we need to compute its $k^{\text{th}}$ power. Notate

$$\boldsymbol{P}_k := \boldsymbol{T}^k - \boldsymbol{\Pi}_k, \tag{5.44}$$

where $\boldsymbol{\Pi}_k \in \mathbb{C}^{n\times n}$ represents element-by-element the error on the matrix powers, including error propagation from the first to the last power. Using the same simplification as in (5.30) and (5.31) we get the error of computing the approximations $\boldsymbol{P}_k$ rather than the exact powers bounded by

$$\sum_{k=0}^{N} \left|\boldsymbol{C'}\boldsymbol{T}^k\boldsymbol{B'} - \boldsymbol{C'}\boldsymbol{P}_k\boldsymbol{B'}\right| \leqslant \sum_{k=0}^{N} \left|\boldsymbol{C'}\boldsymbol{\Pi}_k\boldsymbol{B'}\right|. \tag{5.45}$$

Thus a bound on a norm of $\boldsymbol{\Pi}_k$, say $\|\boldsymbol{\Pi}_k\|_F$, is required.

Since we need all the powers of $\boldsymbol{T}$ from $1$ to $N$, we use an iterative scheme to compute them. It is then evident, that we may write the recurrence

$$\boldsymbol{P}_k = \boldsymbol{T}\boldsymbol{P}_{k-1} + \boldsymbol{\Gamma}_k, \tag{5.46}$$

where $\boldsymbol{\Gamma}_k \in \mathbb{C}^{n\times n}$ is the error matrix representing the error of the matrix multiplication at step $k$.

With $\boldsymbol{P}_0 = \boldsymbol{I}$, $\boldsymbol{P}_1 = \boldsymbol{T}$ and using (5.46) we obtain

$$\boldsymbol{P}_k = \boldsymbol{T}^k + \sum_{l=2}^{k} \boldsymbol{T}^{k-l}\boldsymbol{\Gamma}_l. \tag{5.47}$$

Using the condition $\|\boldsymbol{T}\|_2 \leqslant 1$ and properties of the Frobenius norm we get

$$\|\boldsymbol{\Pi}_k\|_F \leqslant \left\| \sum_{l=2}^{k} \boldsymbol{T}^{k-l}\boldsymbol{\Gamma}_l \right\|_F \leqslant \sqrt{n} \sum_{l=2}^{k} \|\boldsymbol{\Gamma}_l\|_F\,. \tag{5.48}$$

Therefore the impact of approximation of the matrix powers is bounded by

$$\sum_{k=0}^{N} |\boldsymbol{C}'\boldsymbol{\Pi}_k\boldsymbol{B}'| \leqslant \sqrt{n}(N+1) \sum_{l=2}^{N} \|\boldsymbol{C}'\|_F \|\boldsymbol{\Gamma}_l\|_F \|\boldsymbol{B}'\|_F\,. \tag{5.49}$$

Obviously, if the error of matrix multiplication $\boldsymbol{\Gamma}_l$ satisfies

$$\|\boldsymbol{\Gamma}_l\|_F \leqslant \frac{1}{\sqrt{n}} \cdot \frac{1}{N-1} \cdot \frac{1}{N+1} \cdot \frac{\varepsilon_4}{\|\boldsymbol{C}'\|_F \|\boldsymbol{B}'\|_F} \tag{5.50}$$

for $l = 2 \ldots N$, then we have (5.49) to be less than $\varepsilon_4$. Hence using (5.50) we may deduce the required accuracy of matrix multiplications on each step in dependency of $\varepsilon_4$.

**Step 5: computing $\boldsymbol{L}_k$**

Once the matrices $\boldsymbol{C}'$, $\boldsymbol{B}'$ and $\boldsymbol{P}_k$ are precomputed and the error of their computation is bounded, we must evaluate their product. Let $\boldsymbol{L}_k$ be the approximate product of these three matrices at step $k$:

$$\boldsymbol{L}_k := \boldsymbol{C}'\boldsymbol{P}_k\boldsymbol{B}' + \boldsymbol{\Upsilon}_k, \tag{5.51}$$

where $\boldsymbol{\Upsilon}_k \in \mathbb{C}^{p\times q}$ is the matrix of element-by-element errors for the two matrix multiplications. Then it may be shown, that the error of computations induced by this step is bounded by $\sum\limits_{k=0}^{N} |\boldsymbol{\Upsilon}_k|$.

If we want the overall error of approximation on this step to be less than $\varepsilon_5$ then we can easily deduce the required accuracy of each of those multiplications on every iteration of summation algorithm.

**Step 6: final summation**

Finally the absolute value of the $\boldsymbol{L}_k$ must be taken and the result accumulated in the sum. We remind the reader that if all previous computations were exact, the matrix $\boldsymbol{L}_k$ would be a real matrix and the absolute-value-operation would have been an exact sign manipulation. However, as the computations were in finite-precision arithmetic, $\boldsymbol{L}_k$ is complex with a small imaginary part, which is naturally caused by the errors of computations and must not be neglected. Therefore the element-by-element absolute value of the matrix must be computed.

Since we perform $N+1$ accumulations of absolute values in the result sum $\boldsymbol{S}_N$, it is evident that bounding the error of each such computation by $\frac{1}{N+1}\varepsilon_6$ is sufficient.

Therefore, using this bound for each invocation of our basic brick algorithm sumAbs we guarantee bound (5.16).

### 5.2.6 Adapting Precision For Linear Algebra Basic Bricks

In Section 5.2.5, we postulated the existence of three basic FP algorithms, $\mathrm{multiplyAndAdd}$, $\mathrm{sumAbs}$ and $\mathrm{inv}$, computing, respectively, the product-sum, the sum in absolute value and the inverse of matrices. Each of these operators was required to satisfy an absolute error bound $|\boldsymbol{\Delta}| < \delta$ to be ensured by the matrix of errors $\boldsymbol{\Delta}$ with respect to scalar $\delta$, given in argument to the algorithm.

Ensuring such an *absolute* error bound is not possible in general when fixed-precision FP arithmetic is used. Any algorithm of this kind, when returning its result, must round into that fixed-precision FP format. Hence, when the output grows sufficiently large, the unit-in-the-last-place of that format and hence that final rounding error in fixed-precision FP arithmetic will grow larger than a set absolute error bound.

In multiple precision FP arithmetic, such as offered by software packages like MPFR[2] [86], it is however possible to have algorithms determine themselves the output precision of the FP variables they return their results in. Hence an absolute error bound as the one we require can be guaranteed. In contrast to classical FP arithmetic, such as Higham analyzes, there is no longer any clear, overall *computing precision*, though. Variables just bear the precision that had been determined for them by the previous compute step.

This preliminary clarification made, description of our three basic bricks $\mathrm{multiplyAndAdd}$, $\mathrm{sumAbs}$ and $\mathrm{inv}$ is easy.

For $\mathrm{sumAbs}(\boldsymbol{A}, \boldsymbol{B}, \delta) = \boldsymbol{A} + |\boldsymbol{B}| + \boldsymbol{\Delta}$, we can reason element by element. We need to approximate $\boldsymbol{A}_{ij} + \sqrt{\Re \boldsymbol{B}_{ij}{}^2 + \Im \boldsymbol{B}_{ij}{}^2}$ with absolute error no larger than $\delta$, where $\Re z$ and $\Im z$ are the real and imaginary parts of the complex $z$. This can be ensured by considering the FP exponents of each of $\boldsymbol{A}_{ij}$, $\Re \boldsymbol{B}_{ij}$ and $\Im \boldsymbol{B}_{ij}$ with respect to the FP exponent of $\delta$.

For $\mathrm{multiplyAndAdd}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \delta) = \boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C} + \boldsymbol{\Delta}$, we can reason in terms of scalar products between $\boldsymbol{A}$ and $\boldsymbol{B}$. The scalar products boil down to summation of products which, in turn, can be done exactly, as we can determine the precision of the $\boldsymbol{A}_{ik}$ and $\boldsymbol{B}_{kj}$. As a matter of course the very same summation can capture the matrix elements $\boldsymbol{C}_{ij}$. Finally, multiple precision FP summation with an absolute error bound can be performed with a modified, software-simulated Kulisch accumulator [141], which does not need to be exact but bear just enough precision to satisfy the absolute accuracy bound $\delta$.

Finally, once the $\mathrm{multiplyAndAdd}$ operator is available, it is possible to implement the matrix inversion algorithm $\mathrm{inv}$ using a Newton-Raphson-like iteration [204]:

$$\begin{aligned} \boldsymbol{U}_0 &\leftarrow \text{some seed inverse matrix for } \boldsymbol{V}^{-1} \\ \boldsymbol{R}_k &\leftarrow \boldsymbol{V}\boldsymbol{U}_k - \boldsymbol{I}_n \\ \boldsymbol{U}_{k+1} &\leftarrow \boldsymbol{U}_k - \boldsymbol{U}_k \boldsymbol{R} \end{aligned} \tag{5.52}$$

where the iterated matrices $\boldsymbol{U}_k$ converge to $\boldsymbol{V}^{-1}$ provided the $\mathrm{multiplyAndAdd}$ operations computing $\boldsymbol{R}_k$ and $\boldsymbol{U}_{k+1}$ are performed with enough accuracy, i.e. small enough $\delta$ and the seed matrix satisfies some additional properties. In order to ensure these properties with an explicit check, an operator to compute the Frobenius norm of a matrix with a given *a priori* absolute error bound $\delta$ is required. Implementing such a Frobenius norm operator again boils down to summation, as shown above.

2. `http://www.mpfr.org/`

| | Example 1 | | | Example 2 | | | Example 3 | | | Example 4 |
|---|---|---|---|---|---|---|---|---|---|---|
| $n$ | 10 | | | 12 | | | 60 | | | 3 |
| $p$ | 11 | | | 1 | | | 28 | | | 3 |
| $q$ | 1 | | | 25 | | | 4 | | | 3 |
| $1-\rho(\boldsymbol{A})$ | $1.39\times10^{-2}$ | | | $8.65\times10^{-3}$ | | | $1.46\times10^{-2}$ | | | $2^{-60}$ |
| $\max(\boldsymbol{S}_N)$ | $3.88\times10^{1}$ | | | $5.50\times10^{9}$ | | | $2.64\times10^{2}$ | | | - |
| $\min(\boldsymbol{S}_N)$ | $1.29\times10^{0}$ | | | $1.0\times10^{0}$ | | | $1.82\times10^{1}$ | | | - |
| $\varepsilon$ | $2^{-5}$ | $2^{-53}$ | $2^{-600}$ | $2^{-5}$ | $2^{-53}$ | $2^{-600}$ | $2^{-5}$ | $2^{-53}$ | $2^{-600}$ | $2^{-5}$ |
| $N$ | 220 | 2153 | 29182 | 308 | 4141 | 47811 | 510 | 1749 | 27485 | - |
| Inversion iterations | 0 | 2 | 4 | 2 | 3 | 5 | 1 | 2 | 4 | - |
| overall max precision (bits) | 212 | 293 | 1401 | 254 | 355 | 1459 | 232 | 306 | 1416 | - |
| Overall execution time (sec) | 0.11 | 1.53 | 60.06 | 0.85 | 11.54 | 473.20 | 45.62 | 177.90 | 9376.86 | 0.00... |

Table 5.1 – Numerical results for 3 real-world and 1 constructed example

### 5.2.7 Numerical Results

The algorithms discussed above were implemented in `C`, using GNU MPFR version 3.1.12, GNU MPFI[3] version 1.5.1 and CLAPACK[4] version 3.2.1. Our implementation was tested on several real-life and random examples:

— The first example comes from Control Theory: the LTI system is extracted from an active controller of vehicle longitudinal oscillation [169], and WCPG matrix is used to determine the fixed-point arithmetic scaling of state and output.

— The second is a 12th-order Butterworth filter, described in $\rho$-Direct Form II transposed [261] (a particular algorithm, with low complexity and high robustness to quantization and computational errors), where the errors-to-output LTI system $\mathcal{H}_e$ is considered (see Figure 5.1).

— The third one is a large random BIBO stable filter (obtained from the `drss` command of Matlab), with 60 states, 14 inputs and 28 outputs.

— The last one is built with a companion matrix $\boldsymbol{A}$ with spectral radius equal to $1-2^{-60}$.

Experiments were done on a laptop computer with an Intel Core i5 processor running at 2.8 GHz and 16 GB of RAM.

The numerical results detailed in Table 5.1 show that our algorithm for Worst-Case Peak Gain matrix evaluation with *a priori* error bound exhibits reasonable performance on typical examples. Even when the *a priori* error bound is pushed to compute WCPG results with an accuracy way beyond double precision, the algorithm succeeds in computing a result, even though execution time grows pretty high.

Our algorithm includes checks testing that certain properties of matrices are verified, in particular that $\rho(\boldsymbol{A}) < 1$ and $\|\boldsymbol{T}\|_2 \leqslant 1$. Our Example $4$, not taken from a real-world application but constructed on purpose, shows that the algorithm correctly detects that the conditions are not fulfilled for that example.

### 5.2.8 Conclusion

With this work, a reliable, rigorous multiprecision method to compute the Worst-Case Peak Gain matrix has been developed. It relies on Theory of Verified Inclusion, eigenvalue

3. `https://gforge.inria.fr/projects/mpfi/`
4. `http://www.netlib.org/clapack/`

decomposition to perform matrix powering, some multiple-precision arithmetic basic bricks developed to satisfy absolute error bounds and a detailed step-by-step error analysis.

A C program has been developed and now can be used as a tool for the implementation error analysis of LTI systems, and then the design of reliable finite precision digital algorithms for signal processing and control.

However, some efforts are still required to overcome double precision eigenvalue decomposition in LAPACK (specially for close-to-instability LTI systems) by using a multiple precision eigensolver. Additionally, as the proofs on the error bounds are pretty complicated, they should be formalized in a Formal Proof Checker, such as Coq or HolLight.

## 5.3 Exploiting Accuracy Adaptation

Having solved the problem of computing the WCPG gain of an LTI filter in the previous Section 5.2, we can now engage into solving a problem upcoming with the generation of code for an LTI filter: how large the different intermediate variables in a filter need to be and what the weight of their MSB and LSB position in a fixed-point environment is. This problem seems to be of completely different nature at first sight, but it actually is not: the problem still boils down to determine the compute precision of a numerical algorithm –for instance the filter code– in order to meet a certain error bound – this time on the maximum noise introduced by the code. The fact that we speak of MSB and LSB positions and not of compute precision is merely due to the nomenclature of fixed-point arithmetic.

This Section is based on an article we published in [251].

### 5.3.1 Introduction

The implementation of a digital filter can be considered as a path from filter specifications to a physical implementation (DSP device). This process includes generation of a transfer function, which defines the relation between the inputs and outputs of a filter. Further one needs to choose a filter structure to be eventually implemented, i.e. a computational scheme, a specific order of filter evaluation. Finally, there is software and hardware code generation.

However, on each step various issues arise. First of all, every transfer function must be discretized, which implies the modification of filter. Secondly, while in infinite precision filter structures (e.g. Direct Forms [203], State-Space [88], Wave [80] etc.) are equivalent, they are no longer the same objects in finite precision. Since the computational schemes are different, the round-off errors and consequently quality of structures vary. Finally, software and hardware implementations are performed under constraints, e.g. power consumption, area, output error, etc. And on each step of the filter implementation the degradation due to finite precision must be taken into account in order to produce a reliable implementation.

To unify the above-described process of filter implementation for any Linear Time-Invariant (LTI) filter and provide reliable implementation, we develop an automatized filter generator. Its work-flow is described in the Figure 5.2.

In this Section we focus on one of the various steps: the Fixed-Point (FxP) Algorithm generation step and wordlength optimization. On this step we already know target specifications and software implementation constraints. Therefore, during this optimization, we naturally look for a trade-off between area (for hardware implementation) and computational errors. If we take more bits, the implementation cost arises. If we take less bits, there exist a risk of an

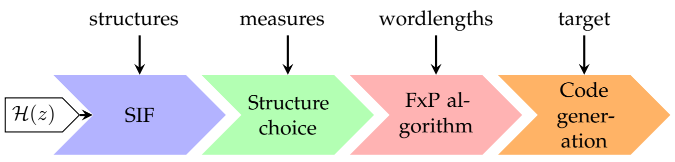

Figure 5.2 – Automatic filter generator flow.

overflow. Depending on the application, one or even both criteria can be neglected. However, in the general case on each step of the optimization procedure we need to determine *rigorously* the Fixed-Point Formats (FxPF), i.e. the most and the least significant bits positions, for all variables in the filter implementation. These formats must allow no overflow and take into account computational errors.

There exist various approaches on determining FxPF, such as Affine Arithmetic [181], or the common approach of numerous simulations [136]. However, these methods are not suitable for our needs since they do not give any mathematical guarantee on the absence of overflow for any possible input and tend to overestimate. We, on the other hand, propose a rigorous algorithm completely based on mathematical proofs.

In Section 5.3.2 we describe the fixed-point arithmetic in question and theoretical background for our approach. In Section 5.3.3 we define the problem for a filter in a state-space representation and introduce a two-step algorithm. Error analysis of the algorithm is presented in Section 5.3.4. Finally, numerical results are provided before conclusion.

**Notation:** Throughout this Section we continue using the notation introduced before: matrices are in uppercase boldface, vectors are in lowercase boldface, scalars are in lowercase. All matrix inequalities and absolute values are considered element-by-element. The matrix $\mathbb{1}$ denotes a matrix of ones with the appropriate size.

## 5.3.2 Basic bricks

### The Worst-Case Peak Gain theorem

Let $\mathcal{H} := (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ be a Bounded-Input Bounded-Output stable MIMO LTI filter in state-space representation:

$$\mathcal{H}\left\{\begin{array}{rcl} \boldsymbol{x}(k+1) & = & \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) & = & \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{array}\right. \tag{5.53}$$

where $\boldsymbol{u}(k) \in \mathbb{R}^{q\times 1}$ is the input vector, $\boldsymbol{y}(k) \in \mathbb{R}^{p\times 1}$ the output vector, $\boldsymbol{x}(k) \in \mathbb{R}^{n\times 1}$ the state vector and $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, $\boldsymbol{B} \in \mathbb{R}^{n\times q}$, $\boldsymbol{C} \in \mathbb{R}^{p\times n}$ and $\boldsymbol{D} \in \mathbb{R}^{p\times q}$ are the state-space matrices.

We will use the following approach on deducing the output interval for a filter: suppose all the inputs are guaranteed to be in a known interval:

$$\forall k, \quad |\boldsymbol{u}_i(k)| \leqslant \bar{\boldsymbol{u}}_i, \quad i = 1, \dots, q. \tag{5.54}$$

Then, there exists a $N_0$ such that $\forall k \geqslant N_0$ the output $\boldsymbol{y}(k)$ lies in the interval:

$$|\boldsymbol{y}(k)| \leqslant \langle\langle \mathcal{H} \rangle\rangle\, \bar{\boldsymbol{u}}, \tag{5.55}$$

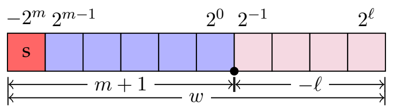

Figure 5.3 – Fixed-point representation (here, $m = 5$ and $\ell = -4$).

where $\langle\langle \mathcal{H} \rangle\rangle$ denotes the so-called Worst-Case Peak Gain (WCPG) matrix of the filter [113]. It can be computed as the $\ell_1$-norm of the impulse response of the filter. The $N_0$ can be determined using reasoning similar to the WCPG error analysis in [252]. There always exists an input sequence $\boldsymbol{u}(k)$ such that the equality in (5.55) is reached.

There exist numerous other approaches to determine the output interval, such as involving Affine Arithmetic [181], which may lead to overestimations, or simulations [136], which in turn are inefficient and not rigorous.

It can be shown that the WCPG approach gives the smallest interval containing all the possible values of $\boldsymbol{y}(k)$ and that the bound (5.55) can be attained. Moreover, in [252] the authors have published an algorithm to evaluate the WCPG at arbitrary precision.

### Fixed-Point arithmetic for filter implementation

The considered filters are implemented using signed FxP Arithmetic, with two's complement representation [199], [83]. Let $t$ be such a FxP number. It can be written as $t = -2^m t_m + \sum_{i=\ell}^{m-1} 2^i t_i$, where its FxPF ($m$, $\ell$) gives the position of the *most* (MSB) significant and *least* (LSB) significant bits respectively, and $t_i \in \mathbb{B} = \{0, 1\}$ is the $i^{\text{th}}$ bit of $t$ as shown in Figure 5.3. The wordlength $w$ is given by $w = m - \ell + 1$. The quantization step of the representation is $2^\ell$. Further in this Section the LSB position $\ell$ sometimes will be substituted directly with $m - w + 1$.

The real number $t$ is represented in machine by the integer $T$, such that $T = t \cdot 2^{-\ell}$, where $T \in [-2^{w-1}; 2^{w-1} - 1] \cap \mathbb{Z}$. Note that this interval is not symmetric.

Let $\boldsymbol{y}(k) \in \mathbb{R}$ be an output of a digital filter. Given the wordlength $\boldsymbol{w}_y$, determining the Fixed-Point Formats for $\boldsymbol{y}(k)$ means to find a MSB vector $\boldsymbol{m}_y$ such that for each $k$

$$\boldsymbol{y}_i(k) \in [-2^{\boldsymbol{m}_{y_i}}; 2^{\boldsymbol{m}_{y_i}} - 2^{\boldsymbol{m}_{y_i} - \boldsymbol{w}_{y_i} + 1}]. \tag{5.56}$$

Obviously, we are interested in the least possible MSB, which guarantees (5.56): any greater MSB would yield to a larger quantization step, hence round-off noise, at constant wordlength.

## 5.3.3 Determining the Fixed-Point Formats

### Exact filter

The problem of determining the FxPF for a filter $\mathcal{H}$ implementation can be formulated as follows. Let $\mathcal{H}$ be a filter in a state-space representation (5.53). Suppose all the inputs to be in an interval bounded by $\bar{\boldsymbol{u}}$.

Given the wordlength constraints vector $\boldsymbol{w}_x$ for the state and $\boldsymbol{w}_y$ for the output variables we look for a FxPF for $\boldsymbol{x}(k)$ and $\boldsymbol{y}(k)$ such that for any possible input no overflow occurs. We wish to determine the least MSB vectors $\boldsymbol{m}_y$ and $\boldsymbol{m}_x$ such that

$$\forall k, \quad \boldsymbol{y}(k) \in [-2^{-\boldsymbol{m}_y}; 2^{\boldsymbol{m}_y} - 2^{\boldsymbol{m}_y - \boldsymbol{w}_y + 1}], \tag{5.57}$$

$$\forall k, \quad \boldsymbol{x}(k) \in [-2^{-\boldsymbol{m}_x}; 2^{\boldsymbol{m}_x} - 2^{\boldsymbol{m}_x - \boldsymbol{w}_x + 1}]. \tag{5.58}$$

**Remark 4.** *Since the filter $\mathcal{H}$ is linear and input interval is centered at zero, the output interval is also centered in zero. Therefore, further we will seek to determine the least $\boldsymbol{m}_y$ and $\boldsymbol{m}_x$ such that*

$$\forall k, \quad |\boldsymbol{y}(k)| \leqslant 2^{\boldsymbol{m}_y} - 2^{\boldsymbol{m}_y - \boldsymbol{w}_y + 1}, \tag{5.59}$$

$$\forall k, \quad |\boldsymbol{x}(k)| \leqslant 2^{\boldsymbol{m}_x} - 2^{\boldsymbol{m}_x - \boldsymbol{w}_x + 1}. \tag{5.60}$$

#### Applying the WCPG to compute MSB positions

Applying the WCPG theorem on the filter $\mathcal{H}$ yields a bound on the output interval:

$$|\boldsymbol{y}_i(k)| \leqslant \left(\langle\langle\mathcal{H}\rangle\rangle\, \bar{\boldsymbol{u}}\right)_i, \quad i = 1, \dots, p. \tag{5.61}$$

Denote the bound vector $\bar{\boldsymbol{y}} := \langle\langle\mathcal{H}\rangle\rangle\, \bar{\boldsymbol{u}}$.

We can determine the FxPF for the output of a LTI filter $\mathcal{H}$ utilizing the following lemma.

**Lemma 4.** *Let $\mathcal{H} = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ be a BIBO-stable MIMO LTI filter and $\bar{\boldsymbol{u}}$ be a bound on the input interval. Suppose the wordlengths $\boldsymbol{w}_y$ are known and $\boldsymbol{w}_{y_i} > 1$, $i = 1, \dots, p$.*

*If for $i = 1, \dots, p$ the MSBs are computed with*

$$\boldsymbol{m}_{y_i} = \left\lceil \log_2(\bar{\boldsymbol{y}}_i) - \log_2\left(1 - 2^{1-\boldsymbol{w}_{y_i}}\right)\right\rceil \tag{5.62}$$

*and the LSBs are computed with $\boldsymbol{\ell}_{y_i} = \boldsymbol{m}_{y_i} + 1 - \boldsymbol{w}_{y_i}$, then for all $k$ $|\boldsymbol{y}_i(k)| \leqslant 2^{\boldsymbol{m}_{y_i}} - 2^{\boldsymbol{m}_{y_i} - \boldsymbol{w}_{y_i} + 1}$ and $\boldsymbol{m}_{y_i}$ is the least.*

*Proof.* We look for the least $\boldsymbol{m}_y$ such that (5.59) holds. Since the bound $\bar{\boldsymbol{y}}$ can be reached, it is sufficient to require:

$$\bar{\boldsymbol{y}}_i \leqslant 2^{\boldsymbol{m}_{y_i}} - 2^{\boldsymbol{m}_{y_i} - \boldsymbol{w}_{y_i} + 1}. \tag{5.63}$$

Solving this inequality for $\boldsymbol{m}_{y_i}$ we obtain that the smallest integer, which satisfies the above inequality is

$$\boldsymbol{m}_{y_i} = \left\lceil \log_2(\bar{\boldsymbol{y}}_i) - \log_2\left(1 - 2^{1-\boldsymbol{w}_{y_i}}\right)\right\rceil. \tag{5.64}$$

□

#### Modification of filter $\mathcal{H}$ to determine bounds on the state variable x

Using Lemma 4 we can determine the FxPF for the output of a filter. In order to determine the FxPF for the state variable, we modify the filter $\mathcal{H}$ by vertically concatenating the state vector and the output and include necessary changes into the state matrices.

Denote vector $\boldsymbol{\zeta}(k) := \begin{pmatrix} \boldsymbol{x}(k) \\ \boldsymbol{y}(k) \end{pmatrix}$ to be the new output vector. Then the state-space relationship (5.53) takes the form:

$$\mathcal{H}_\zeta \left\{ \begin{array}{rcccc} \boldsymbol{x}(k+1) = & \boldsymbol{A}\boldsymbol{x}(k) & + & \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{\zeta}(k) = & \begin{pmatrix} \boldsymbol{I} \\ \boldsymbol{C} \end{pmatrix} \boldsymbol{x}(k) & + & \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{D} \end{pmatrix} \boldsymbol{u}(k) \end{array} \right. . \tag{5.65}$$

Hence the problem is to find the least MSB vector $\boldsymbol{m}_\zeta$ such that (element-by-element)

$$\forall k, \quad |\boldsymbol{\zeta}(k)| \leqslant 2^{\boldsymbol{m}_\zeta} - 2^{\boldsymbol{m}_\zeta - \boldsymbol{w}_\zeta + 1}. \tag{5.66}$$

Now, applying the WCPG theorem on the filter $\mathcal{H}_\zeta$ and using Lemma 4, we can deduce the MSB positions of the state and output vectors for an implementation of the filter $\mathcal{H}$.

**Taking rounding errors into account**

However, due to the finite-precision degradation what we actually compute is not the exact filter $\mathcal{H}_\zeta$ but an implemented filter $\mathcal{H}_\zeta^\Diamond$:

$$\mathcal{H}_\zeta^\Diamond \left\{ \begin{array}{rcl} \boldsymbol{x}^\Diamond(k+1) & = & \Diamond_{\boldsymbol{\ell}_x} \left( \boldsymbol{A}\boldsymbol{x}^\Diamond(k) + \boldsymbol{B}\boldsymbol{u}(k) \right) \\ \boldsymbol{\zeta}^\Diamond(k) & = & \Diamond_{\boldsymbol{\ell}_\zeta} \left( \begin{pmatrix} \boldsymbol{I} \\ \boldsymbol{C} \end{pmatrix} \boldsymbol{x}^\Diamond(k) + \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{D} \end{pmatrix} \boldsymbol{u}(k) \right) \end{array} \right. \tag{5.67}$$

where the Sums-of-Products (accumulation of scalar products on the right side) are computed with some rounding operator $\Diamond_\ell$. Suppose, this operator ensures faithful rounding [197]:

$$|\Diamond_\ell(x) - x| < 2^\ell \tag{5.68}$$

In [61, 180] it was shown that such an operator can be implemented using some extra guard bits for the accumulation.

Denote the errors due to operator $\Diamond_\ell$ as $\boldsymbol{\varepsilon}_x(k)$ and $\boldsymbol{\varepsilon}_y(k)$ for the state and output vectors, respectively. Essentially, the vectors $\boldsymbol{\varepsilon}_x(k)$ and $\boldsymbol{\varepsilon}_y(k)$ may be associated with the noise which is induced by the filter implementation. Then the implemented filter can be rewritten as

$$\mathcal{H}_\zeta^\Diamond \left\{ \begin{array}{rcl} \boldsymbol{x}^\Diamond(k+1) & = & \boldsymbol{A}\boldsymbol{x}^\Diamond(k) + \boldsymbol{B}\boldsymbol{u}(k) + \boldsymbol{\varepsilon}_x(k) \\ \boldsymbol{\zeta}^\Diamond(k) & = & \begin{pmatrix} \boldsymbol{I} \\ \boldsymbol{C} \end{pmatrix} \boldsymbol{x}^\Diamond(k) + \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{D} \end{pmatrix} \boldsymbol{u}(k) + \begin{pmatrix} \boldsymbol{0} \\ \boldsymbol{I} \end{pmatrix} \boldsymbol{\varepsilon}_y(k) \end{array} \right. , \tag{5.69}$$

where

$$|\boldsymbol{\varepsilon}_x(k)| < 2^{\boldsymbol{\ell}_x}, \quad |\boldsymbol{\varepsilon}_y(k)| < 2^{\boldsymbol{\ell}_y}.$$

It should be remarked that since the operator $\Diamond_l$ is applied $\boldsymbol{\varepsilon}_x(k) \neq \boldsymbol{x}(k) - \boldsymbol{x}^\Diamond(k)$ and $\boldsymbol{\varepsilon}_y(k) \neq \boldsymbol{y}(k) - \boldsymbol{y}^\Diamond(k)$. As the rounding also affects the filter state, the $\boldsymbol{x}^\Diamond(k)$ drifts away from $\boldsymbol{x}(k)$ over time, whereas with $\boldsymbol{\varepsilon}_x(k)$ we consider the error due to one step only.

It can be observed that at each instance of time the state and output vectors are computed out of $\boldsymbol{u}(k)$ and error-vectors, which can be considered as inputs as well. Thanks to the linearity of the filters, we can decompose the actually implemented filter into a sum of the exact filter and an "error-filter" $\mathcal{H}_\Delta$ as shown in Figure 5.4. Note that this "error-filter" is

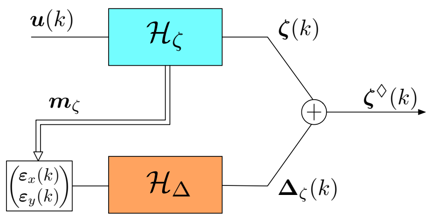

Figure 5.4 – Implemented filter decomposition.

an artificial one; it is not required to be "implemented" by itself and serves exclusively for error-analysis purposes.

The filter $\mathcal{H}_\Delta$ is obtained by computing the difference between $\mathcal{H}_\zeta^\diamondsuit$ and $\mathcal{H}_\zeta$. This filter takes the rounding errors $\boldsymbol{\varepsilon}(k) := \begin{pmatrix}\boldsymbol{\varepsilon}_x(k)\\ \boldsymbol{\varepsilon}_y(k)\end{pmatrix}$ as input and returns the result of their propagation through the filter:

$$\mathcal{H}_\Delta \left\{ \begin{array}{rclcl} \boldsymbol{\Delta}_x(k+1) & = & \boldsymbol{A}\boldsymbol{\Delta}_x(k) & + & \begin{pmatrix}\boldsymbol{I}\\ \boldsymbol{0}\end{pmatrix}\boldsymbol{\varepsilon}(k) \\ \boldsymbol{\Delta}_\zeta(k) & = & \begin{pmatrix}\boldsymbol{I}\\ \boldsymbol{C}\end{pmatrix}\boldsymbol{\Delta}_x(k) & + & \begin{pmatrix}\boldsymbol{0} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{I}\end{pmatrix}\boldsymbol{\varepsilon}(k) \end{array}\right. , \tag{5.70}$$

where, the vector $\boldsymbol{\varepsilon}(k)$ is guaranteed to be in the interval bounded by $\bar{\boldsymbol{\varepsilon}} := 2^{\boldsymbol{\ell}_\zeta}$.

Once the decomposition is done, we can apply the WCPG theorem on the "error-filter" $\mathcal{H}_\Delta$ and deduce the output interval of the computational errors propagated through filter:

$$\forall k, \quad |\boldsymbol{\Delta}_\zeta(k)| \leqslant \langle\langle \mathcal{H}_\Delta \rangle\rangle \cdot \bar{\boldsymbol{\varepsilon}}. \tag{5.71}$$

Hence, the output of the implemented filter is bounded with

$$\left|\boldsymbol{\zeta}^\diamondsuit(k)\right| \leqslant |\boldsymbol{\zeta}(k)| + |\boldsymbol{\Delta}_\zeta(k)|. \tag{5.72}$$

Applying Lemma 4 on the implemented filter and using (5.72) we obtain that the MSB vector $\boldsymbol{m}_\zeta^\diamondsuit$ can be upper bounded by

$$\begin{aligned} \boldsymbol{m}_{\zeta_i}^\diamondsuit = \Big\lceil & \log_2 \left( (\langle\langle \mathcal{H}_\zeta \rangle\rangle \cdot \bar{\boldsymbol{u}})_i + (\langle\langle \mathcal{H}_\Delta \rangle\rangle \cdot \bar{\boldsymbol{\varepsilon}})_i \right) \\ & - \log_2 \left(1 - 2^{1-\boldsymbol{w}_{\zeta_i}}\right) \Big\rceil . \end{aligned} \tag{5.73}$$

Therefore, the FxPF $(\boldsymbol{m}_\zeta^\diamondsuit, \boldsymbol{\ell}_\zeta^\diamondsuit)$ guarantee that no overflows occur for the implemented filter.

Since the input of the error filter $\mathcal{H}_\Delta$ depends on the FxPF chosen for implementation, we cannot directly use (5.73). The idea is to first compute the FxPF of the variables in the exact filter $\mathcal{H}$, where computational errors are not taken into account, and use it as an initial guess for implemented filter $\mathcal{H}_\zeta^\diamondsuit$. Hence, we obtain the following two-step algorithm:

Step 1: Determine the FxPF $(\boldsymbol{m}_\zeta, \boldsymbol{\ell}_\zeta)$ for the exact filter $\mathcal{H}_\zeta$

Step 2: Construct the "error-filter" $\mathcal{H}_\Delta$, which gives the propagation of the computational errors induced by format $(\boldsymbol{m}_\zeta, \boldsymbol{\ell}_\zeta)$; then, compute the FxPF $(\boldsymbol{m}_\zeta^\diamondsuit, \boldsymbol{\ell}_\zeta^\diamondsuit)$ of the actually implemented filter $\mathcal{H}_\zeta^\diamondsuit$ using (5.73).

The above algorithm takes into account the filter implementation errors. However, the algorithm itself is implemented in finite-precision and can suffer from rounding errors, which influence the output result. All operations in the MSB computation will induce errors, so what we actually compute are only floating-point approximations $\widehat{\boldsymbol{m}_\zeta}$ and $\widehat{\boldsymbol{m}_\zeta}^\Diamond$. In what follows, we propose error-analysis of the floating-point evaluation of the MSB positions via (5.62) and (5.73).

### 5.3.4 Error analysis of the MSB computation formula

Let us consider case of $\widehat{\boldsymbol{m}_{\zeta_i}}^\Diamond$ and show afterwards that $\widehat{\boldsymbol{m}_{\zeta_i}}$ is its special case. To reduce the size of expressions, denote

$$\mathfrak{m} := \log_2\left( \left(\langle\langle \mathcal{H}_\zeta \rangle\rangle \cdot \bar{\boldsymbol{u}}\right)_i + \left(\langle\langle \mathcal{H}_\Delta \rangle\rangle \cdot \bar{\boldsymbol{\varepsilon}}\right)_i \right) - \log_2\left(1 - 2^{1-\boldsymbol{w}_{\zeta_i}}\right).$$

Handling floating-point analysis of multiplications and additions in (5.73) is trivial using approach by Higham [111]. The difficulty comes from the WCPG matrices, which cannot be computed exactly. However both approximations $\widehat{\langle\langle \mathcal{H}_\zeta \rangle\rangle}$ and $\widehat{\langle\langle \mathcal{H}_\Delta \rangle\rangle}$, even if computed with arbitrary precision, bear some errors $\varepsilon_{WCPG_\zeta}$ and $\varepsilon_{WCPG_\Delta}$ that satisfy

$$0 \leqslant \widehat{\langle\langle \mathcal{H}_\Delta \rangle\rangle} - \langle\langle \mathcal{H}_\Delta \rangle\rangle \leqslant \varepsilon_{WCPG_\zeta} \cdot \mathbb{1}, \tag{5.74}$$

$$0 \leqslant \widehat{\langle\langle \mathcal{H}_\zeta \rangle\rangle} - \langle\langle \mathcal{H}_\zeta \rangle\rangle \leqslant \varepsilon_{WCPG_\Delta} \cdot \mathbb{1}. \tag{5.75}$$

Introducing the errors on the WCPG computations into the formula (5.73) we obtain that what we actually compute is

$$\widehat{\boldsymbol{m}_{\zeta_i}}^\Diamond \leqslant \left\lceil \mathfrak{m} + \log_2\left( 1 + \frac{\varepsilon_{WCPG_\zeta} \sum\limits_{j=1}^{q} \bar{\boldsymbol{u}}_j + \varepsilon_{WCPG_\Delta} \sum\limits_{j=1}^{n+p} \bar{\boldsymbol{\varepsilon}}_j}{\left(\langle\langle \mathcal{H}_\zeta \rangle\rangle\, \bar{\boldsymbol{u}}\right)_i + \left(\langle\langle \mathcal{H}_\Delta \rangle\rangle\, \bar{\boldsymbol{\varepsilon}}\right)_i} \right) \right\rceil. \tag{5.76}$$

The error term in (5.76) cannot be zero (apart from trivial case with zero $\bar{\boldsymbol{u}}$). However, since we can control the accuracy of the WCPG matrices, we can deduce conditions for the approximation $\widehat{\boldsymbol{m}_{\zeta_i}}^\Diamond$ to be off by at most one.

**Lemma 5.** *If the WCPG matrices* $\langle\langle \mathcal{H}_\zeta \rangle\rangle$ *and* $\langle\langle \mathcal{H}_\Delta \rangle\rangle$ *are computed such that* (5.74) *and* (5.75) *hold with*

$$\varepsilon_{WCPG_\Delta} < \frac{1}{2} \frac{\left(\underline{\langle\langle \mathcal{H}_\Delta \rangle\rangle} \cdot \bar{\boldsymbol{\varepsilon}}\right)_i}{\sum_{j=1}^{p+n} \bar{\boldsymbol{\varepsilon}}_i} \tag{5.77}$$

$$\varepsilon_{WCPG_\zeta} < \frac{1}{2} \frac{\left(\underline{\langle\langle \mathcal{H}_\zeta \rangle\rangle} \cdot \bar{\boldsymbol{u}}\right)_i}{\sum_{j=1}^{q} \bar{\boldsymbol{u}}_i}, \tag{5.78}$$

*where* $\underline{\langle\langle \mathcal{H} \rangle\rangle} := |\boldsymbol{D}| + |\boldsymbol{CB}| + |\boldsymbol{CAB}|$*, then*

$$0 \leqslant \widehat{\boldsymbol{m}_{\zeta_i}}^\Diamond - \boldsymbol{m}_{\zeta_i}^\Diamond \leqslant 1. \tag{5.79}$$

*Proof.* Proof by construction, we reason as follows: since the error-term caused by the WCPG floating-point evaluation is positive and the ceiling function is increasing, then

$$\widehat{\boldsymbol{m}_{\zeta_i}}^{\Diamond} - \boldsymbol{m}_{\zeta_i}{}^{\Diamond} \geqslant 0, \tag{5.80}$$

i.e. the floating-point approximation $\widehat{\boldsymbol{m}_{\zeta_i}}^{\Diamond}$ is guaranteed to never be underestimated. However, it can overestimate the MSB position by

$$\widehat{\boldsymbol{m}_{\zeta_i}}^{\Diamond} - \boldsymbol{m}_{\zeta_i}{}^{\Diamond} \leqslant \left\lceil \underbrace{\mathfrak{m} - \lceil \mathfrak{m} \rceil}_{-1 < \cdot \leqslant 0} + \log_2 \left( 1 + \frac{\varepsilon_{WCPG_\zeta} \sum_{j=1}^{q} \bar{\boldsymbol{u}}_j + \varepsilon_{WCPG_\Delta} \sum_{j=1}^{n+p} \bar{\boldsymbol{\varepsilon}}_j}{\left(\langle\langle \mathcal{H}_\zeta \rangle\rangle \, \bar{\boldsymbol{u}}\right)_i + \left(\langle\langle \mathcal{H}_\Delta \rangle\rangle \, \bar{\boldsymbol{\varepsilon}}\right)_i} \right) \right\rceil . \tag{5.81}$$

The approximation $\widehat{\boldsymbol{m}_{\zeta_i}}^{\Diamond}$ overestimates at most by one bit if and only if the error term is contained in the interval $[0, 1)$, i.e. if

$$0 \leqslant \log_2 \left( 1 + \frac{\varepsilon_{WCPG_\zeta} \sum_{j=1}^{q} \bar{\boldsymbol{u}}_j + \varepsilon_{WCPG_\Delta} \sum_{j=1}^{n+p} \bar{\boldsymbol{\varepsilon}}_j}{\left(\langle\langle \mathcal{H}_\zeta \rangle\rangle \, \bar{\boldsymbol{u}}\right)_i + \left(\langle\langle \mathcal{H}_\Delta \rangle\rangle \, \bar{\boldsymbol{\varepsilon}}\right)_i} \right) < 1. \tag{5.82}$$

Hence, using the above condition we can deduce the upper bounds on the $\varepsilon_{WCPG_\zeta}$ and $\varepsilon_{WCPG_\Delta}$:

$$0 \leqslant \frac{\varepsilon_{WCPG_\zeta} \sum_{j=1}^{q} \bar{\boldsymbol{u}}_j + \varepsilon_{WCPG_\Delta} \sum_{j=1}^{n+p} \bar{\boldsymbol{\varepsilon}}_j}{\left(\langle\langle \mathcal{H}_\zeta \rangle\rangle \, \bar{\boldsymbol{u}}\right)_i + \left(\langle\langle \mathcal{H}_\Delta \rangle\rangle \, \bar{\boldsymbol{\varepsilon}}\right)_i} < 1. \tag{5.83}$$

Since all the terms are positive, the left inequality is always true. The right inequality in (5.83) is satisfied for instance if

$$\frac{\varepsilon_{WCPG_\zeta} \sum_{j=1}^{q} \bar{\boldsymbol{u}}_j}{\left(\langle\langle \mathcal{H}_\zeta \rangle\rangle \cdot \bar{\boldsymbol{u}}\right)_i} < \frac{1}{2} \qquad \frac{\varepsilon_{WCPG_\Delta} \sum_{j=1}^{n+p} \bar{\boldsymbol{\varepsilon}}_j}{\left(\langle\langle \mathcal{H}_\Delta \rangle\rangle \cdot \bar{\boldsymbol{\varepsilon}}\right)_i} < \frac{1}{2}. \tag{5.84}$$

Rearranging terms, we obtain following inequalities on the WCPG computation with error:

$$\varepsilon_{WCPG_\zeta} < \frac{1}{2} \cdot \frac{\left(\langle\langle \mathcal{H}_\zeta \rangle\rangle \, \bar{\boldsymbol{u}}\right)_i}{\sum_{j=1}^{q} \bar{\boldsymbol{u}}_j} \qquad \varepsilon_{WCPG_\Delta} < \frac{1}{2} \cdot \frac{\left(\langle\langle \mathcal{H}_\Delta \rangle\rangle \, \bar{\boldsymbol{\varepsilon}}\right)_i}{\sum_{j=1}^{n+p} \bar{\boldsymbol{\varepsilon}}_j}. \tag{5.85}$$

Unfortunately, the above results cannot be used in practice, since they depend themselves on the exact WCPG matrices.

It can be shown that $\underline{\langle\langle\mathcal{H}\rangle\rangle}$ is a lower bound of the WCPG matrix. We can compute this matrix exactly. Obviously,

$$\frac{\left(\underline{\langle\langle\mathcal{H}_\Delta\rangle\rangle}\cdot\bar{\boldsymbol{\varepsilon}}\right)_i}{\sum_{j=1}^{p+n}\bar{\boldsymbol{\varepsilon}}_i} \leqslant \frac{(\langle\langle\mathcal{H}_\Delta\rangle\rangle\cdot\bar{\boldsymbol{\varepsilon}})_i}{\sum_{j=1}^{p+n}\bar{\boldsymbol{\varepsilon}}_i} \tag{5.86}$$

$$\frac{\left(\underline{\langle\langle\mathcal{H}_\zeta\rangle\rangle}\cdot\bar{\boldsymbol{u}}\right)_i}{\sum_{j=1}^{q}\bar{\boldsymbol{u}}_i} \leqslant \frac{(\langle\langle\mathcal{H}_\zeta\rangle\rangle\cdot\bar{\boldsymbol{u}})_i}{\sum_{j=1}^{q}\bar{\boldsymbol{u}}_i}. \tag{5.87}$$

Hence, if the WCPG matrices in the right sides of (5.85) are substituted with their lower bounds, the condition (5.83) stays satisfied and we obtain bounds (5.77) and (5.78).

□

Analogously, Lemma 5 can be applied to the computation of $\widehat{\boldsymbol{m}}_{\zeta_i}$ with the terms concerning filter $\mathcal{H}_\Delta$ set to zero.

In order to guarantee the output MSB position to be optimal, an instance of the Table Maker's Dilemma [197] must be solved. This is due to the simplification by using the triangle inequality in (5.72), which basically requires both filters $\mathcal{H}_\zeta$ and $\mathcal{H}_\Delta$ reach the worst-case bound with the same input sequence.

### 5.3.5 Complete algorithm

The two-step algorithm, presented in Section 5.3.3 takes into account accumulation of computational errors in a filter over time and Lemma 5 presents error-analysis of the MSB position computation procedure. However, one additional fact has not been taken into account.

In most cases the MSB vectors $\widehat{\boldsymbol{m}}_\zeta$ (computed on Step 1) and $\widehat{\boldsymbol{m}}_\zeta^\Diamond$ (computed on Step 2) are the same. However, in some cases they are not, which can happen due to one of the following reasons:

— the accumulated rounding error due to the FxPF $(\widehat{\boldsymbol{m}}_\zeta, \widehat{\boldsymbol{\ell}}_\zeta)$ makes output of the actually implemented filter pass over to the next binade; or

— the floating-point approximation $\widehat{\boldsymbol{m}}_\zeta^\Diamond$ is off by one.

Moreover, if the MSB position after Step 2 of the algorithm is increased, the LSB position moves along and increases the error. Therefore, the modified format must be re-checked. Hence, the FxPF determination algorithm gets transformed into the following three-step procedure:

Step 1: Determine the FxPF $(\widehat{\boldsymbol{m}}_\zeta, \widehat{\boldsymbol{\ell}}_\zeta)$ for the exact filter $\mathcal{H}_\zeta$;

Step 2: Construct the "error-filter" $\mathcal{H}_\Delta$, which shows the propagation of the computational errors induced by format $(\widehat{\boldsymbol{m}}_\zeta, \widehat{\boldsymbol{\ell}}_\zeta)$; then, compute the FxPF $(\widehat{\boldsymbol{m}}_\zeta^\Diamond, \widehat{\boldsymbol{\ell}}_\zeta^\Diamond)$ of the actually implemented filter $\mathcal{H}_\zeta^\Diamond$;

Step 3: If $\widehat{\boldsymbol{m}}_{\zeta_i}^\Diamond == \widehat{\boldsymbol{m}}_{\zeta_i}$, then return $(\widehat{\boldsymbol{m}}_\zeta^\Diamond, \widehat{\boldsymbol{\ell}}_\zeta^\Diamond)$;
otherwise $\widehat{\boldsymbol{m}}_{\zeta_i} \longleftarrow \widehat{\boldsymbol{m}}_{\zeta_i} + 1$ and go to Step 2.

| | states | | | output |
|---|---|---|---|---|
| | $x_1(k)$ | $x_2(k)$ | $x_3(k)$ | $y(k)$ |
| **Step 1** | 6 | 7 | 5 | 6 |
| **Step 2** | 6 | 7 | 6 | 6 |
| **Step 3** | 6 | 7 | 6 | 6 |

Table 5.2 – Evolution of the MSB positions vector through algorithm iterations

### 5.3.6 Numerical results

The above described algorithm was implemented as a C library, using GNU MPFR [5] [86] version 3.1.12, GNU MPFI [6] version 1.5.1 and the WCPG library [252].

Consider the following example: let $\mathcal{H}$ be a random stable filter with 3 states, 1 input and 1 output. Suppose the inputs are in an interval absolutely bounded by $\bar{\boldsymbol{u}} = 5.125$. In this example we set all the wordlength constraints to 7 bits, however our library supports the multiple wordlength paradigm.

The results of the work of our algorithm can be observed in Table 5.2. On the Step 1 we deduce the MSB positions for the filter $\mathcal{H}$, which does not take computational errors into account. These MSBs serve as an initial guess for the Step 2. We compute the vector $\bar{\varepsilon}_\zeta$ and construct the "error-filter" $\mathcal{H}_\Delta$. Taking into account the error propagation yields changes in the MSB positions: the rounding errors force the third state $\boldsymbol{x}_3(k)$ to pass over to the next binade. However, moving the MSB yields larger quantization step and an addition step is required. Step 3 verifies that the FxPF deduced on the Step 2 guarantees no overflow.

We can verify the result by tracing the state and output vectors in two cases: the exact vectors $\boldsymbol{y}(k)$ and $\boldsymbol{x}(k)$; and the quantized vectors $\boldsymbol{y}^\diamondsuit(k)$ and $\boldsymbol{x}^\diamondsuit(k)$. The quantized vectors are the result of an implementation of the filter $\mathcal{H}$ with FxPF deduced on the Step 1. For each state and output we take an input sequence which makes the state respectively output attain the theoretical bound. However, this sequence is not guaranteed to maximize the quantized error.

In Figure 5.5 we can observe that the exact vector $\boldsymbol{y}(k)$ attains the bound computed with the WCPG theorem. The quantized output stays within the bound $\bar{\boldsymbol{y}}^\diamondsuit$ for the implemented filter but passes over the bound of the exact filter. However, as the bound $\bar{\boldsymbol{y}}$ is far from the next binade, the MSB position is not changed and the computational errors do not result in overflow.

For the third state $\boldsymbol{x}_3(k)$, its bound $\bar{\boldsymbol{x}}_3$ is very close to the $2^{\boldsymbol{m}_{x_3}} - 2^{\boldsymbol{\ell}_{x_3}}$ and taking into account the filter computational errors yields passing to the next binade. It can be clearly observed in Figure 5.6 that the quantized state passes over the value $2^{\boldsymbol{m}_{x_3}} - 2^{\boldsymbol{\ell}_{x_3}}$. If the computational errors were not taken into account, the initial format (deduced on the Step 1) would result in overflow.

Therefore, we observed that deducing the FxPF without taking into account the computational errors can result in overflow but our approach guarantees a reliable implementation without overestimation.

5. `http://www.mpfr.org/`
6. `https://gforge.inria.fr/projects/mpfi/`

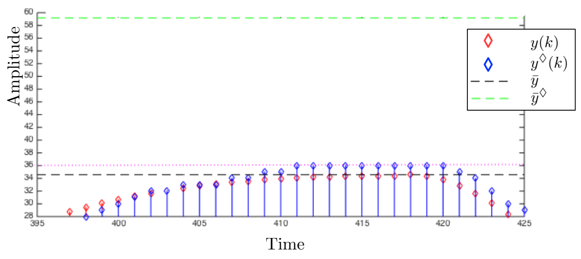

Figure 5.5 – The exact and quantized outputs of the example. Quantized output does not pass over to the next binade.

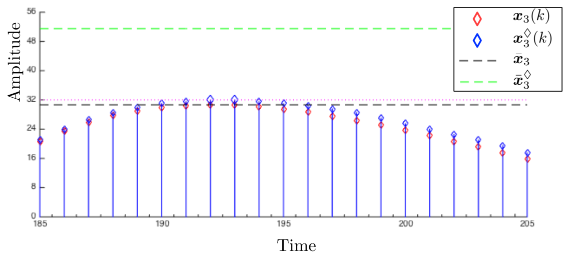

Figure 5.6 – The exact and quantized third state of the example. Quantized output passes over to the next binade.

### 5.3.7 Conclusion on Worst-Case Peak Gain Exploitation

We give an algorithm to determine the FxPF for all variables of a LTI filter, where the computation errors are taken into account. We ensure, by construction, that no overflow occurs. However, the computed MSB positions can overestimate at most by one bit. In order to guarantee no overestimation at all, an instance of the TMD must be solved. Our algorithm can be extended to any LTI filter realization (Direct Forms, Lattice filters, etc.) using the Specialized Implicit Framework (SIF) tool [113]. SIF enables unification of various LTI filter realizations for analysis, comparison and FxP code generation. Therefore, this work is an essential basic brick for the automatic filter-to-code conversion.

## 5.4 *A priori* error bounds: still a long way to go

In this Chapter 5, we have considered numerical algorithms under the aspect of adapting their compute precision to rigorously meet an *a priori* error bound. In the first case, described in Section 5.2, it was up to the multiple-precision algorithm itself to adapt its compute precision to meet a *dynamic a priori* error bound. In Section 5.3, we then used that algorithm to perform *static* compute precision adaption –by finding the right MSB and LSB positions in fixed-point arithmetic– for code generation for an LTI filter.

We think that this novel approach to the precision vs. accuracy issue of numerical algorithms is an interesting field of research for future work: similarly as we did in Chapter 3, when we moved from single floating-point operations, like one single fused-multiply-add operation described in Chapter 3, Section 3.4, to more high-level floating-point operations like vector 2-norms, as we described them in Chapter 3, Section 3.6, we are now to move from single multiple-precision operations, like they are coded in libraries as MPFR, to "multiple-accuracy" algorithms that adapt their compute precision to this dynamic *a priori* accuracy goal. Recent research has made us reconsider basic steps in mathematical function implementation, such as polynomial approximation with *a priori* error control [159]. This research is still too preliminary to be reported here but it clearly does follow the lead of *a priori* error bound work set out here. As a matter of course, there is still a long way to go.

# Chapter 6

# Conclusion and future work

*Et ceterum censeo computationem numericam simplicorem et securam esse reddendam.*

with Cato the elder

## 6.1 An enhanced Floating-Point environment: work done so far

For this work, we came up with the observation that the IEEE754 Standard for Floating-Point Arithmetic [117] is a solid base for proven and reliable floating-point arithmetic but that it is too fine-grained for the exa-scale world. Extensions to the Standard are hence required to move on.

We started with an introduction to the environment defined by IEEE754-2008 and the open implementatory issues that are still to be found on modern computer systems.

We then looked into various operations and associated floating-point algorithms to complete the floating-point environment, e.g. with respect to heterogeneous operations or correctly rounded decimal-to-binary input operations, and, in a second step, how to enhance the IEEE754 environment. We proposed algorithms for mixed-radix floating-point comparisons, allowing a binary and a decimal floating-point number to be compared, and looked into general mixed-radix arithmetic. Our work on a mixed-radix fused-multiply-and-add operation where inputs and outputs of both radices $2$ and $10$ could freely be mixed proved harder than we thought and remains a certain challenge.

We also proposed extended precision floating-point operations in the precision range starting at double precision and ranging up to about $512$ bits, covering a precision range where general purpose arbitrary precision libraries cannot yet leverage their optimality in terms of asymptotic complexity.

Enhancements to the floating-point environment can also be made through operations at a higher abstraction level. We tried to illustrate this point at the instance of a faithfully rounded two-norm algorithm that is easy to parallelize, free of spurious overflow and underflow and up to $4.5$ times faster than the existing reference algorithm.

We then passed to a meta-level where we do no longer try to design numerical codes by hand but instead "use" numerical code generation to produce provably accurate numerical codes. The challenge here is that numerical code generation is not yet ready to "use". We must design them. We took this challenge for the example of mathematical functions. We first looked into classical elementary functions, such as $e^x, \log, \sin, \cos$, in order to see how the different basic bricks for their implementation, like argument reduction, polynomial approximation and reconstruction using tables, could be automatically analyzed for accuracy and transformed into floating-point code. In a second step we got involved in generating code for functions that are less common and less described in the literature than, say, exponential. For these functions, most of them being special functions, code generation has less to bother about algebraic argument reduction but more with optimal domain splitting and polynomial approximation. However, to compute such a polynomial approximation, a chicken-and-egg problem is encountered, as a code is needed to evaluate the given function in order to compute the polynomials that end up in the generated code. We solved this issue for a certain type of functions for which evaluation through computation and evaluation of series with recurrences in the series' coefficient is possible.

This work on code generation for mathematical functions made us exhibit a close link between the steps required to generate code for a mathematical function and the ones required for code generation of a Linear Time Invariant filter: approximation with polynomials resp. rational functions, choice of an evaluation scheme, precision adaption per basic operation, final accuracy proof. We worked towards integration of the two software toolchains, however, we encountered certain issues.

This work resulted in yet another way of considering enhanced floating-point arithmetic: in certain steps of code generation for Linear Time Invariant filters, a certain measure, for instance the Worst Case Peak Gain matrix of the filter, needs to be computed to allow the fixed-point arithmetic formats to be chosen in the final implementation. In order for the final filter implementation to be reliable, the Worst Case Peak Gain matrix needs to be computed reliably. We perform this computation in floating-point arithmetic but require certain features of multiple precision to be available, such as dynamic adaptation of the precision a variable bears in a code. This way we are able to guarantee that an *a priori* error bound is satisfied for the results produced by the algorithm. Hence, the fixed-point arithmetic code generation step is reliable in the sense that it never underestimates the most-significant-bit position of variable in the filter's code. It is also reliable in the sense that the possible overestimate of that position is also bounded, typically by one single bit.

This last work seems important to us in particular as it shows the long-term possibilities of an enhanced floating-point environment. Using non-standard multiple precision floating-point arithmetic, we implemented a numerical algorithm computing a certain measure with an error bounded in an *a priori* manner, just to use this algorithm in a code generator to finally generate a filter algorithm in fixed-point arithmetic that is reliable, accurate and free of overflow.

To sum up again, we have worked at enhancing the floating-point environment at the following levels:

— complete the implementation of IEEE754 on common systems,

— extend the choice of compute precisions in the mid-precision range,

— extend the environment with mixed-radix operations,

— going to high level operations such as faithfully rounded Euclidian norms,

— passing to the meta-level with numerical codes generating other numerical codes, for instance implementations of mathematical functions,

— passing from elementary functions to special functions for which no reference implementation is available and

— using features offered by multiple precision to provide algorithms for certain measures with *a priori* bounds in order

— to generate reliable fix-point implementations of Linear Time Invariant Filters.

## 6.2 Future work

Rich of this experience, we are eager to take up other challenges, which come up in particular when restrictions we imposed on the objects we worked with are removed. For instance, with our work on an enhanced IEEE754 environment, we were always concerned of standard compliance and accuracy but less of the energy consumption. In our work on code generation for mathematical functions, we only looked at transcendental, univariate functions, setting aside all bivariate or multivariate numerical expressions that can come up in numerical codes. However, we have never looked into the energy consumption perspective of a –possibly weakened– IEEE754 environment, code generation for more general numerically challenging compute kernels or automatized analysis of filters that are not easily subsumed as Linear Time Invariant Filters, such as adaptive or Kalman filters.

This research project is divided into 3 possible axes, described in Sections 6.2.1 through 6.2.3 below. We have written these three sections in a style that should allow us to easily transpose them into Ph.D. research subjects. As a matter of course, we intend to work in these three directions, be it in cooperation with Ph.D. students or be it without.

These three research directions which we clearly identify as the next three steps do not form an exhaustive list of fields we want to work on. Section 6.2.4 gives a more general, less project-driven view on our planned research for the next couple of years.

Finally, our work started in the fields of mixed-radix operations and mathematical function implementation is not completely finished and it will stay one of our matters of interest at least in the short term.

### 6.2.1 The energy perspective of floating-point environment features

Computing is more and more limited by the amount of energy spent by the compute systems that are used. Software to recover simplified hardware can be an answer.

**Framework of the project** The IEEE754 Standard's floating-point environment provides a certain amount of features that certainly ease debugging and make floating-point arithmetic more reliable, as for example IEEE754 status flags, dynamic rounding modes, non-standard exception handling [117]. Hardware and software implementations of IEEE754 must compute these additional results, which requires a certain amount of energy.

However, these features are hardly used in numerical applications. Even more, IEEE754 features like correctly rounded basic operations or just the use of IEEE754 binary64 numbers have impact on the energy spent by a computer system. In particular, memory access is energy-intensive [103].

In the recent years, there have been several attempts to address this issue. There are typically two main tendencies: reducing energy consumption by reducing the number of memory accesses, by techniques called Communication Avoidance [103], and by more disruptive proposals requiring a complete redesign of compute hardware, with number systems like the so called Unum [107].

With this project we would like to look on a middle ground between these two competing approaches. We would like to consider simplifications to the IEEE754 environment that would result in lowered energy consumption while maintaining accuracy at a high level. We would like to consider in particular a novel approach where software recovers rare cases that are no longer handled in simplified hardware. Computer arithmetical approaches to memory transfers, such as writing only the high order 32 bit word of a double precision number, also fall into this category.

This research can look into:

— Estimation of the energy cost of rarely used IEEE754 features (flags, rounding modes), while proposing a software emulation layer for the rare occasion they are needed as well as for compliance reasons.

— Feasibility of mixed-precision computations maintaining precision-sensitive parts in registers and caches, while writing rounded results back to memory. This may require working on enhancements on the hardware side.

— Use of extremely low precision formats, such as binary16 (half-float) [117].

— Use of compensated algorithms [97] together with possible use of significance arithmetic [69]. The novelty would lay in compensating just at the points where this has an effect.

**Challenges** This project has to confront a certain number of challenges:

— Hardware vendors are pretty discrete with numbers showing energy consumption of certain parts of the general-purpose processors they sell. To our knowledge there exist only very few studies on the energy consumption of general-purpose floating-point hardware environments that are supposed to comply with the IEEE754-2008 standard [260]. This project will have to work out means to estimate energy consumption of floating-point operations in general purpose processors.

— In order to perform the proposed studies on energy consumption, access to and understanding of large scale applications that can serve as objects of analysis and showcases must be gained.

— In order to try out mixed-precision arithmetic, compensation techniques or other arithmetic approaches, a certain accuracy model of the applications to be optimized for energy consumption is required. Such accuracy models are generally pretty hard to obtain.

We should also add that most of our previous work was on the software side of Computer Arithmetic. This project must consider both the hardware and the software implications of the energy perspective of IEEE754-2008 floating-point environments. Questions of hardware-software co-design will arise [46]. We are eager to take up this challenge.

**Possible outcomes** It is expected that this project gives a better understanding of the energy perspective of certain features of the floating-point environment and on how a good balance between accuracy of the results and energy consumption can be obtained. Given the wideness of the field of possible numerical applications, this project, in particular when implemented as as Ph.D. research project, will not be able to give a conclusive answer but only a first understanding of further fields to be studied.

**Deliverables** This project should start off with the following deliverables in mind:

— Tools and software to estimate the energy consumption when code is run in one or the other floating-point environment.

— Accuracy benchmarks for mixed-precision, compensated, significance arithmetic-driven codes.

— A production-level application using low precision to save up on energy, possibly in the fields of deep-learning, signal processing or linear algebra.

### 6.2.2 Identification of precision-sensitive kernels for code generation

Code generation for large-scale numerical applications is currently out of reach, yet. Significant performance gain is however expected from the speed up obtained with code generation for certain precision-sensitive kernels.

**Framework of the project** Code generation for univariate mathematical function generation has been shown to be feasible, e.g. in Chapter 4 of this work. Numerical applications however need more than univariate functions; they contain quite a few number of expressions that are both multivariate and a bottleneck to the applications' performance and accuracy.

A very easy-to-understand example of such accuracy and performance bottlenecks classically are sum reductions [31,61]. Floating-point addition is most sensible to amplify rounding errors due to cancellation [111]. In many applications, it is hence the additions that dictate the common compute precision, like IEEE754 binary64, binary128 etc. In other parts of the code using a larger precision might however not be required as it just lowers the overall code performance.

It is hence reasonable to try to identify kernels in numerical code that are most precision-sensitive and to apply code generation to these parts. The intended code generator is supposed to take just mathematical expressions in input, together with accuracy bounds to achieve. The code generator can then apply intensive optimization techniques for this kernel, both for accuracy and performance.

The challenging part here is the integration of the kernels, for which code generators may want to choose the input and output formats, into the existing applications [61].

On the other hand, the proposed approach on code generation for compute-intensive and precision-sensitive kernels opens yet another, novel possibility for floating-point environments: performance-wise, these kernels do not actually need to cover the whole range of possible inputs. When a reference kernel is available –which might just be generated with no performance optimization and only accuracy in mind– there may be one or several performance-optimized kernels for the mostly used ranges of inputs. Simple tests use either the reference kernel or the optimized one.

A valid question is how the code generator would know which range of inputs it would be reasonable to apply code optimization on. We claim that this decision can actually be inferred from simple statistics made on reference code, similarly to what has already been done for mathematical functions [210]. It is part of this project to confirm –or refute– this claim.

As no human intervention is required for a code generator for numerical kernels, the three steps consisting in reference code statistics, code generation and final execution of the generated code could actually be integrated. An application would start out with reference kernels which would be enhanced with call and input range statistics code, running in parallel. Still in parallel, code generation would provide optimized versions of the numerical kernels, whenever a particularly compute-intensive input range is identified. These new, optimized kernels could then dynamically be bound to the running code. Code optimization hence becomes an *on-the-fly process*. The aims is to investigate the feasibility of this dynamic approach.

**Challenges** This project is challenged by a number of issues that might be difficult to resolve:

— Its goals is intentionally not well-defined in the sense that most of numerical problems can be seen as multivariate functions to be implemented; however it is clear that defining for example a QR-decomposition of a $n \times n$ matrix as a function in $n^2$ arguments makes less sense. It is hence necessary in this project to clearly define the domain in which working on multivariate functions for code generation makes actually sense.

— Code generation for mathematical functions is based, in its main steps, on the possibility to reliably determine tight upper bounds of approximation errors [40] (see also Chapter 4.2.2). There is no equivalent approach available for multivariate expressions. We expect issues with the combinatorial explosion due to the higher dimensionality.

— The *on-the-fly* code generation process requires code generation to run in a fully automatic manner without requiring any human supervision or intervention. The code generators for mathematical functions and Linear Time Invariant Filters, discussed in Chapter 4, are mainly automated but require some manual intervention in their input setup, like for example with code generation parameters, and in their supervision.

— In order to demonstrate the efficiency of the proposed compute kernel approach, showcase and benchmark numerical applications will be needed. Cooperation with teams from industry who have access to production level numerical applications needs to be established for this project.

**Possible outcomes** The possible outcomes of this project are on the one hand a better understanding of what can be understood as a compute kernel and what numerical optimizations can be performed, in scientific compute applications. On other hand, the project may open up our vision on a modern floating-point environment even more, typically by establishing the possibility of *on-the-fly* optimization of numerical codes.

**Deliverables** This project is supposed to produce the following deliverables:

— a clear analysis on the various types of compute-intensive, precision-sensitive multivariate expressions that figure in compute kernels,

— a prototype implementation of a code generator for a chosen class of compute kernels, including sum-of-products and, more generally, sums-of-univariate-expressions, with *a priori* error bounds,

— a prototype implementation of input range statistics code for exploitation with the code generator, together with a representative testbench of production-level numerical code

— and first tests on a few applications trying to validate –or invalidate– the dynamic kernel optimization scheme set out above.

### 6.2.3 Floating and fixed-point code generation for Kalman filters

Linear Time Invariant Filters are used to amplify or attenuate certain frequencies in a given signal. The signal, and hence the output, is supposed to be exact, though. Kalman filters are used to filter measurement noise out of a signal. Their implementation, especially in fixed point arithmetic, is even more challenging.

**Framework of the project** Enhanced computer arithmetic has been shown to provide efficient tools for the analysis and subsequent code generation of Linear Time Invariant Filters, see e.g. Chapter 5 of this work. Reliable code can be generated for such filters provided dependable advanced multi-precision floating-point codes for the analysis of their dynamical behavior.

Linear Time Invariant Filters are used to amplify or attenuate certain frequency bands in a signal's spectrum. They are hence helpful to select certain bands in a spectrum before submitting them to other signal processing, such as demodulation or to prevent controlled system from oscillation, by removing the system's resonance frequencies [88]. They are also helpful when measurement noise as particular spectral components that require to be filtered out. However, Linear Time Invariant Filters cannot be used to filter out measurement noise by comparing a measured signal to a system model.

Kalman filters [29] are used to match a dynamical system's model in real time to an input signal of measurements, hence filtering out noise in that input stream. Additionally, Kalman filters allow unobservable signals, i.e. signals for which no direct measurement process is available, to the estimated, based on the system model and the input of observable quantities.

Kalman filters are special cases of Stratonovich-Kalman-Bucy filters and closely related to analysis of hidden Markov models [115], which, in turn, are intensively used in genome sequencing, speech recognition and machine learning in general [91].

The basic idea behind a Kalman filter is a feedback loop adjusting a model's internal state by comparing the model's predicted output to the measurements fed into the filter. The model's internal state can then be used to provide output, freed from the random noise nature of the input, as well as additional signals. Software phase-locked-loops (PLLs) [60] are extremely simplified versions of Kalman filters. Those PLLs can be used to perform signal demodulation, something Linear Time Invariant Filters cannot do.

Similarly to Linear Time Invariant Filters, the challenging part when implementing Kalman filters using floating-point and, in particular, fixed-point arithmetic, to predict the dynamic ranges of all variables, i.e. internal signals, that occur in the implementation, as well as the resulting roundoff errors. For Kalman filters, this is particularly challenging, as these filters manipulate signals that are intrinsically probabilistic variables. Classical range and error analyses often work with intervals that express worst-case bounds. Probabilistic considerations are typically left out. Applying this classical methodology to Kalman filters hence provides very pessimistic bounds, in particular, when taking into account that Kalman filters are self-correcting, due to their internal feedback loop [151].

Nevertheless, the basic bricks constituting a Kalman filter, i.e. the prediction and correction computation steps, are often, taken by themselves, linear filters. Applying the existing methodology to analyze them and to implement them with automatically generated code is thus possible. Starting with this observation, this project is concerned with the design of a Kalman filter analysis and implementation approach that can be transformed into a Kalman filter code generator.

**Challenges** This project, concerned with the analysis of and code generation for Kalman filters, faces several challenges among which figure the following ones:

— Kalman filters manipulate probabilistic variables, which is a concept uncommon to computer arithmetic. They are self-correcting, as long as they stay locked onto the input stream; range overflows or extreme approximation errors may hence be tolerable as long as they do not prevent the filter from staying locked. However, the modeling of the probabilistic behavior of the input signal is often approximate, which may hide potential issues.

— Kalman filters contain a feedback loop which prevents the use of pure naive interval arithmetic or even Taylor Models for range and roundoff error analysis.

— As basic bricks, these filters require code generation for the predictor and corrector computational steps. These often imply matrix operations that are ill-conditioned and hence hard to implement in fixed-point arithmetic.

**Possible outcomes** The use of Kalman filters is widespread in industry; reliability is an issue. The main, eventual, possible outcome of this research project is providing a methodology for the reliable implementation of Kalman filters, supported as much as possible, by automatic code generation. Industrial applicability would be immediate.

Given the challenges which this project must take on, a first deep understanding of the required basic bricks and methodology paving the road to implementation of a code generator for Kalman filters should already be a reasonable outcome of this project.

**Deliverables** This project should start off with the following deliverables in mind:

— a unified way of modeling Kalman filters with a framework that allows for their computer-arithmetic centered analysis,

— tools and methodology for error and range analysis in Kalman filters,

— a prototype of a code generator for Kalman filters, in floating and fixed-point arithmetic

— and significant showcase example Kalman filters, together with their –possibly generated– implementations, taken from applicative domains.

### 6.2.4 Open-Ended Future Research Directions

In addition to the mid-term research projects set out in the previous Sections 6.2.1 through 6.2.3, we would like, in the long run, look into the following fields of research. This part can also be done in a less formalized manner, e.g. without advising Ph.D. students.

**Code translation and code generation for Interval Arithmetic** Interval Arithmetic is an efficient tool when it comes to providing reliable numerical results, without going through tedious error analysis [217]. However, Interval Arithmetic is plagued by certain effects that may make its result reliable but useless because not sufficiently accurate. The most preponderant issue here are the decorrelation effect and the wrapping effect.

Interval Arithmetic can hence seldom be used as a drop-in replacement too floating-point arithmetic, which would somehow make legacy floating-point applications instantly, almost magically, reliable.

Interval Arithmetic can be enhanced, for example using Taylor Models [130], to prevent, at least for certain classes of problems, the negative impacts of plain Interval Arithmetic. The theory of verified inclusions [225] is another approach to get around these issues. However, these enhanced approaches require the structure of an existing code to be modify profoundly which eliminates their application as a drop-in replacement to floating-point arithmetic.

It might seem that numerical code analysis, translation and finally code regeneration might allow for transformations that start out with floating-point code and transform it, using enhanced Interval Arithmetic. We would like to address this point in a mid- long-term timeframe.

**Numeric-symbolical computations** The second long-term research area we would like to invest in, symbolic-numeric methods, would study how rigorous, mathematically sound results can be obtained using approximate, but still proven and reliable computational techniques. The resulting software could then lead to safe computer-based mathematical results, similar to the ones provided by a Computer Algebra System, but utilizing classical techniques from scientific computing. In particular, we have thought about multiprecision interval arithmetic techniques which, used with a compiler and library environment supporting it, could on the one hand provide reasonably fast rigorous answers. Such compiler systems would also allow for a more tight coupling of multiprecision (interval) arithmetic with standard compilation, hence enabling dynamic precision adaptation for compiled code and avoiding indirect memory accesses and heap memory handling.

# Publications

In this Section we give an exhaustive list of our publications in journals and in the proceedings of peer-reviewed international conferences and peer-reviewed workshops. In addition, we also give a list of select presentations and workshops. An extensive list of all our publications is available on our webpage

`http://www.christoph-lauter.org/`.

**Journal articles**

[100] S. Graillat, C. Jeangoudoux, and C. Lauter. MPDI: A Decimal Multiple-Precision Interval Arithmetic Library. *Reliable Computing Journal*, pages 38–52, 2017.

[28] N. Brisebarre, C. Lauter, M. Mezzarobba, and J.-M. Muller. Comparison between binary and decimal floating-point numbers. *IEEE Transactions on Computers*, 65(7):2032–2044, July 2016.

[101] S. Graillat, C. Lauter, P. T. P. Tang, N. Yamanaka, and S. Oishi. Efficient calculations of faithfully rounded l2-norms of n-vectors. *ACM Trans. Math. Softw.*, 41(4):24:1–24:20, Oct. 2015.

[66] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres. On Ziv's rounding test. *ACM Trans. Math. Softw.*, 39(4):25:1–25:19, July 2013.

[40] S. Chevillard, J. Harrison, M. Joldeş, and C. Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoretical Computer Science*, 412(16):1523 – 1543, 2011. Symbolic and Numerical Algorithms.

[64] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, Feb. 2011.

[139] P. Kornerup, C. Lauter, V. Lefèvre, N. Louvet, and J.-M. Muller. Computing correctly rounded integer powers in floating-point arithmetic. *ACM Trans. Math. Softw.*, 37(1):4:1–4:23, Jan. 2010.

[166] C. Q. Lauter and V. Lefèvre. An efficient rounding boundary test for pow(x,y) in double precision. *IEEE Transactions on Computers*, 58(2):197–207, Feb 2009.

[65] F. de Dinechin, C. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *RAIRO - Theoretical Informatics and Applications*, 41(1):85–102, April 2007.

**Articles in conference proceedings, including peer-reviewed workshops**

[159] C. Lauter. Rigorous polynomial approximation. In *2018 52nd Asilomar Conference on Signals, Systems and Computers*, pages 120–124, 2018.

[125] C. Jeangoudoux and C. Lauter. A correctly rounded mixed-radix fused-multiply-add. In *2018 IEEE 25th Symposium on Computer Arithmetic*, pages 17–24, July 2018.

[130] M. Joldes, C. Lauter, M. Ceberio, O. Kosheleva, and V. Kreinovich. Why Taylor models and modified Taylor models are empirically successful : A symmetry-based explanation. In *Proceedings of the 8th International Workshop on Reliable Engineering Computing REC'2018*, July 2018. Proceedings published online.

[158] C. Lauter. An efficient software implementation of correctly rounded operations extending FMA: $a + b + c$ and $a \times b + c \times d$. In *2017 51st Asilomar Conference on Signals, Systems and Computers*, pages 452–456, 2017.

[253] A. Volkova, T. Hilaire, and C. Lauter. Reliable verification of digital implemented filters against frequency specifications. In *2017 IEEE 24th Symposium on Computer Arithmetic*, pages 180–187, July 2017.

[157] C. Lauter. A new open-source SIMD vector libm fully implemented with high-level scalar C. In *2016 50th Asilomar Conference on Signals, Systems and Computers*, pages 407–411, Nov 2016.

[156] C. Lauter. Easing development of precision-sensitive applications with a beyond-quad-precision library. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 742–746, Nov 2015.

[251] A. Volkova, T. Hilaire, and C. Lauter. Determining fixed-point formats for a digital filter implementation using the worst-case peak gain measure. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 737–741, Nov 2015.

[32] N. Brunie, F. de Dinechin, O. Kupriianova, and C. Lauter. Code generators for mathematical functions. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 66–73, June 2015. Best Paper Award.

[252] A. Volkova, T. Hilaire, and C. Lauter. Reliable evaluation of the worst-case peak gain matrix in multiple precision. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 96–103, June 2015.

[163] C. Lauter and M. Mezzarobba. Semi-automatic floating-point implementation of special functions. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 58–65, June 2015.

[147] O. Kupriianova and C. Lauter. A domain splitting algorithm for the mathematical functions code generator. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1271–1275, Nov 2014.

[148] O. Kupriianova and C. Lauter. Metalibm: A mathematical functions code generator. In H. Hong and C. Yap, editors, *ICMS 2014*, volume 8592 of *LNCS*, pages 713–717. Springer, 2014.

[150] O. Kupriianova, C. Lauter, and J. M. Muller. Radix conversion for IEEE754-2008 mixed radix floating-point arithmetic. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 1134–1138, Nov 2013.

[27] N. Brisebarre, C. Lauter, M. Mezzarobba, and J.-M. Muller. Comparison between binary64 and decimal64 floating-point numbers. In A. Nannarelli, P.-M. Seidel, and P. T. P. Tang, editors, *ARITH 21*, pages 145–152, Austin, Texas, USA, 2013. IEEE Computer Society.

[41] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An Environment for the Development of Numerical Codes. In *Mathematical Software - ICMS 2010*, volume 6327 of *LNCS*, pages 28–31, Sept 2010.

[43] S. Chevillard, M. Joldes, and C. Q. Lauter. Certified and fast computation of supremum norms of approximation errors. In *19th Symposium on Computer Arithmetic*, pages 169–176. IEEE, 2009.

[160] C. Lauter and F. de Dinechin. Optimizing polynomials for floating-point implementation. In J. D. Bruguera and M. Daumas, editors, *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 7–16, Santiago de Compostela, Espagne, July 2008.

[45] S. Chevillard and C. Q. Lauter. A certified infinite norm for the implementation of elementary functions. In *Seventh International Conference on Quality Software (QSIC 2007), 11-12 October 2007, Portland, Oregon, USA*, pages 153–160, 2007.

[63] F. de Dinechin, C. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318–1322, Dijon, France, 2006.

**Select presentations and workshops, with or without peer-review**

[254] A. Volkova, T. Hilaire, C. Lauter, and M. Mezzarobba. Rigorous determination of recursive filter fixed-point implementation with input signal frequency specifications. In *2017 51st Asilomar Conference on Signals, Systems and Computers*, 2017.

[98] S. Graillat, Y. Ibrahimy, C. Jeangoudoux, and C. Lauter. A parallel compensated Horner scheme. In *Proceedings of ACA 2017, 23rd Conference on Applications of Computer Algebra, Jerusalem, Israel*, page 271, July 2017.

[99] S. Graillat, C. Jeangoudoux, and C. Lauter. A Decimal Multiple-Precision Interval Arithmetic Library. In *17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics*, Uppsala, Sweden, Sept. 2016.

[149] O. Kupriianova and C. Lauter. *Replacing Branches by Polynomials in Vectorizable Elementary Functions*, pages 14–22. Springer International Publishing, Würzburg, 2016.

[146] O. Kupriianova and C. Lauter. Metalibm. GNU Cauldron 2013, Google Headquarters, Montainview, CA, June 2013.

[161] C. Lauter and O. A. Kupriianova. The libieee754 compliance library for the IEEE 754-2008 standard. In *The 15'th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations, 2012*, Novosibirsk, Russia, Sept. 2012.

[162] C. Lauter and V. Ménissier-Morain. There's no reliable computing without reliable access to rounding modes. In *The 15'th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations, 2012*, Novosibirsk, Russia, Sept. 2012.

[155] C. Lauter. Sollya - a numerical software tool for the semi-automatic implementation of efficient correctly rounded mathematical functions. In *Invited presentation at ACA2008*, Hagenberg, Austria, July 2008.

[154] C. Lauter. A survey of multiple-precision using floating-point arithmetic. In *Fourth International Workshop on Taylor Models*, Boca Raton, FL, Dec. 2006.

[44] S. Chevillard and C. Lauter. Certified infinite norm using Interval Arithmetic. In *The 12'th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations, 2006*, Duisburg, Germany, Sept. 2006.

# Bibliography

[1] Digital library of mathematical functions, 2010. Companion to the *NIST Handbook of Mathematical Functions*.

[2] Decimal floating-point user's guide (Technology preview, with advance toolchain), Ver. 13.1. Technical report, IBM, 2013.

[3] AVR 8-bit microcontrollers, AVR42787: AVR software user guide, Application Note. Technical report, Atmel, 2016.

[4] M. Abramowitz and I. A. Stegun. *Handbook of mathematical functions*. National Bureau of Standards, Washington, D.C., 1964.

[5] G. Alefeld and D. Claudio. The basic properties of interval arithmetic, its software realizations and some applications. *Computers and Structures*, 67:3–8, 1998.

[6] J. Alex. *Zur Entstehung des Computers - von Alfred Tarski zu Konrad Zuse*. VDI-Verlag, Düsseldorf, 2007.

[7] C. S. Anderson, S. Story, and N. Astafiev. Accurate math functions on the Intel IA-32 architecture: A performance-driven design. In *7th Conference on Real Numbers and Computers*, pages 93–105, 2006.

[8] E. Anderson et al. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[9] J. Apostolakis, A. Buckley, A. Dotti, and Z. Marshall. Final report of the ATLAS detector simulation performance assessment group. Cern-lcgapp-2010-01, CERN/PH/SFT, 2010.

[10] J. Arndt. *Matters Computational. Ideas, Algorithms, Source Code*. Springer, 2011.

[11] D. H. Bailey and J. M. Borwein. High-precision computation and mathematical physics, 2015.

[12] J.-C. Bajard et al. Floating-point geometry: toward guaranteed geometric computations with approximate arithmetics. volume 7074, 2008.

[13] V. Balakrishnan and S. Boyd. On computing the worst-case peak gain of linear systems. *Systems & Control Letters*, 19:265–269, 1992.

[14] N. H. F. Beebe. A new math library. *International Journal of Quantum Chemistry*, 109(13):3008–3025, 2009.

[15] A. Benoit et al. The Dynamic Dictionary of Mathematical Functions (DDMF). In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 35–41. Springer, 2010.

[16] J. L. Blue. A portable Fortran program to find the Euclidean norm of a vector. *ACM Transactions on Mathematical Software*, 4(1):15–23, 1978.

[17] S. Boldo. Formal verification of tricky numerical computations. In *16th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, Würzburg, Germany, Sept. 2014. Invited talk.

[18] S. Boldo. Formal Verification of Programs Computing the Floating-Point Average. In M. Butler, S. Conchon, and F. Zaïdi, editors, *17th International Conference on Formal Engineering Methods*, volume 9407 of *Lecture Notes in Computer Science*, pages 17–32, Paris, France, Nov. 2015. Springer International Publishing.

[19] S. Boldo and M. Daumas. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms*, pages 45–60, 2004.

[20] S. Boldo, S. Graillat, and J.-M. Muller. On the robustness of the 2sum and Fast2sum algorithms. *ACM Trans. Math. Softw.*, 44(1):4:1–4:14, July 2017.

[21] S. Boldo and G. Melquiond. Emulation of FMA and Correctly-Rounded Sums: Proved Algorithms Using Rounding to Odd. *IEEE Transactions on Computers*, 57(4):462–471, 2008.

[22] S. Boldo and G. Melquiond. Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011.

[23] S. Boldo and J.-M. Muller. Exact and approximated error of the FMA. *IEEE Transactions on Computers*, 60:157–164, 2011.

[24] M. Brain, C. Tinelli, P. Rümmer, and T. Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 160–167, June 2015.

[25] R. P. Brent. Fast multiple-precision evaluation of elementary functions. *Journal of the ACM*, 23(2):242–251, 1976.

[26] N. Brisebarre and S. Chevillard. Efficient polynomial $L^\infty$-approximations. In P. Kornerup and J.-M. Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 169–176, Montpellier, France, June 2007. IEEE Computer Society Press, Los Alamitos, CA.

[27] N. Brisebarre, C. Lauter, M. Mezzarobba, and J.-M. Muller. Comparison between binary64 and decimal64 floating-point numbers. In A. Nannarelli, P.-M. Seidel, and P. T. P. Tang, editors, *ARITH 21*, pages 145–152, Austin, Texas, USA, 2013. IEEE Computer Society.

[28] N. Brisebarre, C. Lauter, M. Mezzarobba, and J.-M. Muller. Comparison between binary and decimal floating-point numbers. *IEEE Transactions on Computers*, 65(7):2032–2044, July 2016.

[29] R. G. Brown and P. Y. C. Hwang. *Introduction to random signals and applied Kalman filtering: with MATLAB exercises and solutions; 3rd ed.* Wiley, New York, NY, 1997.

[30] N. Brunie. *Contributions to computer arithmetic and applications to embedded systems.* PhD thesis, ÉNS de Lyon, 2014.

[31] N. Brunie. Modified fused multiply and add for exact low precision product accumulation. In *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, pages 106–113, July 2017.

[32] N. Brunie, F. de Dinechin, O. Kupriianova, and C. Lauter. Code generators for mathematical functions. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 66–73, June 2015. Best Paper Award.

[33] N. Brunie, H. de Lassus Saint-Geniès, and G. Revy. Meta-implementation of vectorized logarithm function in binary floating-point arithmetic. In *29th IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 2018.

[34] F. Y. Busaba et al. The IBM z900 decimal arithmetic unit. In *Thirty-Fifth Asilomar Conference on Signals, Systems, and Computers*, volume 2, pages 1335–1339, Nov. 2001.

[35] J. Carletta, R. Veillette, F. Krach, and F. Zhengwei. Determining appropriate precisions for signals in fixed-point IIR filters. In *Design Automation Conference, 2003. Proceedings*, pages 656–661, 2003.

[36] S. Carlough, A. Collura, S. Müller, and M. Kröner. The IBM zEnterprise-196 decimal floating-point accelerator. In *20th IEEE Symposium on Computer Arithmetic*, pages 139–146, July 2011.

[37] E. W. Cheney. *Introduction to Approximation Theory*. McGraw-Hill, New York, 1966.

[38] S. Chevillard. *Évaluation efficace de fonctions numériques – Outils et exemples*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2009.

[39] S. Chevillard. The functions erf and erfc computed with arbitrary precision and explicit error bounds. *Information and Computation*, 216:72 – 95, 2012. Special Issue: 8th Conference on Real Numbers and Computers.

[40] S. Chevillard, J. Harrison, M. Joldeş, and C. Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoretical Computer Science*, 412(16):1523 – 1543, 2011. Symbolic and Numerical Algorithms.

[41] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An Environment for the Development of Numerical Codes. In *Mathematical Software - ICMS 2010*, volume 6327 of *LNCS*, pages 28–31, Sept 2010.

[42] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In *Mathematical Software - ICMS 2010*, volume 6327, pages 28–31, 2010.

[43] S. Chevillard, M. Joldes, and C. Q. Lauter. Certified and fast computation of supremum norms of approximation errors. In *19th Symposium on Computer Arithmetic*, pages 169–176. IEEE, 2009.

[44] S. Chevillard and C. Lauter. Certified infinite norm using Interval Arithmetic. In *The 12'th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations, 2006*, Duisburg, Germany, Sept. 2006.

[45] S. Chevillard and C. Q. Lauter. A certified infinite norm for the implementation of elementary functions. In *Seventh International Conference on Quality Software (QSIC 2007), 11-12 October 2007, Portland, Oregon, USA*, pages 153–160, 2007.

[46] V. A. Chouliaras and J. L. Nunez-Yanez. An IEEE 754 floating point engine designed with an electronic system level methodology. In *Norchip 2007*, pages 1–4, Nov 2007.

[47] D. V. Chudnovsky and G. V. Chudnovsky. Computer algebra in the service of mathematical physics and number theory. In D. V. Chudnovsky and R. D. Jenks, editors, *Computers in Mathematics*, page 109–232. Dekker, 1990.

[48] W. J. Cody. Towards sensible floating-point arithmetic. In *COMPCON (20th: 1980: San Francisco, CA) VLSI, New Architectural Horizons: COMPCON, Spring 80, Jack Tar Hotel, San Francisco, California, February 25–28, 1980: Digest of Papers*, pages 488–490, 1980.

[49] M. Cornea et al. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. In P. Kornerup and J.-M. Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 29–37. IEEE Computer Society Conference Publishing Services, June 2007.

[50] M. Cornea et al. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. *IEEE Transactions on Computers*, 58(2):148–162, 2009.

[51] M. Cornea, R. A. Golliver, and P. Markstein. Correctness proofs outline for Newton–Raphson-based floating-point divide and square root algorithms. In Koren and Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 96–105. IEEE Computer Society Press, Los Alamitos, CA, Apr. 1999.

[52] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium®-based Systems*. Intel Press, Hillsboro, OR, 2002.

[53] M. Cornea, C. Iordache, J. Harrison, and P. Markstein. Integer divide and remainder operations in the IA-64 architecture. In J.-C. Bajard, C. Frougny, P. Kornerup, and J.-M. Muller, editors, *Proceedings of the 4th Conference on Real Numbers and Computers*, 2000.

[54] M. Cornea-Hasegan. Methods and arrangements to correct for double rounding errors when rounding floating point numbers to nearest away, Jan. 2012. US Patent US8095586B2.

[55] M. F. Cowlishaw. Decimal floating-point: algorism for computers. In Bajard and Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic (ARITH-16)*, pages 104–111. IEEE Computer Society Press, Los Alamitos, CA, June 2003.

[56] M. F. Cowlishaw, E. M. Schwarz, R. M. Smith, and C. F. Webb. A decimal floating-point specification. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-15)*, pages 147–154, Vail, CO, June 2001.

[57] C. Daramy-Loirat et al. CR-Libm, A library of correctly rounded elementary functions in double-precision.

[58] E. Darulova and A. Volkova. Sound approximation of programs with elementary functions. *CoRR*, abs/1811.10274, 2018.

[59] H. Dawood. *Theories of Interval Arithmetic: Mathematical Foundations and Applications*. LAP Lambert Academic Publishing, 2011.

[60] K. De Brabandere et al. Design and operation of a phase-locked loop with Kalman estimator-based filter for single-phase applications. In *IECON 2006 - 32nd Annual Conference on IEEE Industrial Electronics*, pages 525–530, Nov 2006.

[61] F. de Dinechin, M. Istoan, and A. Massouri. Sum-of-product architectures computing just right. In *Application-Specific Systems, Architectures and Processors (ASAP)*. IEEE, 2014.

[62] F. de Dinechin, M. Joldes, and B. Pasca. Automatic generation of polynomial-based hardware architectures for function evaluation. In *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

[63] F. de Dinechin, C. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1318–1322, Dijon, France, 2006.

[64] F. de Dinechin, C. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2):242–253, Feb. 2011.

[65] F. de Dinechin, C. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *RAIRO - Theoretical Informatics and Applications*, 41(1):85–102, April 2007.

[66] F. de Dinechin, C. Lauter, J.-M. Muller, and S. Torres. On Ziv's rounding test. *ACM Trans. Math. Softw.*, 39(4):25:1–25:19, July 2013.

[67] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, Mar. 2005.

[68] D. Defour. *Fonctions élémentaires: algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, Sept. 2003.

[69] D. Defour. FP-ANR: A representation format to handle floating-point cancellation at run-time. In *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, pages 76–83, June 2018.

[70] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software*, Beijing, China, 2002. Updated version of LIP research report 2002-08.

[71] T. J. Dekker. A floating point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242, 1971.

[72] J. Detrey. *Arithmétiques réelles sur FPGA : virgule fixe, virgule flottante et système logarithmique*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, Jan. 2007.

[73] J. Detrey and F. de Dinechin. Parameterized floating-point logarithm and exponential functions for FPGAs. *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing*, 31(8):537–545, 2007.

[74] J. Detrey, F. de Dinechin, and X. Pujol. Return of the hardware floating-point elementary function. In *18th Symposium on Computer Arithmetic*, pages 161–168. IEEE, June 2007.

[75] M. Dukhan. PeachPy: A Python framework for developing high-performance assembly kernels. In *Python for High Performance and Scientific Computing*, 2013.

[76] M. Dukhan and R. Vuduc. Methods for high-throughput computation of elementary functions. In *Parallel Processing and Applied Mathematics*, LNCS 8384, 2014.

[77] M. D. Ercegovac and T. Lang. *Digital arithmetic*. Morgan Kaufmann Oxford, San Francisco CA, 2004.

[78] M. A. Erle and M. J. Schulte. Decimal multiplication via carry-save addition. In *Application-specific Systems, Architectures and Processors*, pages 348–355. IEEE, 2003.

[79] A. Farrés Basiana et al. High precision symplectic integrators for the solar system. 2013.

[80] A. Fettweis. Wave digital filters: Theory and practice. *Proc. of the IEEE*, 74(2), 1986.

[81] S. A. Figueroa. When is double rounding innocuous? *ACM SIGNUM Newsletter*, 30(3), July 1995.

[82] S. A. Figueroa. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. PhD thesis, Department of Computer Science, New York University, 2000.

[83] T. Finley. Two's complement. Cornell University Lecture Notes, 2000.

[84] L. Fousse. *Intégration numérique avec erreur bornée en précision arbitraire*. PhD in Computer Science, Université Henri Poincaré – Nancy 1, 2006.

[85] L. Fousse. Multiple-Precision Correctly rounded Newton-Cotes quadrature. *RAIRO Theoretical Informatics and Applications*, 41(2):103–121, 2007.

[86] L. Fousse et al. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[87] L. Fox and I. B. Parker. *Chebyshev polynomials in numerical analysis*. Oxford University Press, 1968.

[88] B. Friedland. *Control Systems Design: An Introduction to State-Space Methods*. McGraw-Hill Higher Education, 1985.

[89] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[90] S. Gal and B. Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, 1991.

[91] M. Gales and S. Young. The application of hidden markov models in speech recognition. *Found. Trends Signal Process.*, 1(3):195–304, Jan. 2007.

[92] S. Gershgorin. Über die Abgrenzung der Eigenwerte einer Matrix. *Bull. Acad. Sci. URSS*, 1931(6):749–754, 1931.

[93] K. R. Ghazi, V. Lefèvre, P. Théveny, and P. Zimmermann. Why and how to use arbitrary precision. *Computing in Science and Engineering*, 12(3):5, 2010.

[94] A. Gil, J. Segura, and N. M. Temme. *Numerical methods for special functions*. SIAM, 2007.

[95] T. Glek and J. Hubicka. Optimizing real world applications with GCC link time optimization. *CoRR*, abs/1010.2196, 2010.

[96] K. Goto and R. A. v. d. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[97] S. Graillat. *Contribution à l'amélioration de la précision et à la validation des algorithmes numériques*. Habilitation à diriger des recherches, Université Pierre et Marie Curie, 2013.

[98] S. Graillat, Y. Ibrahimy, C. Jeangoudoux, and C. Lauter. A parallel compensated Horner scheme. In *Proceedings of ACA 2017, 23rd Conference on Applications of Computer Algebra, Jerusalem, Israel*, page 271, July 2017.

[99] S. Graillat, C. Jeangoudoux, and C. Lauter. A Decimal Multiple-Precision Interval Arithmetic Library. In *17th International Symposium on Scientific Computing, Computer Arithmetics and Verified Numerics*, Uppsala, Sweden, Sept. 2016.

[100] S. Graillat, C. Jeangoudoux, and C. Lauter. MPDI: A Decimal Multiple-Precision Interval Arithmetic Library. *Reliable Computing Journal*, pages 38–52, 2017.

[101] S. Graillat, C. Lauter, P. T. P. Tang, N. Yamanaka, and S. Oishi. Efficient calculations of faithfully rounded l2-norms of n-vectors. *ACM Trans. Math. Softw.*, 41(4):24:1–24:20, Oct. 2015.

[102] T. Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.1 edition, 2016.

[103] L. Grigori. *Introduction to Communication Avoiding Algorithms for Direct Methods of Factorization in Linear Algebra*, pages 153–185. Springer International Publishing, Cham, 2017.

[104] D. Grune et al. *Modern Compiler Design*. Springer Publishing Company, Incorporated, 2nd edition, 2012.

[105] J. A. Gubner. A new series for approximating Voigt functions. *Journal of Physics A: Mathematical and General*, 27(19):L745, 1994.

[106] Gustafson and Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomput. Front. Innov.: Int. J.*, 4(2):71–86, June 2017.

[107] J. Gustafson. *The End of Error: Unum Computing*. Feb. 2015.

[108] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[109] J. Harrison. Decimal transcendentals via binary. In *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH-19)*. IEEE Computer Society Press, June 2009.

[110] Y. Hida, X. S. Li, and D. H. Bailey. Algorithms for quad-double precision floating-point arithmetic. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 155–162, Vail, Colorado, June 2001. IEEE Computer Society Press, Los Alamitos, CA.

[111] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, second edition, 2002.

[112] T. Hilaire, P. Chevrel, and J. Whidborne. A unifying framework for finite wordlength realizations. *IEEE Trans. on Circuits and Systems*, 8(54):1765–1774, 2007.

[113] T. Hilaire and B. Lopez. Reliable implementation of linear filters with fixed-point arithmetic. In *Proc. SiPS*, 2013.

[114] E. Hille. *Ordinary differential equations in the complex domain*. Wiley, 1976. Dover reprint, 1997.

[115] D. Ibrahim and F. Abergel. Non-linear filtering and optimal investment under partial information for stochastic volatility models. *Mathematical Methods of Operations Research*, 87(3):311–346, Jun 2018.

[116] IEEE Computer Society. *IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985*. IEEE Standard 754-1985, 1985.

[117] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, Aug. 2008.

[118] IEEE Computer Society. *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX) Base Specifications*. Dec. 2017.

[119] V. Innocente. Floating point in experimental HEP data processing. In *2nd CERN Openlab/INTEL Workshop on Numerical Computing*, 2012.

[120] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Instruction Set Reference, A-Z, Vol. 2*.

[121] A. Ioualalen and M. Martel. Synthesizing accurate floating-point formulas. In *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, pages 113–116, 2013.

[122] ISO/IEC. *International Standard ISO/IEC 9899:1999(E). Programming languages – C*. 1999.

[123] ISO/IEC. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011.

[124] ISO/IEC JTC 1/SC 22/WG 14. Extension for the programming language C to support decimal floating-point arithmetic. Proposed Draft Technical Report, May 2008.

[125] C. Jeangoudoux and C. Lauter. A correctly rounded mixed-radix fused-multiply-add. In *2018 IEEE 25th Symposium on Computer Arithmetic*, pages 17–24, July 2018.

[126] C.-P. Jeannerod et al. A new binary floating-point division algorithm and its software implementation on the ST231 processor. In J. Bruguera, M. Cornea, D. DasSarma, and J. Harrison, editors, *Proceedings of the 19th IEEE Symposium on Computer Arithmetic (ARITH'19)*, pages 95–103, Portland, OR, USA, June 2009. IEEE Computer Society.

[127] C.-P. Jeannerod, J. Jourdan-Lu, C. Monat, and G. Revy. How to Square Floats Accurately and Efficiently on the ST231 Integer Processor. In *20th IEEE Symposium on Computer Arithmetic (ARITH)*, pages 77–81, Tübingen, Germany, Aug. 2011.

[128] F. Johansson. Arb: A C library for ball arithmetic. *ACM Commun. Comput. Algebra*, 47(3/4):166–169, Jan. 2014.

[129] S. G. Johnson. Faddeeva package, 2012.

[130] M. Joldes, C. Lauter, M. Ceberio, O. Kosheleva, and V. Kreinovich. Why Taylor models and modified Taylor models are empirically successful : A symmetry-based explanation. In *Proceedings of the 8th International Workshop on Reliable Engineering Computing REC'2018*, July 2018. Proceedings published online.

[131] M. Joldeş, O. Marty, J. Muller, and V. Popescu. Arithmetic algorithms for extended precision using floating-point expansions. *IEEE Transactions on Computers*, 65(4):1197–1210, April 2016.

[132] W. Kahan. Floating-point arithmetic besieged by business decisions, (invited keynote address). In *2005 IEEE 17th Symposium on Computer Arithmetic*, June 2005.

[133] T. Kailath. *Linear Systems*. Prentice-Hall, 1980.

[134] N. Kapre and A. DeHon. Accelerating SPICE model-evaluation using FPGAs. *Field-Programmable Custom Computing Machines*, pages 37–44, 2009.

[135] A. Y. Khinchin. *Continued Fractions*. Dover, New York, 1997.

[136] S. Kim, K. Kum, and W. Sung. Fixed-point optimization utility for C and C++ based digital signal processing programs. *IEEE Transactions on Circuits and Systems*, 45(11):1455–1464, November 1998.

[137] R. Kirchner and U. Kulisch. Accurate arithmetic for vector processors. *Journal of Parallel and distributed Computing*, Special Issue on Parallelism in Computer Arithmetic(5), 1988.

[138] D. E. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison-Wesley, Reading, MA, USA, third edition, 1998.

[139] P. Kornerup, C. Lauter, V. Lefèvre, N. Louvet, and J.-M. Muller. Computing correctly rounded integer powers in floating-point arithmetic. *ACM Trans. Math. Softw.*, 37(1):4:1–4:23, Jan. 2010.

[140] P. Kornerup, V. Lefevre, N. Louvet, and J. M. Muller. On the computation of correctly rounded sums. *IEEE Transactions on Computers*, 61(3):289–298, March 2012.

[141] U. Kulisch and V. Snyder. The exact dot product as basic tool for long interval arithmetic. *Computing*, 91(3):307–313, Mar. 2011.

[142] U. W. Kulisch. Circuitry for generating scalar products and sums of floating-point numbers with maximum accuracy. United States Patent 4622650, 1986.

[143] U. W. Kulisch. *Advanced Arithmetic for the Digital Computer: Design of Arithmetic Units*. Springer-Verlag, Berlin, 2002.

[144] U. W. Kulisch. *Computer Arithmetic and Validity: Theory, Implementation, and Applications*. de Gruyter, Berlin, 2008.

[145] O. Kupriianova. *Towards a Modern Floating-Point Environment*. PhD thesis, Université Pierre et Marie Curie - Paris VI, 2015.

[146] O. Kupriianova and C. Lauter. Metalibm. GNU Cauldron 2013, Google Headquarters, Montainview, CA, June 2013.

[147] O. Kupriianova and C. Lauter. A domain splitting algorithm for the mathematical functions code generator. In *2014 48th Asilomar Conference on Signals, Systems and Computers*, pages 1271–1275, Nov 2014.

[148] O. Kupriianova and C. Lauter. Metalibm: A mathematical functions code generator. In H. Hong and C. Yap, editors, *ICMS 2014*, volume 8592 of *LNCS*, pages 713–717. Springer, 2014.

[149] O. Kupriianova and C. Lauter. *Replacing Branches by Polynomials in Vectorizable Elementary Functions*, pages 14–22. Springer International Publishing, Würzburg, 2016.

[150] O. Kupriianova, C. Lauter, and J. M. Muller. Radix conversion for IEEE754-2008 mixed radix floating-point arithmetic. In *2013 Asilomar Conference on Signals, Systems and Computers*, pages 1134–1138, Nov 2013.

[151] M. Labbate, R. Petrella, and M. Tursini. Fixed point implementation of Kalman filtering for AC drives: a case study using TMS320F24x DSP. Feb. 2019.

[152] J. Laskar. High precision astronomical solution for paleoclimate studies. In *EGU General Assembly Conference Abstracts*, volume 15, page 11302, 2013.

[153] J. Laskar et al. A long-term numerical solution for the insolation quantities of the earth. *Astronomy & Astrophysics*, 428(1):261–285, 2004.

[154] C. Lauter. A survey of multiple-precision using floating-point arithmetic. In *Fourth International Workshop on Taylor Models*, Boca Raton, FL, Dec. 2006.

[155] C. Lauter. Sollya - a numerical software tool for the semi-automatic implementation of efficient correctly rounded mathematical functions. In *Invited presentation at ACA2008*, Hagenberg, Austria, July 2008.

[156] C. Lauter. Easing development of precision-sensitive applications with a beyond-quad-precision library. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 742–746, Nov 2015.

[157] C. Lauter. A new open-source SIMD vector libm fully implemented with high-level scalar C. In *2016 50th Asilomar Conference on Signals, Systems and Computers*, pages 407–411, Nov 2016.

[158] C. Lauter. An efficient software implementation of correctly rounded operations extending FMA: $a + b + c$ and $a \times b + c \times d$. In *2017 51st Asilomar Conference on Signals, Systems and Computers*, pages 452–456, 2017.

[159] C. Lauter. Rigorous polynomial approximation. In *2018 52nd Asilomar Conference on Signals, Systems and Computers*, pages 120–124, 2018.

[160] C. Lauter and F. de Dinechin. Optimizing polynomials for floating-point implementation. In J. D. Bruguera and M. Daumas, editors, *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 7–16, Santiago de Compostela, Espagne, July 2008.

[161] C. Lauter and O. A. Kupriianova. The libieee754 compliance library for the IEEE 754-2008 standard. In *The 15'th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations, 2012*, Novosibirsk, Russia, Sept. 2012.

[162] C. Lauter and V. Ménissier-Morain. There's no reliable computing without reliable access to rounding modes. In *The 15'th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Verified Numerical Computations, 2012*, Novosibirsk, Russia, Sept. 2012.

[163] C. Lauter and M. Mezzarobba. Semi-automatic floating-point implementation of special functions. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 58–65, June 2015.

[164] C. Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR-5702, INRIA, Sept. 2005.

[165] C. Q. Lauter. *Arrondi Correct de Fonctions Mathématiques*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, Oct. 2008.

[166] C. Q. Lauter and V. Lefèvre. An efficient rounding boundary test for pow(x,y) in double precision. *IEEE Transactions on Computers*, 58(2):197–207, Feb 2009.

[167] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, Sept. 1979.

[168] D.-U. Lee, P. Cheung, W. Luk, and J. Villasenor. Hierarchical segmentation schemes for function evaluation. *IEEE Transactions on VLSI Systems*, 17(1), 2009.

[169] D. Lefebvre, P. Chevrel, and S. Richard. An $H_\infty$ based control design methodology dedicated to the active control of longitudinal oscillations. *IEEE Trans. on Control Systems Technology*, 11(6):948–956, 2003.

[170] V. Lefèvre. *Developments in Reliable Computing*, chapter An Algorithm that Computes a Lower Bound on the Distance Between a Segment and $\mathbb{Z}^2$, pages 203–212. Kluwer Academic Publishers, Dordrecht, 1999.

[171] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[172] V. Lefèvre. New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic (ARITH-17)*, pages 68–75. IEEE Computer Society Press, Los Alamitos, CA, June 2005.

[173] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, Vail, CO, June 2001.

[174] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1997.

[175] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, Nov. 1998.

[176] K. Lei and Y. Xiao-Ying. Design and implementation for quadruple precision floating-point multiplier based on FPGA with lower resource occupancy. In *2014 Fifth International Conference on Intelligent Systems Design and Engineering Applications*, pages 326–329, June 2014.

[177] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, Apr. 2001.

[178] X. S. Li et al. Design, implementation and testing of extended and mixed precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

[179] S. Loosemore et al. *The GNU C Library Reference Manual*. Free Software Foundation, Inc.

[180] B. Lopez, T. Hilaire, and L. S. Didier. Formatting bits to better implement signal processing algorithms. In *Proc.PECCS, Portugal*, pages 104–111, 2014.

[181] J. Lopez, C. Carreras, and O. Nieto-Taladriz. Improved interval-based characterization of fixed-point LTI systems with feedback loops. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(11):1923–1933, 2007.

[182] N. Louvet. *Compensated algorithms in floating point arithmetic : accuracy, validation, performances*. PhD thesis, Université de Perpignan Via Domitia, Nov. 2007.

[183] D. Lutz and N. Burgess. Overcoming double-rounding errors under IEEE 754-2008 using software. In *Signals, Systems and Computers (ASILOMAR), 2010 Conference Record of the Forty Fourth Asilomar Conference on*, pages 1399–1401, Nov 2010.

[184] D. R. Lutz. Fused multiply-add microarchitecture comprising separate early-normalizing multiply and add pipelines. In *IEEE Symposium on Computer Arithmetic*, pages 123–128, 2011.

[185] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[186] P. Markstein. Accelerating sine and cosine evaluation with compiler assistance. In *16th Symposium on Computer Arithmetic*, pages 137–140. IEEE, 2003.

[187] J. H. Mathews. Bibliography for Taylor series method for D.E.'s, 2003.

[188] G. Melquiond. *De l'arithmétique d'intervalles à la certification de programmes*. PhD thesis, ÉNS Lyon, 2006.

[189] M. Mezzarobba. NumGfun: a package for numerical and analytic computation with D-finite functions. In S. M. Watt, editor, *ISSAC '10*, page 139–146. ACM, 2010.

[190] M. Mezzarobba. *Autour de l'évaluation numérique des fonctions D-finies*. Thèse de doctorat, École polytechnique, 2011.

[191] M. Mezzarobba and B. Salvy. Effective bounds for P-recursive sequences. *Journal of Symbolic Computation*, 45(10):1075–1096, 2010.

[192] O. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.

[193] S. L. Moshier. Cephes mathematical function library, 1984.

[194] C. Mouilleron and G. Revy. Automatic Generation of Fast and Certified Code for Polynomial Evaluation. In *20th Symposium on Computer Arithmetic* , pages 233–242, Aug. 2011.

[195] J.-M. Muller. On the definition of $ulp(x)$. Technical Report RR-5504, INRIA, Feb. 2005.

[196] J.-M. Muller. *Elementary Functions: Algorithms and Implementation (2nd edition)*. Birkhauser, 2006.

[197] J.-M. Muller et al. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Boston, 2010.

[198] M. Neher, K. R. Jackson, and N. S. Nedialkov. On Taylor model based integration of ODEs. *SIAM Journal on Numerical Analysis*, 45(1):236–262, 2007.

[199] J. v. Newmann. First draft of a report on the EDVAC. Technical report, 1945.

[200] T. Ogita, S. M. Rump, and S. Oishi. Accurate sum and dot product. *SIAM Journal on Scientific Computing (SISC)*, 2005.

[201] T. Ogita, S. M. Rump, and S. Oishi. Verified solutions of linear systems without directed rounding. Technical Report 2005-04, Advanced Research Institute for Science and Engineering, Waseda University, Tokyo, Japan, 2005.

[202] F. W. J. Olver. *Asymptotics and special functions*. A K Peters, 1997.

[203] A. V. Oppenheim and R. W. Schafer. *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

[204] V. Pan and J. Reif. Efficient parallel solution of linear systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 143–152. ACM, 1985.

[205] T. Parks and J. McClellan. Chebyshev approximation for nonrecursive digital filters with linear phase. *IEEE Transactions on Circuit Theory*, 19(2):189–194, March 1972.

[206] G. Paul and M. W. Wilson. Should the elementary functions be incorporated into computer instruction sets? *ACM Transactions on Mathematical Software*, 2(2):132–142, 1976.

[207] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18:19–24, 1983.

[208] O. Perron. *Die Lehre von den Kettenbrüchen*. Teubner, Stuttgart, 3rd edition, 1954–57.

[209] D. Piparo. The VDT mathematical library. In *2nd CERN Openlab/INTEL Workshop on Numerical Computing*, 2012.

[210] D. Piparo and V. Innocente. The CptnHook profiler - a tool to investigate usage patterns of mathematical functions. *J. Phys.: Conf. Ser.*, 762(1):012038. 5 p, 2016.

[211] D. Piparo, V. Innocente, and T. Hauth. Speeding up HEP experiment software with a library of fast and auto-vectorisable mathematical functions. *Journal of Physics: Conference Series*, 513(5):052027, 2014.

[212] M. Püschel et al. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[213] B. Randell. From analytical engine to electronic digital computer: The contributions of Ludgate, Torres, and Bush. *Annals of the History of Computing*, 4(4):327–341, Oct 1982.

[214] E. Remez. Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation. *Comptes Rendus de l'Académie des Sciences, Paris*, 198:2063–2065, 1934.

[215] Е. Я. Ремез (E. Ya. Remez). Основы численных методов чебышевского приближения. Академия Наук Украинской ССР, Институт математики, Наукова Думка, Kiev, 1969.

[216] N. Revol. Multiple precision interval arithmetic and application to linear systems. In *International Conference on Applications of Computer Algebra*, 2003.

[217] N. Revol and F. Rouillier. Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library. *Reliable Computing*, 11(4):275–290, 2005.

[218] G. Revy. *Implementation of binary floating-point arithmetic on embedded integer processors*. PhD thesis, ÉNS Lyon, 2009.

[219] G. Revy. Automated design of floating-point logarithm functions on integer processors. In *23rd IEEE Symposium on Computer Arithmetic*, pages 172–180, 2016.

[220] E. J. Riedy and J. Demmel. Augmented arithmetic operations proposed for IEEE-754 2018. In *25th IEEE Symposium on Computer Arithmetic, ARITH 2018, Amherst, MA, USA, June 25-27, 2018*, pages 45–52, 2018.

[221] R. Rojas. Konrad Zuse: 1910-2010. In S. Conrad and P. Molitor, editors, *it - Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik*, volume 52. De Gruyter.

[222] C. Rubio-González et al. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2013.

[223] S. M. Rump. New results on verified inclusions. In *Accurate Scientific Computations, Symposium, 1985, Proceedings*, pages 31–69, 1985.

[224] S. M. Rump. Solution of linear systems with verified accuracy. *Applied numerical mathematics*, 3(3):233–241, 1987.

[225] S. M. Rump. Algorithms for verified inclusion. In R. Moore, editor, *Reliability in Computing, Perspectives in Computing*, pages 109–126. Academic Press, New York, 1988.

[226] S. M. Rump. Reliability in computing: The role of interval methods in scientific computing. pages 109–126. Academic Press, 1988.

[227] S. M. Rump. Guaranteed inclusions for the complex generalized Eigenproblem. *Computing*, 42(2-3):225–238, Sept. 1989.

[228] S. M. Rump. Ultimately fast accurate summation. *SIAM J. Sci. Comput.*, 31(5):3466–3502, 2009.

[229] S. M. Rump. Error estimation of floating-point summation and dot product. *BIT Numerical Mathematics*, 52(1):201–220, Mar 2012.

[230] S. M. Rump, T. Ogita, Y. Morikura, and S. Oishi. Interval arithmetic with fixed rounding mode. *Nonlinear Theory and Its Applications, IEICE*, 7(3):362–373, 2016.

[231] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation. I. Faithful rounding. *SIAM J. Sci. Comput.*, 31(1):189–224, 2008.

[232] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation. II. Sign, $K$-fold faithful and rounding to nearest. *SIAM J. Sci. Comput.*, 31(2):1269–1302, 2008/09.

[233] M. Sadeghian, J. E. Stine, and E. G. Walters. Optimized linear, quadratic and cubic interpolators for elementary function hardware implementations. *Electronics*, 5(2), 2016.

[234] F. Schreier. The Voigt and complex error function: a comparison of computational methods. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 48(5):743–762, 1992.

[235] S. P. Shary. Solving the linear interval tolerance problem. *Mathematics and Computers in Simulation*, 39(1):53 – 85, 1995.

[236] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.

[237] B. Solomon et al. Micro-operation cache: A power aware frontend for the variable instruction length ISA. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ISLPED '01, pages 4–9, New York, NY, USA, 2001. ACM.

[238] P. H. Sterbenz. *Floating-Point Computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[239] P. T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989.

[240] P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378 – 400, 1990.

[241] The GNU project. GNU compiler collection, 1987–2014.

[242] D. Thomas. A general-purpose method for faithfully rounded floating-point function approximation in FPGAs. In *Computer Arithmetic (ARITH), 2015 IEEE 22nd Symposium on*, pages 42–49, June 2015.

[243] W. J. Thompson et al. Numerous neat algorithms for the Voigt profile function. *Computers in Physics*, 7(6):627–631, 1993.

[244] C. Tsen, S. González-Navarro, and M. Schulte. Hardware design of a binary integer decimal-based floating-point adder. In *ICCD 2007. 25th International Conference on Computer Design*, pages 288–295. IEEE, 2007.

[245] J. van der Hoeven. Fast evaluation of holonomic functions. *Theoretical Computer Science*, 210(1):199–216, 1999.

[246] J. van der Hoeven. Fast evaluation of holonomic functions near and in regular singularities. *Journal of Symbolic Computation*, 31(6):717–743, 2001.

[247] G. W. Veltkamp. ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie. Technical Report 22, RC-Informatie, Technische Hogeschool Eindhoven, 1968.

[248] G. W. Veltkamp. ALGOL procedures voor het rekenen in dubbele lengte. Technical Report 21, RC-Informatie, Technische Hogeschool Eindhoven, 1969.

[249] N. Veyrat-Charvillon. *Opérateurs arithmétiques matériels pour des applications spécifiques*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, June 2007.

[250] A. Volkova. *Towards reliable implementation of digital filters*. PhD thesis, Université Pierre et Marie Curie - Paris VI, Sept. 2017.

[251] A. Volkova, T. Hilaire, and C. Lauter. Determining fixed-point formats for a digital filter implementation using the worst-case peak gain measure. In *2015 49th Asilomar Conference on Signals, Systems and Computers*, pages 737–741, Nov 2015.

[252] A. Volkova, T. Hilaire, and C. Lauter. Reliable evaluation of the worst-case peak gain matrix in multiple precision. In *2015 IEEE 22nd Symposium on Computer Arithmetic*, pages 96–103, June 2015.

[253] A. Volkova, T. Hilaire, and C. Lauter. Reliable verification of digital implemented filters against frequency specifications. In *2017 IEEE 24th Symposium on Computer Arithmetic*, pages 180–187, July 2017.

[254] A. Volkova, T. Hilaire, C. Lauter, and M. Mezzarobba. Rigorous determination of recursive filter fixed-point implementation with input signal frequency specifications. In *2017 51st Asilomar Conference on Signals, Systems and Computers*, 2017.

[255] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, 1999.

[256] L.-K. Wang and M. J. Schulte. Decimal floating-point division using Newton–Raphson iteration. In *Application-Specific Systems, Architectures and Processors*, pages 84–95. IEEE, 2004.

[257] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, Feb. 2005.

[258] N. Whitehead and A. Fit-Florea. Floating point and IEEE-754 compliance for NVIDIA GPUs. Nvidia Whitepaper.

[259] A. Wittig and M. Berz. Rigorous high precision interval arithmetic in COSY INFINITY. *Proceedings of the Fields Institute*, 2009.

[260] H. Zhang, W. Zhang, and J. Lach. A low-power accuracy-configurable floating point multiplier. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 48–54, Oct 2014.

[261] Z. Zhao and G. Li. Roundoff noise analysis of two efficient digital filter structures. *IEEE Trans. on Signal Processing*, 54(2):790–795, 2006.

[262] Y.-K. Zhu and W. B. Hayes. Correct rounding and a hybrid approach to exact floating-point summation. *SIAM J. Sci. Comput.*, 31(4):2981–3001, 2009.

[263] Y.-K. Zhu and W. B. Hayes. Algorithm 908: Online exact summation of floating-point streams. *ACM Trans. Math. Softw.*, 37(3), 2010.

[264] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, Sept. 1991.

## Résumé :

Ce travail commence par une analyse succincte de la norme IEEE754 sur l'arithmétique flottante, dans sa version de 2008. Une comparaison des opérations requises par la norme avec la liste de celles implantées dans l'environnement flottant d'un système actuel montre que certaines opérations, par exemple celles qui sont hétérogènes dans leurs arguments, manquent encore. Des algorithmes sont proposés pour ces problèmes toujours ouverts. Ce travail part ensuite de cette analyse pour proposer des extensions possibles futures de la norme. Celles-ci sont diverses : des ajouts comme la précision étendu ou bien des opérations plus haut niveau comme des normes euclidiennes ainsi que des opérations qui permettraient de mélanger des nombres en virgule flottante des bases binaire et décimales.

Ce manuscrit essaie ensuite de passer du niveau de codes numériques manuellement écrits à des générateurs de codes numériques. Cette approche meta-arithmétique est illustrée à l'exemple de fonctions mathématiques comme $\log, \sin, \cos$ ou encore spéciales, définies par des équations différentielles. Les techniques proposées montrent que la génération de codes numériques est un moyen efficace pour étendre l'environnement flottant moderne.

Enfin, ce travail considère des algorithmes en virgule flottante fiable, garantissant une précision en sortie spécifiée *a priori*. Ces algorithmes sont ensuite exploités pour la génération de codes en virgule fixe implantant des filtres linéaires invariants dans le temps.

## Mots-clefs :

Arithmétique en virgule flottante et en virgule fixe, précision étendue et arbitraire, arithmétique en base mixte, arrondis correct et fidèle, fonctions mathématiques, génération de code, Metalibm, découpage en sous-domaines, filtres LTI, matrice Worst Case Peak Gain

## Abstract:

This works starts with shedding some light on the 2008 version of the IEEE754 Standard for Floating-Point Arithmetic and its realization in the Floating-Point environment offered by current systems. Analyzing the parts of the Standard that are still missing, such as support for heterogeneous operations, it proposes algorithmic solutions to these open issues. The work then extends these considerations to possible future extensions to the Standard. These extensions come in manifold aspects: classical enhancements such as extended precision or higher level operations such as faithfully rounded Euclidian Norms as well as operations that allow the binary and decimal floating-point formats to be mixed.

It then tries to pass from the level of manually written numerical codes to code generation of numerical codes. These meta-arithmetic approaches are studied at the example of mathematical functions such as $\log, \sin, \cos$ or even special functions, defined by differential equations. Code generation is shown to be an effective way of providing a modern floating-point environment.

Finally, this work looks into reliable floating-point algorithms, providing *a priori* error guarantees on their output. These algorithms are utilized as a convenience for the generation of fixed-point codes for the implementation of Linear Time Invariant Filters.

## Keywords:

Floating-point and Fixed-Point Arithmetic, Extended and Multiple Precision, Mixed-Radix Arithmetic, Correct and Faithful Rounding, Mathematical Functions, Code Generation, Metalibm, Domain Splitting, Linear Time Invariant Filters, Worst Case Peak Gain