Towards a modern floating-point environment

Olga Kupriianova

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Olga KUPRIIANOVA

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Towards a Modern Floating-Point Environment

version corrigée

soutenue le 11 décembre 2015

devant le jury composé de :

M.	Jean-Claude BAJARD	Directeur de thèse
Mme	Sylvie BOLDO	Rapporteur
M.	Jean-Marie CHESNEAUX	Examinateur
M.	Laurent-Stéphane DIDIER	Examinateur
M.	Florent de DINECHIN	Examinateur
M.	Philippe LANGLOIS	Rapporteur
M.	Christoph LAUTER	Encadrant de thèse

Acknowledgements

First of all I want to thank Christoph for all his time and forces that he devoted to my guidance in the field of Floating-Point Arithmetic. I am grateful for giving me such a wonderful possibility to come to Paris first for a short internship and then for thesis; for his endless help with filling long and sometimes annoying administrative forms, for translating and explaining cultural differences and being an encouraging example of a multilingual fluent speaker. I am also grateful for forcing me submit papers to different conferences, and discovering new places due to scientific travelling. Thanks to my another advisor, Jean-Claude, for useful advices and for forcing me in improvement of my French by telling untranslatable jokes. I want to thank also all the members of PEQUAN team for their reception and kindness to have me these three years, attend my seminars and GdT, give useful comments on my presentations that helped me to improve; for helping to get over the language barrier. I'm also grateful for Aric team at ENS-Lyon, the great team in Floating-Point arithmetic, who helped me a lot with the defense presentation preparation. Thanks for everyone from my thesis committee to accept this role, especially Philippe and Sylvie for being kind and patient with the delay of my thesis submission. I want to mention also the big delegation of courageous French researchers that were not scared to go for a conference to Syberia. For me it is a remarkable conference not only because of the troubles and surprises happening to foreigners in Russia, nor because of making French people eat "Napoleon", it was my first conference far away from home.

I want to express my considerable appreciation to my family too: my father who formed my base of mathematical mind helping with homeworks in maths and physics when I was a kid; my mother who contributed the most in my English knowledge in my childhood; my sister, for being an example for me in schooldays and for "opening the window to Europe" for me in my 18. I'd say all this was a great contribution to my thesis. And of course thanks for coming to my defense and helping with its organization.

I want to mention also Cité Universitaire which is probably the greatest place to stay in Paris during the studies, the place where I met most of my "parisian" friends. Thanks to my numerous "Russian mafia" especially Anya, Katya, Ira, Andrey and the crowd from Télécom for their sincere warm friendship, support and help. I am sure we'll have more funny борщ-eating or balloon parties and unforgettable trips

like the one to Barcelona! And a final huge thanks to Oscar for those long but motivating Skypes from other part of the world during my writing period and for his help and support during my trips to Paris.

Contents

List of Figures

List of Tables

Introduction

Many who have never had an opportunity of knowing any more about mathematics confound it with arithmetic, and consider it an arid science. In reality, however, it is a science which requires a great amount of imagination.

Sofia Kovalevskaya[1]

This is a work in the domain of Computer Arithmetic. This is neither 100% Mathematics nor pure Computer Science. The subject of Computer Arithmetic was mostly developed and demanded with the appearing and spreading of modern computers. However, the part of quote by S. Kovalevskaya *"it is a science which requires a great amount of imagination"* concerns not only Mathematics but Computer Arithmetic as well.

Floating-point Issues

We learn arithmetics during our first years at school: we start with basic operations on integer numbers $(+, -, \times, /)$, then we learn real numbers, fractions, elementary functions, matrices and operations on them. Arithmetic or arithmetics (in Greek means number) is a branch of mathematics that studies numbers, their properties and operations on them. Thus, computer arithmetic studies all these operations and computations on machine's numbers.

Mathematicians use real number ($\mathbb{R}$) to model different processes in various fields: finance, physics, engineering, space mechanics, etc. As there are irrational numbers in the set $\mathbb{R}$, not all real numbers are representable in machines. In fact, computations are held on so-called floating-point (FP) numbers. This is a discrete set of numbers that model $\mathbb{R}$. As the length of computer-stored number is limited the real numbers are somehow rounded, e.g. values for π, e. These roundings cannot be neglected as they may bring serious errors. Each machine operation gives a rounded result, so is

[1] Sofia Kovalevskaya (1850 - 1891) is the first major Russian female mathematician and the first woman appointed to a full professorship in Northern Europe (Stockholm).

executed with an error. Therefore in order to be sure in the results, serious analysis is often required.

There are some famous FP bugs that led to serious consequences. For example, in June 1994 a bug with FP division on some models of Intel's Pentium processor was discovered [21]. It gave only four correct digits as a result for this operation:

$$\frac{4\,195\,835}{3\,145\,727} = 1.333739068902037589,$$

while the correct value is

$$\frac{4\,195\,835}{3\,145\,727} = 1.333820449136241002$$

In December 1994 Intel decided to replace all flawed Pentium processors, and defective chips were given to the employees as key chains. This story costed Intel about $475 million. In 1996 the first test flight of Ariane 5 rocket finished by crash in 37 seconds after the launch [60]. The reason of crash was a software bug: 64-bit floating point value converted to 16-bit signed integer was too large to fit into destination format.

Thus, software bugs and small rounding errors accumulated in several basic FP operations during computing of some result may cause serious problems. Applied research has interest in reliable results which can be obtained with some tricky computation techniques. Computer Arithmetic is not only about basic arithmetic operations $(+, -, \times, /)$, but about all the computations in FP.

FP Numbers and Standards

Roughly speaking, FP numbers are the numbers of the form

$$\pm m_0 m_1 m_2 \dots m_{k-1} \cdot \beta^E,$$

where all the $m_i \in \mathbb{Z}, 0 \le m_i \le \beta - 1$, the integer number $m_0 m_1 m_2 \dots m_{k-1}$ is called mantissa or significand, E exponent and $\beta \in \mathbb{Z}$ is radix. The term "floating" relates to fraction point: it may be placed anywhere in significand digits in number. For instance, $12345 \cdot 10^{-1} = 1234.5$ and $12345 \cdot 10^{-5} = 0.12345$. Since 1985 we have the IEEE754 Standard for FP arithmetic [42]. It declared how to store and operate binary FP numbers. The basic arithmetic operations were mandatory, as well as comparisons and conversion to other formats, to and from "human-readable" decimal string representation.

Computers use binary numbers, while people are more used to operate in decimal. An important problem in financial computations comes from radix conversion: finite decimal number 0.1 is infinite in binary and therefore rounded. This rounding error may influence the final result. Thus, there is a need in decimal FP arithmetic. The

IEEE854 Standard published in 1987 was an attempt to unify decimal and binary arithmetic [44]. However, it did not require any operations or formats as IEEE754 and was never implemented. Since 1985 hardware and software evolved, new operations appeared, programming languages supported new operations. Some of them, as fused multiply-add (FMA) or heterogeneous arithmetic operations were fundamental for Computer Arithmetic. Therefore, in 2008 IEEE754 revision appeared: it added decimal FP numbers, allowed to mix precisions in one operation (heterogeneous operations) and required an FMA. It also contained recommendation chapter on correctly-rounded mathematical functions (exp, log, sin, etc).

Mathematical Functions Implementations

The main problem in implementation of mathematical functions is that their values are transcendental, so they have infinite number of digits after fraction point, e.g.

$$e^2 = 7.389056098930650227230427460575007813180315570\ldots$$

So, if we want a correct result up to some quantity of digits after a point how can we get it? All the numbers that we can compute are only approximations of the real transcendental result. Roughly speaking this is a description of the so-called Table Maker's dilemma that has to be solved to obtain correctly-rounded results. This is not a trivial task and its solution may require the use of high-precise numbers or long computational time [64, 87]. So the 2008 revision of the Standard only recommends correctly-rounded mathematical functions but does not require them.

The evaluation of mathematical functions is not a new problem and there are already several libraries containing implementations of various functions: Intel's MKL [2], ARM's mathlib [4], Sun's libmcr, GNU glibc libm [32], CRlibm by ENS Lyon [28], etc. They differ for instance by platforms, implementation language, results accuracy. Hardware manufacturers spend a lot of resources on optimization of their libraries for any new processor. Open-Source versions may lag behind in performance all the more 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 [87] (now in the GNU glibc) and CRLibm [26]. All the existing mathematical libraries (they are often called libms) are static: they contain only one implementation per function. Modern libms have to provide several versions of each mathematical function in some precision and a mechanism of choosing the appropriate one. For example, some may be interested in "relatively slow" implementations and correct results, others in fast and "dirty" functions. Till today there was no mean to use non-standard implementations of mathematical functions and those who really needed them were obliged to write their own code.

We tried to estimate the quantity of all the possible choices for a libm, this number is tremendously high and tends to infinity. Therefore, libms cannot stay static, we propose to get function implementations "on demand" for some particular specification. This task may be solved with a code generator for such flexible function implementations. It should take a set of parameters and produce ready-to-use code.

Mixing the Decimal and Binary Worlds

Besides recommendations on mathematical functions IEEE745-2008 brought decimal arithmetic and several new operations for binary. The worlds of decimal and binary FP numbers intersect only in conversion operation: there must be a conversion from binary FP formats to decimal FP numbers and one back, as well as conversion of user's input FP number in character sequence to FP format and back. The last conversion existed also in first version of the standard and is known as a scanf/printf operation in C programming language.

The 2008 revision allows us to mix numbers of different precisions within one operation requiring so-called heterogeneous operations. However, there is no possibility to mix numbers of different radices. As we have today for example machine-stored binary constants and decimal numbers stored in banking databases, it becomes more common to mix decimal and binary FP numbers in code. Thus, we use radix conversions (that produce rounded results) and then use homogeneous or heterogeneous operations. Recent papers like [9] contain research on the first mixed-radix operation, i.e. comparison. Mixed-radix operations might gain in accuracy and performance in comparison with execution of conversion and the needed operation. The 2008 revision added about 280 operations mixing the precision, so it is quite natural to assume one of the following revisions of the Standard adds even more operations mixing numbers of different radices. Besides an attempt to implement mixed-radix comparison there is a paper on atomic operation of radix conversion for IEEE754 mixed-radix arithmetic [55]. As FMA operation is now required, feasibility of its implementation is a good start for research in mixed-radix arithmetic operations. Implemented FMA gives addition, subtraction and multiplication. There are also some algorithms for division and square root based on FMA [46, 61, 84].

Besides absence of mixed-radix operations, conversion from decimal character sequence to binary FP number can be noticeably improved. This is about analogue of scanf function in C language. Current implementations in glibc work only for one rounding mode, while the Standard defines four of them (revision in 2008 added fifth, optional for binary), it allocates memory and therefore it is not re-entrant. We propose a novel algorithm independent of the current rounding mode. Its memory consumption is known beforehand, so this can be reused for embedded systems.

Contribution to the Research Field and Outline

This work has two main directions: the first one is code generation of parametrized mathematical libraries and the second one is research in mixed-radix operations. Both topics belong to the idea of improvement of the current FP environment with the support of more operations. We provide all the needed definitions and background as well as an overview of existing methods in Chapter 1. Chapter 2 is devoted to the research in function implementations and their automatic generation. We first cover in Section 2.1 the basic steps in manual implementation of mathematical functions that are essential to know how to create the code generator. Generally, there are three basic steps to implement a mathematical function: argument reduction (Section 2.1.1), approximation (Section 2.1.2) and reconstruction (Section 2.1.3). All these steps are different for different functions, and the challenge for the code generator is to execute them automatically. In Section 2.2 we present our code generator highlighting its most important aspects. We perform detection of algebraic properties of the function to reduce properly the argument (Section 2.2.3), then additional reduction through domain splitting may be executed. We decrease the number of split subdomains and get an arbitrary splitting that depends on the function characteristics. This algorithm is explained in Section 2.2.4. One of the goals for our code generator is to produce vectorizable code, that may be tricky with arbitrary domain splitting. In Section 2.2.5 we propose a method to replace branching for domain decomposition by polynomial function. Particular algorithms for this code generator were presented on internationally recognized conferences [51, 53, 54]. The entire work on generator [12] earned "best paper award" on 22nd IEEE Symposium on Computer Arithmetic ARITH22 in June 2015.

The second part of the work (Chapter 3) is devoted to research in mixed-radix arithmetic. We start with the search of worst cases for mixed-radix FMA operation in Section 3.4. This gives an estimation for hardness of its implementation. Then we present our novel radix conversion algorithm [55] in Section 3.2. The ideas proposed in that section are reused for the new algorithm on conversion from decimal character sequence to a binary FP number, so the analogue for scanf function. Our improved algorithm for scanf [51] is presented in Section 3.3. Less publications were done in the field of mixed-radix arithmetic research, they are planned to be submitted after thesis defense.

CHAPTER 1

State of the Art

As integer numbers are not sufficient in modeling and computation of various processes, and the infinitely precise real numbers are not representable in computers, floating-point (FP) numbers were invented as a machine model for real numbers. Not all real numbers can be represented in FP and are therefore rounded:

$$\pi = 3.1415926535\ldots$$

The classical way of manipulating, rounding and storing of FP numbers is specified in IEEE754 Standard [42] and is overviewed in this chapter. We explain here basic bricks of FP arithmetic, provide the state of the art for subjects covered by this thesis: elementary functions implementation and mixed-radix arithmetic (MRA), and explain mathematical model used later in this work.

1.1 The Standard for FP Arithmetic

1.1.1 Brief Historical Review

Leonardo Torres y Quevedo [72] and Konrad Zuse are considered as the first inventors of FP arithmetic [68]. Zuse's computer Z3, built in 1941, was the first real implementation of FP arithmetic [15]. Since then, various hardware implementations of FP formats and operations appeared. They were not compatible between each other; therefore numerical programs were not portable. In the end of 70s, Intel was designing a FP co-processor and hired W. Kahan to develop a standard for all their computations. Thereafter a draft for their new architecture design became the base of the FP standard. Kahan pointed out in 1981 the need of a simple abstract model for the machine's computational environment [47]. After several years of discussions the IEEE754 Standard was approved. Its first implementation on chip, the Intel 8087 was announced in 1980. The first Standard in FP arithmetic was published in 1985. It defined binary FP formats, rounding modes, basic operations and behavior in different operations. Thus, human readable decimal numbers had to be

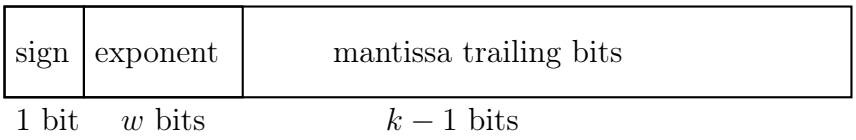


Figure 1.1: IEEE754 FP format

rounded to binary. In 1987 IEEE854 Standard was announced [44]. It was focused on numbers in radix two or ten, but unlike to IEEE754 did not specify any formats, it mostly contained constraints on numbers and their parameters. It did not allow to mix radices in one FP operation and was never implemented.

1.1.2 IEEE754-1985 Standard for Binary FP Numbers

The IEEE754-1985 Standard defines two basic binary formats: single and double precision[1]. The basic formats are the most used and they correspond to types `float` and `double` in C. Single precision numbers are represented on 32 bits, and double on 64 [42]. Each FP number is stored in three fields: sign, mantissa and exponent. The first bit is a sign bit s, then there are w exponent bits and $k-1$ mantissa trailing bits as it is shown on Figure 1.1.

Generally, mantissas in precision k may be written as $m_0.m_1m_2\ldots m_{k-1}$. As only binary formats are defined, all digits $m_i \in \{0,1\}$, therefore mantissas are usually normalized so that the first digit $m_0 = 1$. This bit is stored implicitly[2]. The rest is called *mantissa trailing bits* and is stored as an integer number on $k-1$ bits. Positive FP numbers are stored with a zero in the first bit, negative ones with a one.

Exponents are integer numbers that are stored on w bits. The IEEE754 FP formats use biased exponents. The used bias is $2^{w-1}-1$ which means that the numbers from the interval $[1-2^{w-1}; 2^{w-1}]$ can be encoded in a field of w bits. Actually, exponents take values from the interval $[2-2^{w-1}; 2^{w-1}-1]$ and the values $1-2^{w-1}$ and 2^{w-1} are reserved for special inputs that are explained later. All the upper-mentioned values associated with the IEEE754 formats are in Table 1.1.

In order clarify this paragraph, let us consider an example. The real number 23.5 is stored in single precision format as `0100 0001 1011 1100 0000 0000 0000 0000` or in hexadecimal `0x41bc0000`. The first bit is zero, as the number is positive, the next eight bits containing biased exponent are `100 0001 1` which is the number 131. So, to get the exponent value we subtract the bias and get 4. Mantissa trailing bits are `011 1100 0000 0000 0000 0000` which is 3932160 in decimal. Thus, to get our number from the IEEE754 encoding we compute the following

$$(-1)^0 \cdot 2^4 \cdot (1 + 2^{-23} \cdot 3932160)$$

[1] It also defines extended formats but the basic ones are used more often.

[2] The case when $m_0 = 0$ is covered later.

format	single	double	single extended	double extended
total length in bits	32	64	≥ 43	≥ 79
exponent field length, w	8	11	≥ 11	≥ 15
bias	127	1023	unspecified	unspecified
E_{min}	-126	-1022	≤ -1022	≤ -16382
E_{max}	127	1023	≥ 1023	≥ 16383
precision, k	24	53	≥ 32	≥ 64

Table 1.1: IEEE754 formats specification

Rounding Modes

Real numbers may have an infinitely long fraction part that is not representable in FP as mantissas have finite length. In this case the real number is somewhere between two adjacent FP numbers, thus we have to decide which of these two numbers should represent it. This process is called rounding. The Standard defines four rounding modes, the first three of which are called *directed*:

1. *rounding up* (rounding toward $+\infty$), RU(x). The smallest FP number greater than x is returned in this case.

2. *rounding down* (rounding toward $-\infty$), RD(x). The largest FP number less than x is returned.

3. *rounding toward zero*, RZ(x). For positive inputs returns RD(x), for negative ones RU(x).

4. *rounding to the nearest*, RN(x). The closest FP number to x is returned. In the case when x is exactly between two FP numbers, the number with the last even mantissa digit is chosen.

Figure 1.2 illustrates these roundings. The thick vertical lines correspond to FP numbers. The shorter thin lines in the middle of them are called *midpoints*. The points where the rounding functions change are called *rounding boundaries (or*

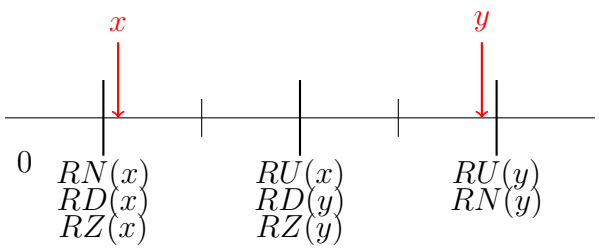


Figure 1.2: IEEE754 FP roundings

breakpoints). Directed roundings have the FP numbers as rounding boundaries, and rounding to the nearest has midpoints as boundaries.

The computation result is called *exact* if it is a FP number, so when no rounding is needed. Portability can be achieved with *correctly rounded* result, i.e. when it matches a result computed with infinite precision and then rounded. If there is no other FP number between the computed FP result and its exact version, such rounding is called *faithful* [68].

Special Values and Denormalized Numbers

We mentioned that the extreme values of the exponent (all ones or all zeros) are reserved for special data. With all ones in exponent field are stored infinities and NaNs and with all zeros so-called *denormalized numbers*.

It may happen that the result of the operation is larger (or smaller) than the largest (the least) representable FP number. The Standard defines special values for such cases: infinities (both positive and negative) and NaNs (Not-a-Number). These values are coded with all the ones in exponent field. If the mantissa field contains only zeros, this corresponds to an infinity, if there is at least one non-zero bit, it is a NaN. The last value is used in case when the result cannot be defined on real numbers, e.g. division by zero or a square root of a negative number.

We considered earlier the case with normalized mantissas, so the first bit was implicitly $m_0 = 1$. The number 0.0 obviously cannot be stored this way. The IEEE754 Standard supports signed zeros, which are stored with zeros in both exponent and mantissa fields. The least (in absolute value) FP number is $2^{E_{\min}}$ and has zero trailing mantissa bits, the next one differs by one in the last trailing mantissa bit. Let us compute a subtraction $x - y$, where $x = 2^{E_{\min}} \cdot 1.11$ and $y = 2^{E_{\min}} \cdot 1.0$, rounding mode is RD. If we compute the result mathematically, it is $x - y = 2^{E_{\min}} \cdot 0.11$, which cannot be represented in notation with 1 in hidden mantissa bit (it is $2^{E_{\min}-1} \cdot 1.1$) and thus the result may be rounded to zero. This effect is called *gradual (sometimes graceful) underflow* [38]. To avoid this *denormalized* numbers were introduced in IEEE754 Standard. These numbers are encoded with the format's minimal exponent i.e. with zeros in exponent field and the mantissa's hidden bit is 0. Thus, for our example, the result of difference is representable in IEEE754 FP and is encoded as $2^{E_{\min}} \cdot 0.11$. Inclusion of denormalized numbers guarantees that the result $x - y$ is non-zero when $x \neq y$.

As we described all values supported by IEEE754 we can consider now a "toy" example set of binary IEEE754-like FP numbers. We use precision $k = 3$ and biased exponent will be stored on three bits (so we can store exponents from $[-4, 4]$). For normal numbers (with $m_0 = 1$) the range of exponents is $[-3, 3]$. Thus, our set of FP numbers contains 28 positive and 28 negative finite normal numbers (four variants for mantissa trailing bits and seven exponents). The largest value of exponent $E = 4$ is reserved for infinities and NaNs. With the least exponent value $E = -4$ we encode

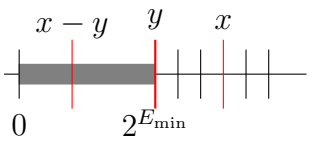


Figure 1.3: Gradual underflow zone is shown with gray

signed zero and six denormalized numbers: three positive and three negative. It has also positive and negative infinity and NaNs.

Exceptions

The set of FP numbers is discrete and has maximum and minimum values. So, the Standard also defines behavior when users operate on non-representable numbers (e.g. larger then the maximal FP number or smaller than the least). Here is the list of the supported exceptions:

1. *Overflow* occurs when we try to represent a number larger than the maximum representable FP number.

2. *Underflow* occurs when we try to represent a number smaller than the minimal representable FP number.

3. *Invalid* occurs when the input is invalid for an operation to be performed, e.g. $\sqrt{-17}$.

4. *Inexact* occurs when the result of an operation cannot be represented exactly, e.g. $\sqrt{17}$.

5. *Division by zero* occurs when a finite number is divided by zero.

When these exceptions are detected, they must be signaled. By default it is done with corresponding processor's flags, another option is a trap. Trap handler is a piece of code ran when the exception occurs. The system provides default trap handlers and there is a mechanism to use a custom trap handler [1, 74].

Operations

Due to Goldberg [38] a *correctly-rounded result* (CR) is obtained as if the computations were done with infinite precision and then rounded.

The Standard requires the four basic arithmetic operations ($+, -, \times, \div$), remainder, squared root, rounding to integer in FP format, conversion between FP formats, conversion between integer and FP formats, comparison of FP numbers and conversion between binary FP formats and decimal strings to be implemented. The four arithmetic operations and remainder have to be correctly-rounded. Algorithms for

basic arithmetic operations may be found in [31, 41]. Besides the arithmetic operations square root may also be made CR. These operations use the finite quantity of bits known in advance.

1.1.3 Table Maker's Dilemma

A clever person solves a problem. A wise person avoids it.

Albert Einstein[3]

A usual phenomenon in FP arithmetic is presented in this section. It occurs when correctly-rounded result is the goal of implementation.

About the ulp measure

Roundings introduce some errors to the computational results. To make sure that the result is reliable or accurate enough, these errors often have to be analyzed and bounded. Suppose that x is a real number and $\widehat{X}$ its FP representation. Absolute $\varepsilon = |\widehat{X} - x|$ or relative $\varepsilon = |\frac{\widehat{X}}{x} - 1|$ error may be computed.

Besides that an ulp function is often used. This abbreviation means *unit in the last place* and it was introduced by W. Kahan in 1960: "ulp(x) *is the gap between the two FP numbers nearest to* x*, even if* x *is one of them*". According to J.-M. Muller there are plenty of different definitions of ulp function [67]. We explain later in Section 1.2 how to compute this ulp function, or the weight of the last bit.

What is a Table Maker's Dilemma

The term Table Maker's Dilemma (TMD) was first pointed by W. Kahan and we cite it here [48]: *"Nobody knows how much it would cost to compute* y^w *correctly rounded for every two floating-point arguments at which it does not over/underflow. Instead, reputable math libraries compute elementary transcendental functions mostly within slightly more than half an ulp and almost always well within one* ulp*. Why can't* y^w *be rounded within half an* ulp *like SQRT? Because nobody knows how much computation it would cost... No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits. Even the fact (if true) that a finite number of extra digits will ultimately suffice may be a deep theorem".*

Consider a transcendental function f. Its value $f(x)$ cannot be computed exactly as it is a transcendental number. It means that the real value $f(x)$ has infinitely

[3]Albert Einstein (1879 - 1955) was a Nobel-prize awarded physicist, a "father" of modern theoretical physics. He is best known in popular culture for his mass–energy equivalence formula $E = mc^2$.

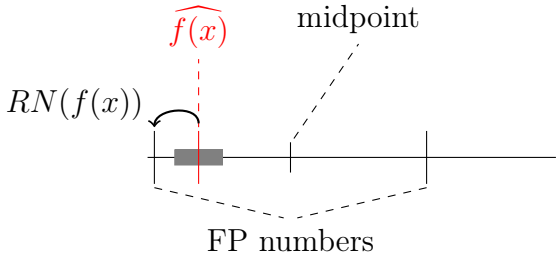


Figure 1.4: The case of easy rounding

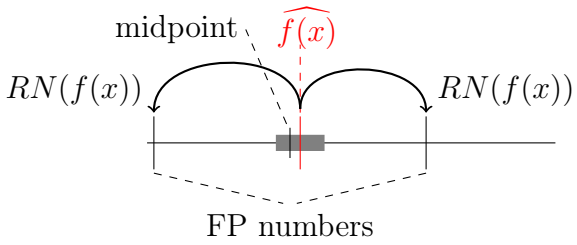


Figure 1.5: The case of hard rounding

long precision, but we may compute the function value only with some approximation procedure and thus we get a finite-precision number $\widehat{f(x)}$. The value $\widehat{f(x)}$ has finite accuracy m, probably much larger than k, precision of the format. We assume that $\widehat{f(x)}$ approximates the real value $f(x)$ with an error bounded by β^{-m}. The only information we have about the value $f(x)$ is the interval, where it belongs: it is usually centered in $\widehat{f(x)}$ and has length $2\beta^{-m}$. Consider an example in RN mode, so the rounding boundaries are the midpoints. There are two situations possible and they are shown on Figure 1.4 and Figure 1.5. The Figure 1.4 shows an example of easy rounding and the Figure 1.5 an example of hard rounding. When the interval that contains the real value $f(x)$ includes a rounding boundary, we cannot decide to which of the two nearest FP numbers the result should be rounded.

Hardness to Round and Worst Cases

Ziv proposed to increase the approximation precision m iteratively and recompute $\widehat{f(x)}$ until the interval for real function value does not contain rounding boundaries [87]. This strategy is often criticized as it is not known beforehand when the computation stops. Let be $\star \in \{RN, RD, RU, RZ\}$ one of the rounding modes. If we want to get correctly-rounded result, we have to be sure that the function returns $\star(f(x))$ and not just $\star(\widehat{f(x)})$. The TMD can be reformulated as the following question [68]: *can we make sure; if m is large enough that $\star(\widehat{f(x)})$ will always be equal to $\star(f(x))$?*

To get a positive answer to this question, our function f has to verify the following

condition: the infinitely-precise value $f(x)$ cannot be closer than β^{-m} to a rounding boundary. However, it may happen that some function values $f(x)$ are rounding boundaries. These values of x (in precision p), when the infinitely-precised mantissa of $f(x)$ is closest to a break-point are called *worst cases* for the TMD. The lowest bound for m is called *hardness to round*.

Definition 1.1 (Hardness to round)**.** *For a given FP format of radix β, for a given rounding mode, the hardness to round a function f on an interval $[a, b]$ is the smallest integer m such that $\forall x \in [a, b]$ either $f(x)$ is a break-point or the infinitely-precise mantissa of $f(x)$ is not within β^{-m} from a break-point.*

The TMD is solved if the lowest possible m and the worst cases are found. Some worst cases search will be performed in the framework of this thesis (Section 3.4). We do not review all the existing methods to solve it; we send the reader to [39,59,68] for more details.

1.1.4 Evolution of the Standard

The IEEE754 Standard was a great achievement in Computer Arithmetic. It unified all the approaches for FP implementations and thus was a beginning for research in the area of FP arithmetic. However, it had certain disadvantages that were supposed to be solved with the new version of the Standard in 2008 [43].

- financial computations suffer from rounding-off errors due to the use of binary arithmetic: conversion from decimal to binary always contains an error. For instance, decimal number 0.1 is infinite in binary 0.000110011(0011). This problem was partially addressed with IEEE854 Standard. The IEEE754-2008 defined decimal FP formats and requires CR conversions between formats of different radices

- for portability reasons the Standard required correct implementation of five basic arithmetic operations, while behavior on some special cases for mathematical functions is not defined. The new version of the Standard contains a chapter of recommendations on CR mathematical functions. We would like to emphasize that these are only recommendations, CR results are not required. However, this chapter contains a list of special values and exceptions for elementary functions.

- since 1985 some new features were developed and therefore had to be standardized too. For instance, in filter-processing and in computer graphics operations similar to $a \times b \pm c$ are often used. As we know, each arithmetic operation in FP computes the rounded result. Thus, performing multiplication and addition (subtraction) as two separate operations may lead to double rounding and

an unwelcome error [6]. IEEE754-2008 added to the list of required operations fused multiply-add (FMA) and heterogeneous arithmetic operations.

The IEEE754 Standard was being revised since early 2000s, and finally in 2008 a new version was published. It brought decimal formats and operations over decimal numbers, more than 200 new operations and recommendations on correctly-rounded function implementations. We do not discuss here how the decimal numbers are stored and manipulated, detailed explanations are in [22–24, 43, 68, 79].

Among the operations not only FMA appeared, the IEEE754-2008 defines so-called heterogeneous operations. The 1985 Standard declared operations on the FP numbers of the same format. The 2008's version allows us to mix the formats, for instance, a 32-bit FP number can be a result of addition of 64-bit number with a 32-bit one. However, it does not allow the radices to be mixed within one operation. The new standard renames the old binary formats: single precision is called binary32, double precision is binary64, denormal numbers are now subnormals.

Besides the old rounding modes from 1985's version, the 2008 revision added a new mode: *roundTiesToAway* (RA(x)). This is a rounding to the nearest mode, but in a case of midpoints the larger by magnitude value is returned. Binary implementations have roundTiesToEven mode as default and have to support also all three directed roundings. The fifth rounding mode roundTiesToAway is not mandatory for binary implementations, but is required for decimal.

The 2008 version of the Standard contains a recommendation chapter on correctly-rounded transcendental functions. As we have seen, such implementations require solving the TMD and for general case of transcendental function this solution is unknown. However, there are several functions that may be implemented correctly (already done in CRLibm [28]). The theses by D. Defour [27] and Ch. Lauter [56] addressed correct implementations of mathematical functions. Current work is a sequel of this work on automatic code generation for mathematical functions. Difference and similarity of ours approach and N. Brunie's code generator [11] is more detailed later and in [12].

1.2 Mathematical Model for FP Arithmetic

The computer is important but not to mathematics.

PAUL HALMOS[4]

This section contains a mathematical formalization of FP numbers that will be used later in algorithm development (Section 3.2, Section 3.3, Section 3.4). The Standard contains verbal descriptions of FP numbers in different formats. However, these descriptions are not convenient to use in new algorithms or proofs because of the lack of mathematical formalism. This model covers only finite normalized FP numbers that can be all handled in the similar way, infinities and NaNs need to be filtered out at the beginning of computations, subnormals need special treatment and are handled separately. All the definitions and theorems are inspired by the MPFR documentation [33].

Definition 1.2 (FP set)**.** *Let $k \geq 2$, $k \in \mathbb{N}$ be an integer. Numbers from the set*

$$\mathbb{F}_k = \left\{ x = \beta^E m \,|\, E, m \in \mathbb{Z}, E_{min} \leq E \leq E_{max},\ \beta^{k-1} \leq |m| \leq \beta^k - 1 \right\} \cup \{0\}$$

are called FP numbers of precision k in base β. The number E is called exponent and m mantissa.

We are going to define the four FP rounding directions with the use of different integer roundings. The floor function $\lfloor x \rfloor$ returns the integer number, not larger than x, the ceiling function $\lceil x \rceil$ returns the integer not smaller than x, the third function $\lfloor x \rceil$ rounds x to the closest integer number. The fourth FP rounding is defined as a combination of the two existing modes.

Definition 1.3 (Nearest integer)**.** *The rounding function $\lfloor \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ satisfies the following:*

$$\begin{aligned} \forall x \in \mathbb{R},\ & |\lfloor x \rceil - x| \leq 1/2 \\ & |\lfloor x \rceil - x| = 1/2 \Rightarrow \lfloor x \rceil \in 2\mathbb{Z} \end{aligned}$$

The second line in this definition ensures that in the case when x is situated between two integer numbers, the result of $\lfloor x \rceil$ is an even number.

Once we define the integer roundings we can define the FP roundings, that give a FP approximation of a real number. To make these formulas look simpler, we define first functions $E_k(x)$ and $\mathrm{ulp}_k(x)$.

[4]Paul Halmos (1916 - 2006) was a Hungarian-born American mathematician who was also recognized as a great mathematical expositor. He has been the first to use the "tombstone" (□) notation to signify the end of a proof, and this is generally agreed to be the case.

Definition 1.4. *Let be* $E_k(x) : \mathbb{R} \to \mathbb{Z}$ *the function defined for some* $k \in \mathbb{N}$ *as*

$$E_k(x) = \left\lfloor \log_\beta |x| \right\rfloor - k + 1$$

Definition 1.5. *Let be* $\mathrm{ulp}_k(x) : \mathbb{R} \to \mathbb{R}$ *the function defined for some* $k \in \mathbb{N}$ *as*

$$\mathrm{ulp}_k(x) = \beta^{E_k(x)}$$

Definition 1.6 (FP roundings)**.** *Let* $\circ_k, \nabla_k, \Delta_k$ *and* $\bowtie_k : \mathbb{R} \to \mathbb{F}_k$ *be defined as follows:*

$$\circ_k(x) = \begin{cases} 0 & \text{if } x = 0 \\ \mathrm{ulp}_k(x) \left\lfloor \frac{x}{\mathrm{ulp}_k(x)} \right\rceil & \text{otherwise} \end{cases}$$

$$\nabla_k(x) = \begin{cases} 0 & \text{if } x = 0 \\ \mathrm{ulp}_k(x) \left\lfloor \frac{x}{\mathrm{ulp}_k(x)} \right\rfloor & \text{otherwise} \end{cases}$$

$$\Delta_k(x) = \begin{cases} 0 & \text{if } x = 0 \\ \mathrm{ulp}_k(x) \left\lceil \frac{x}{\mathrm{ulp}_k(x)} \right\rceil & \text{otherwise} \end{cases}$$

$$\bowtie_k(x) = \begin{cases} 0 & \text{if } x = 0 \\ \Delta_k(x) & x < 0 \\ \nabla_k(x) & x > 0 \end{cases}$$

The functions $\circ_k, \nabla_k, \Delta_k$ *and* $\bowtie_k$ *are called respectively rounding-to-the-nearest, rounding-down, rounding-up and rounding-to-zero for FP numbers in precision* k*.*

This definition allows us to get the exponent and mantissa of the FP approximation of x like in Def. 1.2. For instance, let us represent a decimal number 12.345 as a binary FP number in $\mathbb{F}_{24}$ (using $\nabla_{24}(x)$ rounding). We start with computation of $E_{24}(x)$:

$$E_{24}(12.345) = \lfloor \log_2 |12.345| \rfloor - 24 + 1 = -20$$

Now we can compute $\nabla_{24}(x)$:

$$\nabla_{24}(12.345) = 2^{-20} \left\lfloor \frac{12.345}{2^{-20}} \right\rfloor = 2^{-20} \cdot 12944670$$

In the computed representation exponent $E = -20$, mantissa is $m = 12944670$ and is bounded by one binade $[2^{23}, 2^{24})$.

The same number 12.345 is encoded in IEEE754 single precision as

$$2^3 \cdot 1.10001011000010100011110,$$

so its mantissa is $m = 1 + 2^{-23}(2^{22} + 2^{18} + 2^{16} + 2^{15} + 2^{10} + 2^8 + 2^4 + 2^3 + 2^2 + 2)$ or $m = 1 + 2^{-23} \cdot 4556062$. Scaling the mantissa in IEEE754 representation by 2^{23} and

exponent by 2^{-23}, matches the result of $\nabla_{24}(12.345)$. Thus, there is the same idea in the IEEE754 representation and Def. 1.2, the only difference is in the bounds of mantissa: in our model we bound m by one binade $[2^{k-1}, 2^k)$ and in IEEE754 it is in $[1, 2)$[5]. Thus, the two representations are equivalent and one may be obtained from the other by scaling mantissa.

Therefore, numbers of binary32 (single) format after scaling mantissa by 2^{23} become binary numbers of $\mathbb{F}_{24}$, and the numbers of binary64 (double) format make the set $\mathbb{F}_{53}$ from Def. 1.2 with scaling mantissa by 2^{52}. This mathematical model does not take into account infinities and NaNs. All the algorithms for FP numbers usually perform filtering of special cases first, thus, the resting numbers are described by our model.

Property 1.1 (Factor of β)**.** *Let $k \in \mathbb{N}, k \geq 2$ be a precision. Let $\star_k \in \{\circ_k, \nabla_k, \Delta_k \bowtie_k\}$ be a rounding. Hence,*

$$\forall x \in \mathbb{R},\ \star_k(\beta \cdot x) = \beta \star_k(x)$$

Proof. The proof is based on the previous definitions. To start with, let us compute $E_k(\beta \cdot x)$.

$$E_k(\beta \cdot x) = \left\lfloor \log_\beta |\beta \cdot x| \right\rfloor = \left\lfloor \log_\beta |x| + 1 \right\rfloor = E_k(x) + 1$$

Thus, the exponent of the FP representation of $\beta \cdot x$ is larger than this for x by one. Consider the case with rounding down, for other modes this property is proven in the same way.

$$\nabla_k(\beta \cdot x) = \beta^{E_k(\beta \cdot x)} \left\lfloor \frac{\beta \cdot x}{\beta^{E_k(\beta \cdot x)}} \right\rfloor = \beta \cdot \beta^{E_k(x)} \left\lfloor \frac{\beta \cdot x}{\beta^{E_k(x)+1}} \right\rfloor = \beta \cdot \nabla_k(x)$$

□

So, binary FP numbers may be scaled by two without changing the rounding, and the decimal ones by ten.

Lemma 1.1 (Roundings and FP numbers)**.** *Let $\star_k \in \{\circ_k, \nabla_k, \Delta_k, \bowtie_k\}$ be a rounding (as defined in Def. 1.6) and $k \in \mathbb{N}$, $k \geq 2$ be a precision. Hence,*

$$\forall x \in \mathbb{R}, \star_k(x) \in \mathbb{F}_k.$$

Thus, rounding operations correspond to the common idea of rounding to a FP format. The lemma is applied to both binary and decimal FP numbers.

Proof. The proof is based on the definitions 1.2 - 1.6. We will prove this lemma for binary numbers and rounding-to-the-nearest mode, for other roundings and bases the same approach is used. The case $x = 0$ is trivial: $\circ_k(x) = 0 \in \mathbb{F}_k$. Otherwise by the definition Def. 1.6 we have

$$\circ_k(x) = 2^{E_k(x)} \left\lfloor \frac{x}{2^{E_k(x)}} \right\rceil$$

[5]We remind that only normal IEEE754 numbers are considered in our model.

It is clear that $E_k(x) = \lfloor \log_2 |x| \rfloor - k + 1 \in \mathbb{Z}$. We also have

$$\log_2 |x| - 1 \leq \lfloor \log_2 |x| \rfloor \leq \log_2 |x|$$

Therefore, this may be rewritten for the powers of two:

$$|x| \cdot 2^{-1} \leq 2^{\lfloor \log_2 |x| \rfloor} \leq |x|$$

We may multiply the inequality by 2^{-k+1} and get the following

$$|x| \cdot 2^{-k} \leq 2^{\lfloor \log_2 |x| \rfloor - k + 1} \leq |x| \cdot 2^{-k+1}.$$

Using this inequality we may get the bounds for fraction $\frac{|x|}{2^{\lfloor \log_2 |x| \rfloor - k + 1}}$

$$2^{k-1} \leq \left| \frac{x}{2^{\lfloor \log_2 |x| \rfloor - k + 1}} \right| \leq 2^k$$

Thus, $\circ_k(x) \in \mathbb{F}_k$ and the lemma is proven. □

1.3 Towards a Modern FP Environment

The IEEE754-2008 Standard brought some new aspects. It forced the research in decimal FP arithmetic and required 354 operations for binary format in comparison with 70 from the 1985 Standard (with requiring an FMA and heterogeneous operations). The next revision might appear in 2018, and it might bring even more operations. This work proposes to focus on two new aspects: to provide more flexibility for mathematical functions and to include mixed-radix (MR) operations. More flexibility means support of a huge family of implementations for each mathematical function (implementations different by final accuracy, domain, etc.). Mixed-radix operations should allow radices of inputs and output to be mixed without an extra call to conversion that could give better performance or accuracy results.

1.3.1 Mathematical Functions Implementations

Computing a value of mathematical function at some point requires execution of a sequence of arithmetical operations: as the functions are transcendental, we compute only their approximations. Elementary functions (e.g. $\exp, \log$) are used as basic bricks in various computing applications. The IEEE754-2008 Standard contains already a chapter with recommendations for correctly-rounded functions, therefore mathematical functions become the part of the FP environment. Software libraries containing functions implementations are called *libms*.

Currently, there are plenty different libms for different platforms, languages, accuracies. There are Open-Source libraries and proprietary codes: Intel's MKL, ARM's mathlib, Sun's libmcr, GNU glibc libm [32], Newlib for embedded systems [85], CRlibm by ENS Lyon [28] for correctly-rounded implementations, etc. Despite such great choice of libraries, users are not all satisfied with the current offer. The existing libraries are static and provide only one implementation per function and precision. Users need more today, e.g. choice between latency and throughput, possibility to change domains or to require final accuracy. There is a compromise between accuracy and performance, and as TMD is hard to solve, correctly-rounded implementations may suffer of the lack of performance in comparison with faithfully-rounded results.

The need of providing several implementation variants for each mathematical function has been discussed since longtime[6]. Modern mathematical libraries should contain several implementation variants of each mathematical function and a mechanism of choosing the right one. Profiling the SPICE circuit simulator shows that it spends most of its time on the evaluation of elementary functions [49]. The same holds for large-scale simulation and analysis code run at CERN [3, 45, 70]. So these are the two "use-cases" for "quick and dirty" function implementations. Users may

[6]for example `https://gcc.gnu.org/ml/gcc/2012-02/msg00469.html` or `https://gcc.gnu.org/ml/gcc/2012-02/msg00298.html`

be interested not only in CR or faithful implementations but also in codes that give more exotic accuracy, e.g. 2^{-50} or 2^{-45} or even 2^{-15}. Existing libraries may be improved and enlarged not only in terms of accuracy: we may add performance options (for compromise between latency and throughput), portability options (support of some special instructions). Besides that, we may let users change the domain: for small domains there is no need to handle infinities, NaNs, under/overflows and other exceptions. Supporting all these choices means implementation of a tremendous number of different variants for each mathematical function from a libm. It is also desired that new flexible libms support even "non-standard" functions. The quantity of all the implementation variants is tremendous and may be made infinite. That makes the task of manual writing a new libm impossible and thus we need a code generator to produce all these implementations. Code generators have already shown their efficiency in mathematical software, e.g. ATLAS [86] project for linear algebra, and the FFTW [34] and SPIRAL [71] projects for fast Fourier transforms.

Implementation of a libm requires extensive knowledge in numerical analysis, FP computing, and probably architecture specification for optimization. This means that some "exotic" libms may be developed by people without special (large) knowledge of mathematical functions implementation or floating-point tricks [28]. Generating correct-by-construction implementations prevents from appearance of poor implementations.

With such code generator we pass to a "higher" level of programming (writing code to produce the code instead of direct result is usually called *metaprogramming*). So we call this generator *Metalibm*. It is a part of the ANR[7] Metalibm project [25] that covers two basic aspects: code generation for mathematical functions and for digital filters. Generation of digital filters is out of scope of the current work, we focus only on function generation.

Since the existence of the project two main branches were developed: the one described here, and the other one found in N. Brunie's thesis [11]. N. Brunie's version was developed mainly as an assistance framework for libm developers: once all the needed steps for function implementations are determined, the corresponding classes and methods may be called. One of the motivations for it was lack of performance of standard libms on Kalray's processors, so for a specified processor it chooses the right CPU instructions.

Our version was created to be a push-button generator with the support of black-box functions: the tool takes a set of different parameters like final accuracy, domain, etc. and the function itself. Roughly speaking, our generator does not know which function it is generating (exp, log or other), but is able to establish its significant algebraic properties on-the-fly. However, the both approaches are based on the same idea: take a set of parameters for function implementation and generate corresponding code automatically. It is hard to make a clear distinction between them and of

[7] ANR is an abbreviation for *Agence National de Recherche* for French National Research Agency

course an interesting future concept is to combine the two generators into one [12]. Here and after we are going to use the word "Metalibm" for our version of code generator.

1.3.2 Mixed-radix Arithmetic

Decimal FP arithmetic is used in financial computations to avoid rounding issues when decimal inputs are transformed to binary. However, the overwhelming majority of computer architecture is binary. It means that we mix binary and decimal FP numbers inevitably. The IEEE754-2008 Standard required operations, rounding modes, exceptions, etc. for each radix, so there are two FP worlds: binary and decimal, and they only intersect in conversion operations. According to the Standard we have to provide conversion from binary format to a decimal character sequence and back, and when the decimal formats are available, then there should be conversions in both directions in all formats between binary and decimal FP numbers. Operations that mix decimal and binary inputs are not yet specified. We propose to call them *mixed-radix* (MR) operations. For instance, comparison between binary and decimal numbers is already investigated [9]. Before specifying these operations we must evaluate their feasibility and cost.

Research on mixed-radix operations started with Brisebarre's et al. paper on comparison of binary64 and decimal64 numbers [9]. Another existing paper in mixed-radix arithmetic is about an atomic operation of radix conversion [55] and this approach is explained in Section 3.2.

We start the research in mixed-radix arithmetic operations on FMA operation. As mentioned, having an FMA implemented, we get mixed-radix addition (subtraction) and multiplication. It is also an often-used operation in polynomial evaluation, in filter processing and computer graphics [40, 41, 83]. We start with evaluation of the cost of mixed-radix FMA implementation, so we search its worst-cases (see Section 3.4).

An interesting widely-used conversion operation is between decimal character sequence and binary FP number. It is not a trivial operation as we are working with an arbitrary user input. Therefore the user-given number may be of an arbitrarily large precision, which makes computation of TMD worst cases impossible. The existing `scanf` from glibc does not support all rounding modes and uses memory allocation, which means it cannot be reused in embedded systems. In Section 3.3 a new algorithm to transform user's decimal string representation of FP numbers is presented. It is re-entrant, works for all rounding modes and its memory consumption is known beforehand.

1.3.3 Conclusion

Transition from 1985 to 2008 version of the Standard brought about 280 new operations, and we have noted several more useful operations for FP environment. Inclusion of mixed-radix operations to the Standard means its enlarging up to 500 new operations. There is a growing interest in various implementations of mathematical functions. The number of functions is huge, the number of their variants tends to infinity, thus each new version of the Standard might require or at least recommend more and more operations. This work paves the way to the research in mixed-radix arithmetic and to optimization of generated mathematical functions implementations.

CHAPTER 2

Automatic Implementation of Mathematical Functions

Computing is kind of a mess. Your computer doesn't know where you are. It doesn't know what you're doing. It doesn't know what you know.

LARRY PAGE[1]

Existing mathematical libraries (which are called *libms*) provide code for evaluation of mathematical functions. However, they are static as they give one implementation per function and too generalized: argument range is specified "as far as the input and output format allows" and accuracy "as accurate as the output format allows" [32]. This means that for a specific context or an application, the functions in the libm are often overkill.

It was already mentioned that modern libms should contain several implementations of each mathematical function: e.g. for small domain, for large domain, for various specified accuracies. Manual implementation of such a tremendous number of all possible versions is not feasible that is why we aim to write a code generator [54].

To know how to generate code for mathematical functions, we review first the basic steps for manual implementation (Section 2.1). There are usually three steps, so-called argument reduction (Section 2.1.1), approximation (Section 2.1.2) and reconstruction (Section 2.1.3). Besides some routines on each of mentioned steps usually serious error analysis is required. We explain how to perform it on an example of exponential function(Section 2.1.4).

Afterwards we explain how our code generator works (Section 2.2). It generates the needed steps for function implementations automatically. Interesting non-trivial

[1]Larry Page (1973) is an American computer scientist and internet entrepreneur who co-founded Google Inc. with Sergey Brin, is the corporation's current CEO, is the inventor of PageRank, Google's best-known search ranking algorithm.

algorithms are used to split domains [53] (see Section 2.2.4) and to replace branching in reconstruction by polynomials [52] (see Section 2.2.5).

2.1 Background for Manual Implementation

A univariate mathematical function is called *elementary* if it consists of a finite number of compositions, additions, subtractions, multiplications and divisions of complex exponentials, logarithms, constants and n-th roots. Examples of elementary functions are $\exp, \log$, trigonometric functions. Functions that are not elementary are usually called *special*, e.g. Gamma function. The existing libms contain code to evaluate elementary and several special functions (e.g. Gamma, Bessel, erf).

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, comparisons, and memory for precomputed tables. A generic technique exploiting all this is to approximate a function by tables and polynomials (possibly piecewise).

Usually implementations of an elementary (mathematical) function consist of the following steps [66]: filtering special cases (NaNs, infinities, numbers that produce overflow/underflow and other exceptions), argument reduction, approximation and reconstruction. We use simple *if-else* statements and comparisons to filter out the inputs that produce non-finite result or exceptions. Implementing a function on some domain requires its approximation on the same domain. Usually polynomials or rational functions are used for approximations. As we know [16], the approximation domain, degree and accuracy are related. So the smaller is the domain, smaller is the degree and more accurate is the approximation. Thus, to use the approximations of low degrees (that accumulate less FP round-off errors), the specified domain for function implementation is usually reduced. After computing approximation on the small domain, we perform the inverse transition that allows to evaluate the function on the whole domain.

Metalibm relies on the existing software in numerical computations *Sollya* [18] and *Gappa* [26] that will be detailed later. These tools deal well with polynomials and are developed as basic bricks for mathematical functions generator. Sollya contains useful routines for polynomial approximations and Gappa certifies accuracy of FP code. This is the main reason why we focus on polynomial approximations.

The argument reduction step is based on algebraic properties of the function to be implemented (see Section 2.1.1), reconstruction is then an inverse transformation. Exploiting mathematical properties to reduce implementation domain means that there does not exist any general argument reduction procedure that can be applied to any type of the function. For some functions e.g. erf or functions purely defined by ODE, we do not have such useful properties to reduce the domain. As we are still interested in polynomials of small degree, piecewise approximations are used instead.

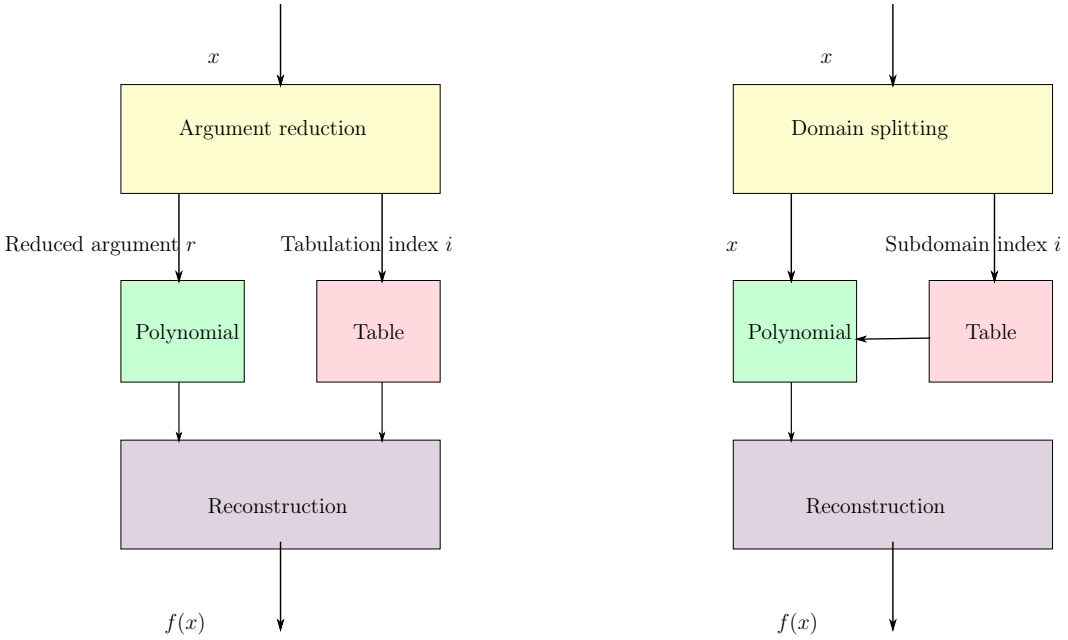


Figure 2.1: Two basic schemes to implement a mathematical function

In this case, argument reduction is an execution of some domain splitting algorithm and reconstruction is domain decomposition. To summarize, mathematical functions are implemented within one of schemes shown on Figure 2.1. This is a classical division of two approaches to implement mathematical function. In practice, we may combine these schemes too. The naive example is a symmetric function like asin for instance, its domain may be reduced twice first and then piecewise polynomials are computed. We explain the mix of these schemes later in Section 2.2.2.

In the current section we review existing argument reduction procedures, techniques to find and evaluate the approximation polynomial and how to perform error analysis of the whole implementation.

2.1.1 Argument Reduction

As mentioned, the purpose of this step is to reduce the degree of polynomial approximation with the reduction of implementation domain. Figure 2.2 illustrates exponential function with its two approximating polynomials. On the left there is a polynomial computed for a small domain, on the right for a larger domain. On small domains they almost match the function values, but for larger domain the error gets larger. Argument reduction procedures usually depend on algebraic properties of the function to be implemented, for instance $e^{x+y} = e^x e^y$, $\log(xy) = \log(x) + \log(y)$, $\sin(x + 2\pi k) = \sin(x)$. As we mentioned, some functions do not have any useful properties to reduce the domain and we use piecewise polynomial approximations

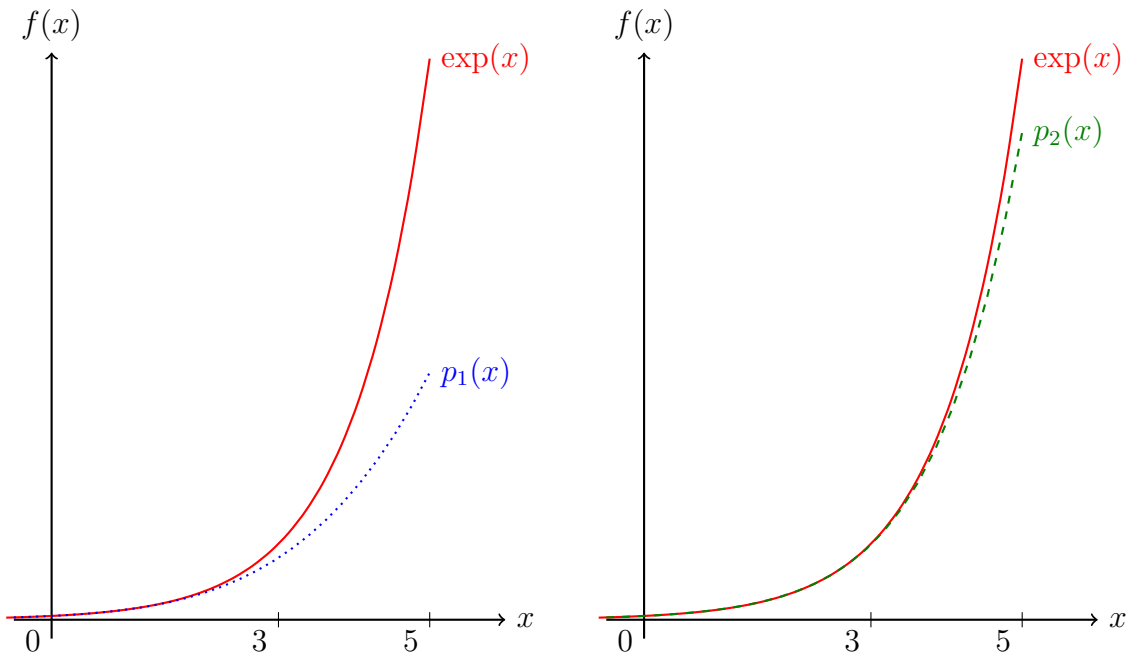


Figure 2.2: Polynomials approximating $\exp(x)$

then. There are different ways to *split* the initial domain into several small *subdomains* to get several polynomial approximations of small degree. We review these methods later with the details on our splitting algorithm (Section 2.2.4) and here we focus on several properties-based reduction schemes.

The goal for argument reduction is to represent the function value $f(x)$ through some function value $g(r)$, where r is a reduced argument and has to be deduced from x, we use different function g because it may not be the same as f. The way to reduce argument for some functions is not unique and may depend on the requirements of the implementation. We show how it is possible to reduce the argument on several examples.

Exponential Function e^x

The useful property of this function is that $e^{(a+b)} = e^a \cdot e^b$. We will try to represent the value e^x as a multiplication of 2^E, $E \in \mathbb{Z}$ and $2^r, |r| < 1/2$, so we use representation similar to Def. 1.2. The first term corresponds to the exponent of the result, and the second one to the mantissa. This way, the exponential may be transformed as follows:

$$e^x = 2^{x \log_2 e} = 2^{\frac{x}{\ln 2}} = 2^{\left\lfloor \frac{x}{\ln 2} \right\rceil} \cdot 2^{\frac{x}{\ln 2} - \left\lfloor \frac{x}{\ln 2} \right\rceil} = 2^E \cdot e^{x - E \ln 2} = 2^E \cdot e^r \tag{2.1}$$

Thus, exponent of the result is

$$E = \left\lfloor \frac{x}{\ln 2} \right\rceil, \tag{2.2}$$

and the reduced argument is

$$r = x - E \cdot \ln 2; \quad r \in \left[-\frac{\ln 2}{2}, \frac{\ln 2}{2} \right] \tag{2.3}$$

With reduced memory cost, in late 80s-90s table-driven methods appeared [35,80–82]. Tang's method allows us to reduce the argument even more with a precomputed table of 2^t values [80]. Here we do almost the same as before. We start with representing e^x as a power of two, and then we multiply and divide this power by 2^t:

$$e^x = 2^{x \log_2 e} = 2^{\frac{x}{\ln 2}} = 2^{\frac{1}{2^t} \cdot \frac{x \cdot 2^t}{\ln 2}} \tag{2.4}$$

Then, we represent the fraction $\frac{x \cdot 2^t}{\ln 2}$ as

$$\frac{x \cdot 2^t}{\ln 2} = E + \left(\frac{x \cdot 2^t}{\ln 2} - E \right), \tag{2.5}$$

where the integer number $E = \left\lfloor \frac{x \cdot 2^t}{\ln 2} \right\rfloor$. This integer number E may be represented as $E = 2^t \cdot m + i$, where m is an integral part of division E by 2^t and i is the remainder, so takes integer values from $[0, 2^t - 1]$. We take $r \in \left[-\frac{\ln 2}{2}, \frac{\ln 2}{2} \right]$, then the expression in brackets from (2.5) way be written as

$$\frac{r}{\ln(2)} = \left(\frac{x \cdot 2^t}{\ln 2} - E \right).$$

Thus, coming back to (2.4) we have the following expression for exponential:

$$e^x = 2^{1/2^t \cdot \frac{x \cdot 2^t}{\ln 2}} = 2^{1/2^t \cdot (2^t m + i)} \cdot 2^{1/2^t \cdot \frac{r}{\ln(2)}}$$

Simplifying the expression and taking $r^* = r/2^t$ we get

$$e^x = 2^m \cdot 2^{i/2^t} \cdot e^{r^*}$$

Thus, m is the new exponent of the result, values $2^{i/2^t}$ are precomputed and stored in a table for $i = 0, 1, \ldots, 2^t - 1$. The reduced argument is

$$r^* \in \left[-\frac{\ln 2}{2^{t+1}}, \frac{\ln 2}{2^{t+1}} \right].$$

Tang proposed to compute the approximation polynomial p for function $e^{r^*} - 1$, thus the final expression for exponential is

$$e^x = 2^m \cdot 2^{i/2^t} \cdot (1 + p(r^*)). \tag{2.6}$$

Logarithm function $\ln(x)$

Due to IEEE754 standard, we may decompose binary FP numbers to the form $x = 2^{E'}m'$ where $1 \leq m' < 2$, $E \in \mathbb{Z}$. Then, according to logarithm properties, we get

$$\ln(x) = E' \ln(2) + \ln(m')$$

This simple direct decomposition cannot be used for final argument reduction: we get catastrophic cancellations when $E' = -1$ and m' is close to 2. Therefore, two more enhancements are performed. Firstly, we decompose the number x differently, so that $x = 2^E m$ where $\frac{1}{\sqrt{2}} \leq m < \sqrt{2}$.

$$E = \begin{cases} E' & \text{if } m' \leq \sqrt{2} \\ E' + 1 & \text{if } m' > \sqrt{2} \end{cases}$$

With such decomposition of argument x, we get

$$\ln(x) = E \ln(2) + \ln(m),$$

where $-\frac{1}{2}\ln 2 \leq \ln m \leq \frac{1}{2}\ln 2$. Magnitude of the reduced argument is still large and $\ln m$ cannot be approximated by a polynomial of low degree. We are going to use tabulated values to reduce the argument even more. We use the first n bits of mantissa m as index i to look up tabulated value t_i that approximates $1/m$. Taking $r = mt_i - 1$ we get

$$\ln m = \ln(1 + r) - \ln t_i.$$

We approximate the value of $\ln(1 + r)$ by polynomial $p(r)$ and then assigning $l_i = -\ln t_i$, we get the final formula:

$$\ln x = E \ln 2 + p(r) + l_i.$$

The reduced argument is $|r| < 2^{-n}$. Usually n is chosen between 6 and 8, which makes two tables of 64 to 256 entries.

2.1.2 Polynomial Approximation

Once the implementation domain is somehow reduced, we can compute polynomial coefficients to approximate $g(r)$. There are different techniques to find this polynomial. One of them is minimax polynomial. Infinite norm is usually defined with the following formula:

$$\|f - p\|_\infty^{[a,b]} = \max_{a \leq x \leq b} |p(x) - f(x)|$$

Minimax approximation p for the function f minimizes this infinite norm.

We review the basic theorems used to find a minimax-like polynomial or to establish relation between the approximation accuracy, domain and polynomial degree [16, 17].

Theorem 2.1 (Weierstraß, 1885)**.** *Let f be a continuous real-valued function on $[a, b]$ and if any $\varepsilon > 0$ is given, then there exists a polynomial p on $[a, b]$ such that*

$$\|f - p\|_{\infty}^{[a,b]} \leq \varepsilon.$$

Proofs for the theorems may be found in [16].

This means that any continuous on $[a, b]$ function may be approximated by a polynomial as accurately as needed.

Theorem 2.2 (Chebychev's alternance)**.** *Polynomial p is the best approximation of continuous function f over an interval $[a, b]$ if and only if there are at least $n + 2$ points $x_0 < x_1 < \ldots < x_{n+1}$ in $[a, b]$ such that*

$$f(x_i) - p(x_i) = \alpha(-1)^i \|f - p\|_{\infty}^{[a,b]},$$

where $i = 0, 1, \ldots, n + 1$ and $\alpha = 1$ or $\alpha = -1$ for all i at the same time.

These points $x_0, x_1, \ldots, x_{n+1}$ from the theorem are called *Chebychev's nodes*. This theorem about alternance shows that the sign of error for minimax polynomial of degree n alternates and reaches its extrema $n+2$ times. The next theorem gives the bounds for the optimal error of a minimax polynomial.

Theorem 2.3 (of de la Vallée-Poussin)**.** *Let f be a continuous function on $[a, b]$, p its approximation polynomial on $n + 2$ points $x_0 < x_1 < \cdots < x_{n+1}$ 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 μ verifies*

$$\min_{i=0,1\ldots,n+1} |f(x_i) - p(x_i)| \leq \mu \leq \max_{i=0,1\ldots,n+1} |f(x_i) - p(x_i)|.$$

So, we have a theorem that establishes relation between approximation error, domain and polynomial degree. Therefore, it may be used to make a decision if domain splitting is needed.

Remez [73] proposed an iterative convergent algorithm to find a minimax-like polynomial starting with Chebyshev nodes. It has quadratic convergence to a minimax polynomial when the function f is twice differentiable and with additional conditions for approximation points x_i [17]. We do not explain here the whole algorithm, we just give a short overview. We start with $n + 2$ points $x_0, \ldots, x_{n+1}$ from $[a, b]$. Chebyshev nodes are often chosen on the first step [16]:

$$x_i = \frac{a+b}{2} + \frac{b-a}{2} \cos\left(\frac{(n+1-i)\pi}{n+1}\right), \; i = 0, \ldots, n+1.$$

Then the following actions are repeated in a loop until the needed approximation accuracy is reached. First, an interpolation polynomial p of f has to be computed on the chosen $n + 2$ points. Current accuracy of this polynomial is

$\varepsilon = \max_{i=0,\ldots,n+1} |p(x_i) - f(x_i)|$. We find a set of points M where the local maximum of error $|p - f|$ is reached. If the errors for each point in M alternate in sign and have equal magnitude, the minimax polynomial is found. If not, we choose another set of $n+2$ points and repeat the computations. The next set of points contains the point ξ such that the error $|p - f|$ reaches its global maximum.

Thus, this algorithm computes a sequence of polynomials $p^{[j]}$ that converges to a minimax polynomial. A condition to stop the iterations is that the approximation error on j-th step is less than the required accuracy. There is a FP modification of the classical algorithm proposed in 1934 that produces FP coefficients for the specified precisions [8]. This Remez-like algorithm is implemented in Sollya numerical tool and therefore is used in Metalibm. To compute this polynomial we should also specify the desired accuracy and polynomial degree.

For our example with exponential function, for an interval $[1; 3]$, desired accuracy 10^{-5} and degree 7, the approximation polynomial with coefficients rounded to double is

$$\begin{aligned} p(x) =& 4398690705538611 \cdot 2^{-52} + 4976570300619493 \cdot 2^{-52} \cdot x \\ &+ 42051858177809 \cdot 2^{-47} \cdot x^2 + 26740262321023 \cdot 2^{-46} \cdot x^3 \\ &- 6813315854891583 \cdot 2^{-56} \cdot x^4 + 8849499087929315 \cdot 2^{-57} \cdot x^5 \\ &- 6118517341039965 \cdot 2^{-59} \cdot x^6 + 3499742641603825 \cdot 2^{-61} \cdot x^7 \end{aligned}$$

Domain $[1; 3]$ is quite large. Returning to the example from Section 2.1.1, we have domain $\left[-\frac{\ln 2}{32}, \frac{\ln 2}{32}\right]$, we may use the following polynomial of degree four.

$$\begin{aligned} p(x) =& 562949953421317 \cdot 2^{-49} + 9007199249577219 \cdot 2^{-53} \cdot x \\ &+ 9007199250265725 \cdot 2^{-54} \cdot x^2 + 3002487796550777 \cdot 2^{-54} \cdot x^3 \\ &+ 6004963853677043 \cdot 2^{-57} \cdot x^4 \end{aligned}$$

Coefficients are rounded to double precision. Approximation error over the specified domain determined with Sollya is about $2.48371 \cdot 10^{-12}$.

Polynomial Evaluation

Finding polynomial coefficients for a good approximation is a mathematical problem. In computer science we have to choose also a good evaluation scheme to implement. For a polynomial of degree n all the evaluation schemes perform n additions (unless some of the coefficients are zeros), so the main difference between them is in the quantity of the multiplications.

It is clear that the direct computation of polynomial cannot be used: execution of all the multiplications in this expression as

$$a_n \underbrace{x \cdot x \cdot \ldots \cdot x}_{n} + a_{n-1} \underbrace{x \cdot x \cdot \ldots \cdot x}_{n-1} + \ldots + a_1 x + a_0$$

requires too many operations and therefore brings too many round-off errors. According to Mouilleron and Révy [65], for the polynomials with some specific properties on the coefficients some non-trivial methods are applied (e.g. E-method [30], Estrin's method). For manual implementations the approximations are found manually, so programmers choose the corresponding evaluation scheme depending on some properties of this polynomial. In general case, when there is no specific information about the coefficients of the polynomial, Horner's scheme is used.

2.1.3 Reconstruction

Argument reduction step is roughly speaking a transformation of $f(x)$ to some $g(r)$, where r is reduced argument from a small domain and g is some function, may be the same as f. Then this function g is approximated by a polynomial p on a small domain for its argument r. Reconstruction is an inverse process: from polynomial values on the small domain $p(r)$ get the function values on the large initial domain $f(x)$. In the case of piecewise polynomial approximation, reconstruction is a domain decomposition: we have to choose the corresponding polynomial coefficients for the input $x \in [a, b]$.

For our example with exponential function, reconstruction formula is

$$e^x = 2^E \cdot 2^{i/2^t} \cdot (1 + p(r)).$$

For piecewise approximations different domain decomposition techniques may be applied. They depend on the domain splitting procedure used. If the domain was split into N equal parts, we may use a linear function to determine the right index of the set of polynomials $\{p_i\}_{i=0}^{N-1}$.

2.1.4 Error Analysis

Mathematical libraries have to produce reliable results, therefore all arithmetic operations used to get the function value have to be thoroughly analyzed: we perform computations in FP arithmetic and round-off errors are accumulating on each step. For function implementations there are three main error sources: approximation error ε_{appr}, polynomial evaluation error, and error from all other FP computations. For functions with domain splitting there are only two error sources: polynomial approximation and evaluation.

We explain how to perform error analysis continuing the example with exponential function. We compute it within the following formula

$$e^x = 2^E \cdot 2^{i/2^t} \cdot (1 + p(r)).$$

Polynomial $p(r)$ is evaluated with some error ε_{eval}, thus we should replace $p(r)$ in the reconstruction formula by $p(r)(1 + \varepsilon_{eval})$. This polynomial approximates $e^r - 1$,

thus we may write $(e^r-1)(1+\varepsilon_{eval})(1+\varepsilon_{appr})$ instead of $p(r)$. Tabulated values $2^{i/2^t}$ are FP numbers, thus they are computed with some error too, so we replace them by $2^{i/2^t}(1+\varepsilon_{tbl})$.

Taking into account all these errors, we compute some $\widehat{e^x} = e^x(1+\varepsilon)$, and the goal is to represent this ε through the mentioned errors. So, we rewrite the whole expression for $\widehat{e^x}$ with all the errors:

$$\widehat{e^x} = 2^m \cdot 2^{i/2^t} \cdot (1+\varepsilon_{tbl}) + 2^m \cdot 2^{i/2^t} \cdot (e^r-1)(1+\varepsilon_{tbl})(1+\varepsilon_{appr})(1+\varepsilon_{eval})$$

After multiplications and simplifications we get

$$\widehat{e^x} = e^x\left(1 + \frac{2^m \cdot 2^{i/2^t}}{e^x}\varepsilon_{tbl} + \frac{2^m \cdot 2^{i/2^t} \cdot (e^r-1)}{e^x}\Big(\varepsilon_{tbl} + \varepsilon_{appr} + \varepsilon_{eval} + O\big(\varepsilon_{tbl} + \varepsilon_{appr} + \varepsilon_{eval}\big)\Big)\right)$$

Thus, the overall error of such function evaluation is

$$\varepsilon = \frac{2^m \cdot 2^{i/2^t}}{e^x}\varepsilon_{tbl} + \frac{2^m \cdot 2^{i/2^t} \cdot (e^r-1)}{e^x}\Big(\varepsilon_{tbl} + \varepsilon_{appr} + \varepsilon_{eval} + O(\varepsilon_{tbl} + \varepsilon_{appr} + \varepsilon_{eval})\Big)$$

This allows us to perform forward and backward error analysis [41]. In forward analysis we get an estimation or a bound for overall error when we know all the errors $\varepsilon_{appr}, \varepsilon_{tbl}, \varepsilon_{eval}$. In backward error analysis we determine the bounds for $\varepsilon_{appr}, \varepsilon_{tbl}, \varepsilon_{eval}$ knowing only the final error ε.

There exist a software tool called Gappa for automatic certification of the approximation accuracy. We review it later in Section 2.2.2.

2.2 Code Generation for Mathematical Functions

The idea of writing a Metalibm, a code generator for mathematical functions, appeared first in [56], since then two different approaches were developed. They are reviewed in Section 2.2.1.

2.2.1 Use Cases for Generators of Mathematical Functions

As it was mentioned, the two main use cases are distinguished and developed for generator of mathematical function implementations. The first one targets the widest audience of programmers. It is a push-button approach that tries 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 specified 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.

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 the needs of libm developments have to be addressed too. Although, the techniques used in open-ended approach 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 cannot be yet automated. 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. 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. More details may be found in [11] and [12].

The first, open-ended use case is more ambitious, more high-level, and top-down from the most abstract mathematical description of a function. This is a subject of current work, the second one is N. Brunie's version of Metalibm.

We start with a brief general overview of the open-ended generator, and formulate its objectives (see Section 2.2.2). Then we explain how to perform automatically each of previously described steps in function implementation (Section 2.2.3, Section 2.2.4, Section 2.2.5). Current section is finished with examples, results and analysis of

Metalibm's perspectives in Section 2.2.7. Different aspects that will be explained in this section, were published in [12, 52–54].

2.2.2 Open-ended generator

We are going to produce code for various function specifications. We call them function variants or sometimes flavors. Each flavor is determined by its parameters (listed below) among which we find the function itself. So, Metalibm does not know in the beginning which function it is generating. However, it should be able to evaluate these functions over an interval with arbitrary accuracy. This enables exploiting algebraic properties and generating code of comparable to glibc libm performance.

Parameter Set

We aim to give users more choices in implementations of libm functions. The most important choices include specification of a function f, desired domain of implementation $[a, b]$, requirements for accuracy of the result $\bar{\varepsilon}$, limit for approximation polynomial degree d_{max} and size of the table t (in case when table-driven argument reduction is used). We cannot give any guidelines for the choice of these parameters, it should be determined by user. There is no option to require CR implementations for the moment, however there are two strategies for users if they need CR functions:

- require higher accuracy than needed for the worst case. Then manually patch the generated code: perform rounding to the needed accuracy and test if such result gives a CR function version;
- refer to CRlibm that implements efficiently CR functions from a usual libm.

One of the objectives for Metalibm is to support black-box function specifications: create a generic generator for arbitrary function. There is no dictionary with fixed set of the supported functions, user may provide code to evaluate some "exotic" function over an interval with arbitrary accuracy. This has to be a function continuous with its first few derivatives.

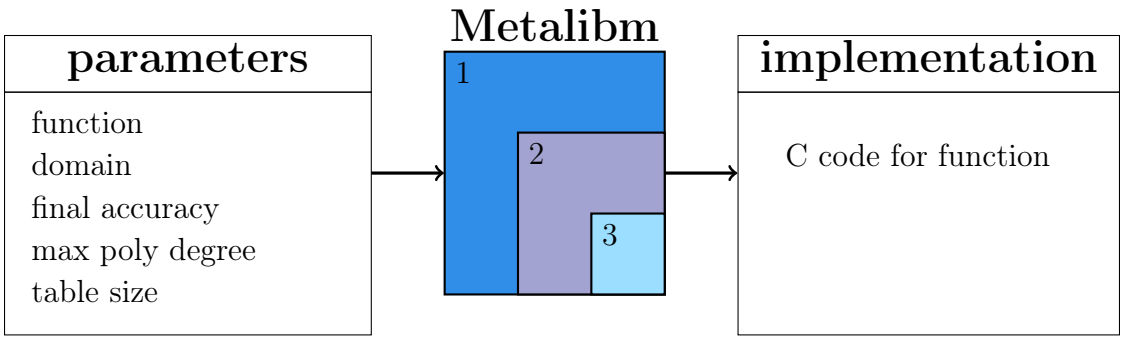
Toolkit for Function Generation

The aim of the generator is to create function implementations "accurate"[2] and hence "correct"[3] by construction. There are already several existing tools useful in implementation of such generator (we have already mentioned them before). Sollya[4] is a numerical tool for reliable (safe) computations that contains different numerical

[2] at least not worse than the specified accuracy $\bar{\varepsilon}$.

[3] not in the sense of CR

[4] `http://sollya.gforge.inria.fr/`



1 - Properties detection
2 - Domain splitting
3 - Approximation

Figure 2.3: Metalibm scheme

algorithms implemented [18, 20]. For instance, state-of-the-art polynomial approximations [8], safe algorithms to compute $\varepsilon_{\text{approx}} = \|f - p\|_{\infty}^{I}$ [19] as well as a scripting language. As mentioned, it is important to handle FP error rigorously and to guarantee result's accuracy. For manual implementations errors are managed manually, for Metalibm we can use Gappa formal proof assistant [63]. Compared to [26], 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).

Three Levels of Code Generation

In the end of Section 2.1 we put the two basic schemes of mathematical function implementation (see Figure 2.1). The first one contains the following steps: argument reduction, approximation and reconstruction. The function is approximated by one polynomial. The second scheme is for piecewise polynomial approximations and has these steps: domain splitting, polynomial approximations and reconstruction that is done usually with *if-else* statements. Roughly speaking, Metalibm combines these two schemes and executes all the steps automatically. Despite it is a black-box function generator, it performs specific argument reduction procedures that depend on mathematical properties. So, it detects these properties automatically (will be explained later how). The produced code is ready-to-use and the demanded accuracy is guaranteed. To do this Metalibm chooses the precision of the internal computations automatically: for highly-accurate flavors it may use double-double or triple-double arithmetic [29, 77]. This is equivalent to automatic error analysis as in Section 2.1.4.

The scheme of Metalibm code generation may be found on Figure 2.3. There are three levels of code generation:

- We start with detecting algebraic properties of functions (several examples are explained later in Section 2.2.3). In case of success appropriate argument reduction scheme is applied.

- It may happen that after argument reduction we cannot approximate the function with one polynomial with the given constraints (reduced domain, accuracy and polynomial degree). We use piecewise-polynomial approximations. Thus, domain has to be split into several subdomains. We decide if the splitting is needed with the help of Theorem 2.3 given in Section 2.1.2. Our algorithm for domain splitting is explained in Section 2.2.4.

- Finally, on the last level we have small domain (or subdomains) and we compute the approximation polynomial (or polynomials) and generate Gappa proofs for their approximation error. As function flavors are arbitrary, the approximation polynomials are arbitrary too. Thus, Horner scheme is used for its evaluation.

When the three basic steps are executed, the only thing left is to put together all the performed actions into C code. On Figure 2.3 the block on the right may contain different parts depending on the used generation levels and therefore on the function specification, too. We discuss later on examples what is in the generated C code.

2.2.3 Properties Detection and Argument Reduction

We have seen in Section 2.1.1 several algorithms of argument reduction. They reduce the domain to a small one and for most of function flavors it becomes possible to use only one approximation polynomial. Thus, we use memory just to save its coefficients. Domain splitting procedure produces several subdomains from the initial one and computes polynomial approximations on each of the subdomains. In this case we store the splitpoints as well as all the polynomial coefficients. Therefore, it is better to use these property-based reduction algorithms than domain splitting for simple functions. However, these algorithms depend on the function to be implemented. The challenging point here is in construction of Metalibm: we tried to create a tool that does everything automatically without knowledge of the function it is trying to generate. The solution here is to detect automatically algebraic properties that will define the corresponding argument reduction scheme. Here is a list of currently detectable properties:

- exponential functions $f(x+y) = f(x)f(y)$;

- periodic $f(x+C) = f(x)$;

- logarithmic $f(x) + f(y) = f(xy)$;

- sinh-like functions family $f(x) = \beta^x - \beta^{-x}$;
- symmetric (both even and odd) $f(x) = f(-x)$; $f(x) = -f(x)$.

Other properties can be useful too and their detection may be easily added to Metalibm. The following examples of properties detection give an idea how this can be done automatically.

Exponential Function Detection

To exploit a property $f(x+y) = f(x)f(y)$ we first choose two distinct random points ξ and η in $[a, b]$, and the tool checks then if there exists $|\varepsilon| < \bar{\varepsilon}$ such that $f(\xi+\eta) = f(\xi)f(\eta)(1+\varepsilon)$. If not, the property is not true. Then we make a hypothesis that the function $f(x)$ to be implemented is $f(x) = \beta^x$, with an unknown base β. The generator knows neither the function, nor the base β. Applying logarithm to the hypothesis equation allows us to find the base. Therefore,

$$\beta = \exp\left(\frac{\ln(f(\xi))}{\xi}\right)$$

for some random $\xi \in [a, b]$. Only in this case it checks that the property is true up to the required accuracy, by computing

$$\widetilde{\varepsilon} = \left\| \frac{\beta^x}{f(x)} - 1 \right\|_\infty^{[a,b]}$$

and checking if $\widetilde{\varepsilon} \leq \bar{\varepsilon}$.

If the investigated function is exponential indeed, computation of this infinite norm is problematic. The function $g(x) = \beta^x - f(x)$ will be wobbling around zero staying in the band $(-\bar{\varepsilon}; \bar{\varepsilon})$. It is not the exact zero, as the computations are in FP arithmetic. We accept hypothesis about f if the functions $g + 2\bar{\varepsilon}$ and $g - 2\bar{\varepsilon}$ do not have zeros and stay in bands $[\bar{\varepsilon}, 3\bar{\varepsilon}]$ and $[-3\bar{\varepsilon}, -\bar{\varepsilon}]$ respectively.

Periodical Function Detection

The function f is called periodical with a period C if for the least constant C there holds $f(x + C) = f(x)$. The challenge for Metalibm is to find this period automatically.

Detection of periodical function requires more function evaluations in comparison with exponential detection. We start with a simple test: a function should have at least two local extrema to be periodic. If it is the case we choose the smallest period C from zeros of expression $f(\xi + C) - f(\xi)$ for some random $\xi \in [a, b]$. It is done in Metalibm with numerical zero search for expression $f(\xi + C) - f(\xi) = 0$. Final

decision to admit that the function was periodic is done with the computation of error

$$\left\| \frac{f(x+C)}{f(x)} - 1 \right\|_{\infty}^{[a,b]} \leq \bar{\varepsilon}.$$

The computation is done with the previously described technique.

Sinh-like family of functions

We have seen how to detect exponential functions family. Another interesting example to detect and to generate is the family of functions $f(x) = \beta^x - \beta^{-x}$. When $\beta = e$, $f(x) = 2\sinh(x)$. This is a composite function and its domain (full domain in mathematical sense) can be divided into five subdomains where the function can be implemented in different ways. It cannot be implemented as a subtraction of two exponentials around zero because of cancellations. For large (as well as for small) arguments one of the addends gets too small, so depending on the desired final accuracy, there are intervals, where only one exponential function should be returned. This reduces execution time, while we evaluate one function instead of evaluating two of them and performing subtraction.

The described family of functions is detected in the same way as exponentials, the base β can be determined with a numerical search for zero of the following function

$$g(x) = f(x) - \beta^x + \beta^{-x}.$$

The admission of hypothesis happens as in previous cases if the function $g(x)$ stays small and bounded by $(-\bar{\varepsilon}, \bar{\varepsilon})$.

Argument reduction is performed as follows. Suppose here $|\beta| > 1$, then for values $x \geq -0.5\log_{\beta}(\bar{\varepsilon}/3)$, $f(x) = \beta^x$. For $x < 0.5\log_{\beta}(\bar{\varepsilon})$, $f(x) = -\beta^{-x}$. For the case with $|\beta| < 1$, subdomains can be defined in the similar way. The rest of the domain has to be divided into three more intervals: in the case of cancellations around zero, we approximate the expression $\beta^x - \beta^{-x}$ by polynomial, and in two other symmetrical parts of the interval we may compute directly $\beta^x - \beta^{-x}$. So, in order to implement function $f(x) = \beta^x - \beta^{-x}$, Metalibm has to perform two recursive calls to generate β^x and β^{-x}, build an approximation polynomial and put all the generated parts together.

2.2.4 Domain Splitting and Approximation

Once algebraic property is detected an appropriate argument reduction is applied. However, it may happen that after the first level of code generation argument still needs to be reduced, for instance for symmetric functions. For some functions there is no efficient argument reduction procedures. Thus, having constraint on maximum allowable polynomial degree, the only way to reduce domain is to split it into several parts. The evaluation scheme should be simple and deterministic, so the subdomains

should not overlap. The need of splitting is checked on the second level of code generation (Figure 2.3) and we detail later how.

The problem statement for splitting procedure may be formulated as following: having a function f, an initial domain $[a, b]$ and the constraint on approximation polynomial degree $d_{\max}$, split the initial domain into non-overlapping intervals $I_0, I_1, \ldots, I_N$ so that on each of them we can approximate our function by a polynomial of low degree $d \leq d_{\max}$. We also take a lower bound for the subdomain length $w_{\min}$, this parameter is added to the function flavor specification.

Splitting may be uniform, hierarchical [58] or arbitrary [53]. Uniform splitting is relatively easy: having some large number N, we split the domain $[a, b]$ into N equal subdomains. This constant N may be too large if the function had in some region derivatives or high magnitude. For these large values of N we get a lot of subdomains. For regions where function derivative does not change a lot, uniform splitting produces too many subdomains with approximation polynomials of low degree. Thus, we may have a headroom between the actual polynomial degree and $d_{\max}$. Subdomains with low-degree polynomials might be merged together to increase the degree and therefore to save time and memory on the approximation.

The unique constant N cannot be used for all the functions and their flavors, it is inefficient. Different functions and even different flavors of the same function require different quantity of subdomains. Thus, this N should depend on the flavor to be implemented. Checking several splittings and merges of subdomains to determine the best N for uniform splitting may be expensive.

The hierarchical approach is more adapted for function behavior and should be easily implementable as lengths of subdomains are powers of two. Each time there is a need to split the domain, it is split twice, so the subdomains form a binary tree. Polynomial degrees are bounded with the parameter $d_{\max}$, so it is quite natural to approximate our function on each of the subdomains by a polynomial of this maximal allowable degree. This means that the evaluation scheme would be the same for each polynomial, and the quantity of the subdomains is minimal, therefore there are memory savings. The main idea of hierarchical splitting is to compute polynomial approximation of $d_{\max}$ on some domain until the approximation error gets less than the required error $\bar{\varepsilon}$. If the current error is larger than the required one, domain is split into two equal subdomains. In this way we get easy computable subdomains representable by the powers of two.

We propose a new algorithm to compute a non-uniform splitting: we split the domain only if it is impossible to approximate the function with a polynomial of maximum degree $d_{\max}$. We exploit approximation theory results for this: we may check whether some degree d is enough to approximate function f on some domain with an error not larger then $\bar{\varepsilon}$. This is slightly different of the classical approximation problem that may be formulated as following: having function f on domain $[a, b]$ and degree d compute the approximation polynomial and its accuracy ε.

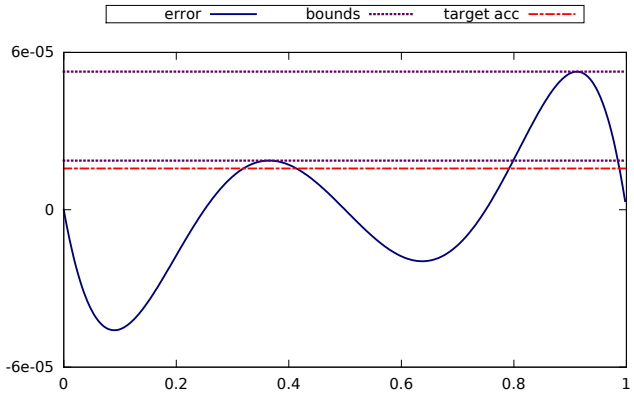


Figure 2.4: Illustration of de La Vallée-Poussin theorem. Domain has to be split

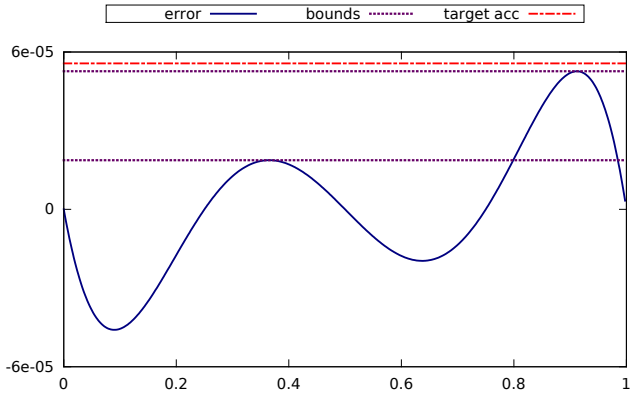


Figure 2.5: Illustration of de La Vallée-Poussin theorem. No splitting needed

The Base for a New Splitting Algorithm

The theorem of de la Vallée-Poussin mentioned in Section 2.1.2 is a base for our algorithm that decides if domain splitting is needed. We start with a polynomial p computed on Chebyshev nodes. Then we find minimal and maximal values for the expression $|f(x_i) - p(x_i)|$, where the points x_i are Chebyshev nodes. These values are the bounds from Theorem 2.3. When the desired accuracy $\bar{\varepsilon}$ is less than the found minimum, the degree d of this polynomial p is not sufficient, therefore domain has to be split. This is illustrated on Figure 2.4. Figure 2.5 shows an example when no splitting is needed: the bounds for error of polynomial approximation are smaller than the target error. When the specified accuracy is inside the bounds for polynomial approximation obtained from de la Vallée-Poussin theorem like in Figure 2.6, it is not clear whether the split is needed. The theorem does not give a value for the optimal error, but only its bounds. Therefore, we do not have a precise value for the optimal error to compare it with the target error. In this case a few Remez iterations are needed. The pseudocode for the described technique may be found on Algorithm 1.

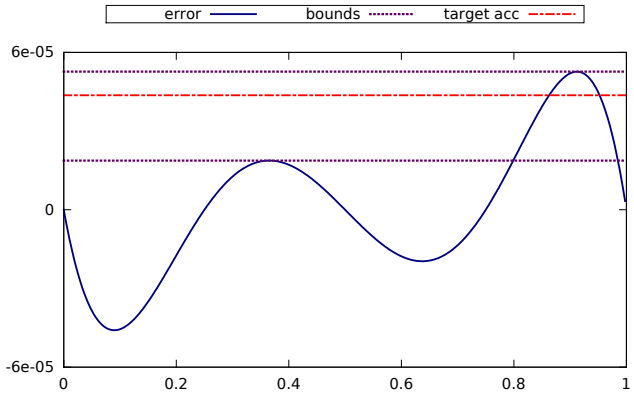


Figure 2.6: Illustration of de La Vallée-Poussin theorem. Remez iterations needed

Procedure `checkIfSufficientDegree`(f, I, $d_{\max}$, $\overline{\varepsilon}$):
Input : function f, domain $I = [a; b]$, bounds for degree $d_{\max}$, and accuracy $\overline{\varepsilon}$
Output: true in the case of success, false in the case of fail
$X \leftarrow$ `computeChebyshevNodes`(I, d_{max});
$p \leftarrow$ `computeApproximationOnChebyshevNodes`(f, X, d_{max});
$m \leftarrow \min_{x_i \in X} |f(x_i) - p(x_i)|$;
$M \leftarrow \max_{x_i \in X} |f(x_i) - p(x_i)|$;
if $\overline{\varepsilon} \geq M$ **then** $result$ = true;
if $\overline{\varepsilon} \leq m$ **then** $result$ = false;
if $\overline{\varepsilon} > m$ *and* $\overline{\varepsilon} < M$ **then**
    $p \leftarrow$`Remez`$(f, I, d_{max}, \overline{\varepsilon})$;
    $\delta \leftarrow$ `supnorm`$(f - p, I)$;
    $result \leftarrow \delta \leq \overline{\varepsilon}$;
end
return $result$;

Algorithm 1: Procedure to check approximation polynomial degree

Bisection Splitting

We take the problem statement for domain splitting procedure (see beginning of Section 2.2.4) and add one more condition. We want to split in such a way that corresponding polynomial degrees on each subdomain are as close to $d_{\max}$ as possible, and the difference between polynomial degrees on adjacent intervals is as small as possible.

As we have a limit on polynomial degree, it is a waste to use polynomials of degrees much lower than the given limit $d_{\max}$: we get too many subdomains and use memory to save too many of polynomial coefficients, while several subdomains could be merged.

For instance, on Figure 2.7 there are corresponding polynomial degrees for naive

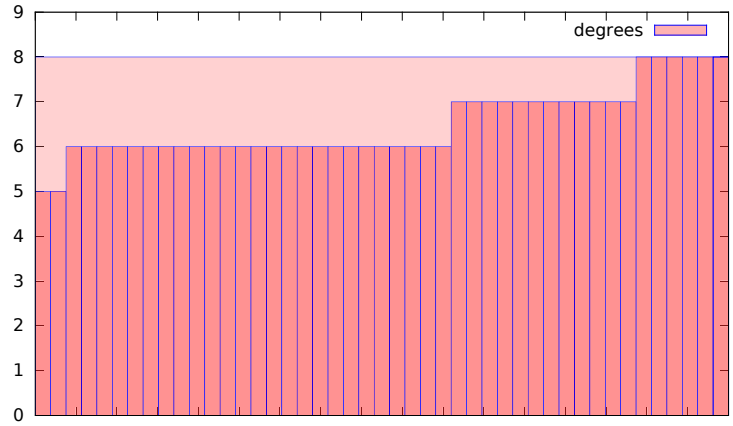


Figure 2.7: Polynomial degrees for uniform split for asin function on domain $[0; 0.85]$

Procedure `computeBisectionSplitting`(f, I, d, $\bar{\varepsilon}$):
Input : function f, domain $I = [a; b]$, max. degree d, target accuracy $\bar{\varepsilon}$
Output: list ℓ of points in I where domain needs to be split
if `checkIfSufficientDegree`$(f,\ I,\ d,\ \bar{\varepsilon})$ **then return** $\ell = [\,]$;
$m \leftarrow b$;
while *not* `checkIfSufficientDegree`$(f,\ [a; m],\ d,\ \bar{\varepsilon})$ **do** $m \leftarrow (a + m)/2$;
$J \leftarrow [m; b]$;
$\ell \leftarrow$**prepend**$(m,$ `computeOptimizedSplitting`$(f,\ J,\ d,\ \bar{\varepsilon}))$;
return ℓ;

Algorithm 2: Pseudocode for bisection splitting

uniform splitting: for asin function with $d_{\max} = 8$ we split the domain $[0; 0.85]$ into 50 equal subdomains. Requiring small differences between polynomial degrees on adjacent intervals implies the use of the same evaluation scheme for all the subdomains: we are going to use polynomials of the same (almost) degree.

So, we start checking whether the degree $d_{\max}$ is sufficient to approximate our function f on $[a, b]$ with error bounded by $\bar{\varepsilon}$. If yes, no split is needed, otherwise, we divide the whole domain into two equal subdomains and continue recursive calls of `checkIfSufficientDegree` to the left subdomain. This is a classical application of bisection, and the procedure returns a list of splitpoints. Pseudocode for bisection splitting is in Algorithm 2.

For the same asin example bisection splits the domain into 23 subdomains and the degrees diagram is on Figure 2.8. The quantity of subdomains is reduced in comparison with uniform splitting and the diagram of the corresponding degrees gets more uniform. For this example degrees on the adjacent subdomains differ maximum by one. However, adjacent degrees are not too unified: there are many of them less than the bound $d_{\max}$. Therefore, if we slightly change the borders of our subdomains, degrees attain their bound $d_{\max}$ and we might get even less subdomains.

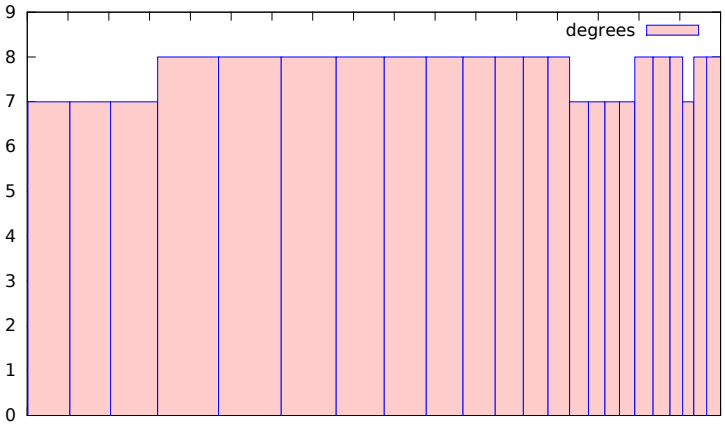


Figure 2.8: Polynomial degrees for bisection split for asin function on $[0; 0.85]$

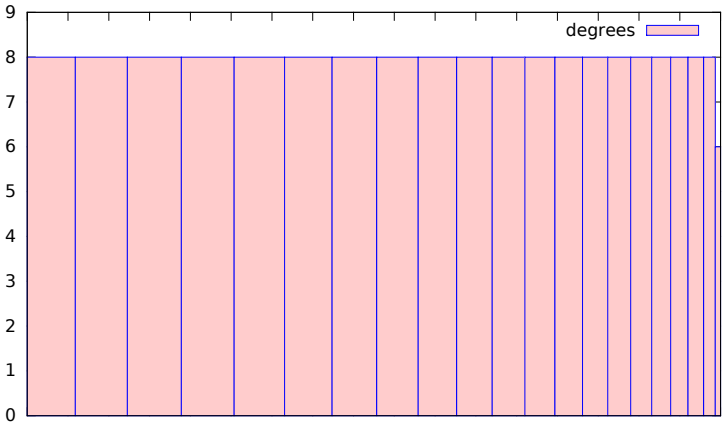


Figure 2.9: Polynomial degrees for improved bisection splitting for asin on $[0; 0.85]$

Improvement of Bisection Splitting

Bisection produces less intervals than the uniform 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 by some value δ as it is shown on Algorithm 3. Thus, the diagram of degrees gets more uniformed, we get less subdomains. However, correlation between the splitpoints vanishes: this splitting may be called "arbitrary". This means that the only way to implement the reconstruction is the execution of *if-else* statements.

As it is an improvement of bisection, the algorithm contains two procedures: bisection and enlarging. As soon as we find a leftmost suitable interval, we try to move its right border. Then this moved right border is added to a list of splitpoints. The value δ determines how far we try to move the right border. In the while loop (line 3 of Algorithm 4), we have a constraint that this value δ has to stay larger than some constant $\bar{\delta}$. This is a heuristic constant and its currently used value is

$\overline{\delta} = w_{\min}/2$. We leave for the future work the search for the best constant $\overline{\delta}$ as well as the initial value of δ (in line 2 of Algorithm 4).

We admitted intervals from left to right, so we may call the described scheme *left-to-right approach*. The same idea holds for splitting from right to the left: with bisection we find the rightmost suitable interval and then we move down its lower border. The weak point of our algorithm is that there is no control of the polynomial degree for the last subdomain (the rightmost for left-to-right splitting or the leftmost for right-to-left splitting). Theoretically there is nothing to prevent obtaining very small last subdomain with a polynomial of low degree (one or two). During tests this situation was not observed. However, it should be taken into account. The two approaches may be combined to get a set of tolerable intervals containing the splitpoints. This is left for future work.

```
1 Procedure computeOptimizedSplitting(f, I, d, ε̄):
  Input  : function f, domain I = [a; b], max. degree d, target accuracy ε̄
  Output: list ℓ of points in I where domain needs to be split
2 if checkIfSufficientDegree(f, I, d, ε̄) then return ℓ = [ ];
3 m ← b;
4 while not checkIfSufficientDegree(f, [a; m], d, ε̄) do m ← (a + m)/2 ;
5 s ←enlargeDomain(f, [a; m], [m; b], ε̄, d) ;
6 ℓ ←prepend(s, computeOptimizedSplitting(f, [s; b], d, ε̄));
7 return ℓ;
```

Algorithm 3: Pseudocode for our improved bisection splitting

```
1 Procedure enlargeDomain(f, I, J, ε̄, d):
  Input  : function f, domain I = [a; b], remaining domain J = [b; c], ε̄, d
  Output: optimal splitpoint location s ∈ J
2 δ ← (b − a)/3;
3 while δ > δ̄, δ̄ a constant, and b < c do
4 |   s ← b + δ;
5 |   while checkIfSufficientDegree(f, [a; s], d, ε̄) do s ← s + δ ;
6 |   s ← b − δ;
7 |   δ ← δ/2;
8 end
9 return s;
```

Algorithm 4: Procedure of enlarging of the suitable subdomain

For the asin example improved bisection method produces 21 subdomains, Figure 2.9 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. Some other examples that compare bisection with our improved bisection can be found in Table 2.2.

name	function f	target accuracy	domain I	degree bound
f_1	asin	$\bar{\varepsilon} = 2^{-52}$	$I = [0, 0.75]$	$d_{\max} = 8$
f_2	asin	$\bar{\varepsilon} = 2^{-45}$	$I = [-0.75, 0.75]$	$d_{\max} = 8$
f_3	erf	$\bar{\varepsilon} = 2^{-51}$	$I = [-0.75, 0.75]$	$d_{\max} = 9$
f_4	erf	$\bar{\varepsilon} = 2^{-45}$	$I = [-0.75, 0.75]$	$d_{\max} = 7$
f_5	erf	$\bar{\varepsilon} = 2^{-43}$	$I = [-0.75, 0.75]$	$d_{\max} = 6$

Table 2.1: Flavor specifications

measure	f_1	f_2	f_3	f_4	f_5
subdomains in bisection	24	15	9	12	39
subdomains in improved bisection	18	10	5	8	25
subdomains saved	25%	30%	44%	30%	36%
coefficients saved	42	31	27	24	79
memory saved (bytes)	336	248	216	192	632

Table 2.2: Table of measurements for several function flavors

When the domains are reduced, Metalibm generates code to evaluate Remez-like approximation polynomial for a small domain and launches Gappa proof for approximation error.

2.2.5 Reconstruction

The goal for splitting and argument reduction is to reduce the degree of polynomial approximation. Polynomial coefficients are computed on a small domain. Reconstruction procedure aims to give the values of the function f on a large initial domain through the evaluation of polynomial(s) on a small domain. When the argument reduction was done only with property-based algorithms (for instance for exponential) reconstruction is the process of applying the backward transition from p to f. After splitting we get the list of splitpoints and the subdomains $I_0, \ldots, I_N$. Thus, the transition from the evaluation of polynomial to function values lies in determination of the subdomain index k that contains the current input $x \in I_k$. This is sometimes called *domain decomposition*.

While for property-based reduction reconstruction is simple, this section covers reconstruction for implementations with piecewise approximations. Decomposition process depends on the way of domain splitting. For uniform splitting it is straightforward. We split the domain $[a, b]$ into N parts, so the splitpoints may be represented as $\{a + ih\}_{i=0}^{N}$, where $h = (b - a)/N$. For a given input $x \in [a, b]$, the corresponding subdomain and therefore the index of approximation polynomial may be determined as $\lfloor \frac{x-a}{h} \rfloor$. For arbitrary splittings, however, this is commonly done with the execution of *if-else* statements (see Listing 2.1).

Since the prevalence of SIMD instructions on modern processors, the code generation of vectorizable implementations is of big interest. A usual way to vectorize an algorithm is to get rid of branching. For exponential and logarithmic functions vectorized loop calls reduce the computation time by 1.5-2 times. With our arbitrary splitting we use *if-else* statements to determine the corresponding subdomain I_n that contains the input value x and then with this index n we get the right polynomial coefficients. We started research in generating vectorizable implementations with construction of a mapping function $M(x)$ that allows to perform domain decomposition without branching.

```
/* compute i so that a[i] < x < a[i+1] */
        i=31;
        if (x < arctan_table[i][A].d) i-= 16;
        else i+=16;
        if (x < arctan_table[i][A].d) i-= 8;
        else i+= 8;
        if (x < arctan_table[i][A].d) i-= 4;
        else i+= 4;
        if (x < arctan_table[i][A].d) i-= 2;
        else i+= 2;
        if (x < arctan_table[i][A].d) i-= 1;
        else i+= 1;
        if (x < arctan_table[i][A].d) i-= 1;
        xmBihi = x-arctan_table[i][B].d;
        xmBilo = 0.0;
```

Listing 2.1: Code sample for arctan function from `crlibm` library

Polynomial-based Reconstruction Technique

We propose to use a polynomial to find a mapping function $M(x)$. Having a set of subdomains $\{I_k\}_{k=0}^{N-1}$ or of splitpoints $\{a_k\}_{k=0}^{N}$ and argument $x \in [a, b]$ the problem consists in obtaining the index k of a corresponding subdomain $x \in [a_k, a_{k+1}]$. Thus, our mapping function $M(x)$ should return the index of the corresponding subdomain for each input value from $[a, b]$:

$$M(x) = k,\ x \in I_k,\ k = 0, 1, \ldots, N.$$

The function $M(x)$ is a piecewise-constant function as it is shown on Figure 2.10. We propose to find a polynomial $p(x)$ on $[a, b]$ such that

$$M(x) = \lfloor p(x) \rfloor,\ x \in [a, b].$$

An example of such polynomial is shown on Figure 2.11. It may not be a strictly monotonic function, it might have zeros in its derivative. The main point is that

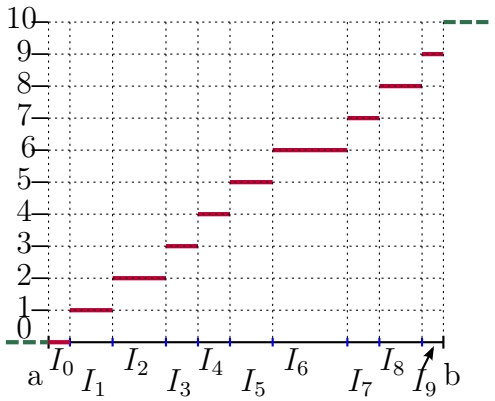


Figure 2.10: Piecewise-constant mapping function $M(x)$

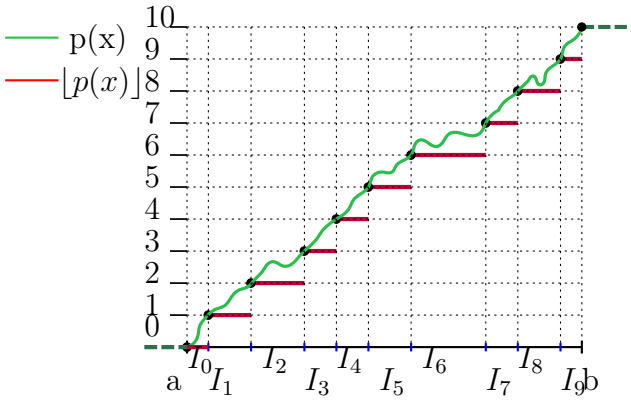


Figure 2.11: Mapping function and a corresponding polynomial $p(x)$

$\lfloor p(x) \rfloor$ returns the step function M. Thus, the suitable polynomial p have to verify the following conditions:

$$p(x) \in [k, k+1), \quad x \in [a_k, a_{k+1}]. \tag{2.7}$$

We may compute p as an interpolation polynomial that passes through the abscissas $\{a_k\}$ and ordinates $\{k\}$. However, interpolation techniques guarantee only that $p(a_k) = k$ by construction of the polynomial, thus the condition (2.7) has to be checked *a posteriori*. This can be done with the evaluation of this polynomial $p(x)$ over the interval $[a_k, a_{k+1}]$. 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 give the same value $p(a_k) = k$, and 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.

Interpolation Polynomial Let us remember the classical interpolation problem [5]. Having a set of points $\{x_i, y_i\}_{i=0}^{N}$ we are looking for a degree-N polynomial

$p = c_0 + c_1 x + \ldots + c_N x^N$ such that $p(x_i) = y_i$ for all integer $i \in [0, N]$. Mathematically, this problem is equivalent to the solution of a system of linear equations with Vandermonde's matrix:

$$\begin{pmatrix} 1 & x_0 & \cdots & x_1^N \\ 1 & x_1 & \cdots & x_1^N \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_N & \cdots & x_N^N \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_N \end{pmatrix} \tag{2.8}$$

Solving this system of linear algebraic equations explicitly is one of the ways to find interpolation polynomial. As it may have huge conditional number, we use interpolation through divided differences in Metalibm.

Taking into Account FP Roundings We take the couples $\{a_k, k\}_{k=0}^N$ as interpolation points. The polynomial p has FP coefficients, therefore 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 (2.7) have to be modified a little. As the set of FP numbers is discrete, for a given FP 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 (2.7) should be narrowed to the following:

$$p(x) \in [k, k+1), \text{ where } x \in [\mathrm{succ}(a_k), \mathrm{pred}(a_{k+1})] \subset I_k,\ 0 \le k \le N-1. \tag{2.9}$$

The conditions for the splitpoints should be added then.

$$p(x) \in [k-1, k+1), \text{ where } x = a_k, k = 1, \ldots, N-1 \tag{2.10}$$

For $k = 0$ or $k = N$, conditions for $p(a_k)$ do not change: it should stay in $p(a_k) \in [k-1, k)$. The modified conditions for the polynomial ranges are shown on Figure 2.12 with filled rectangles.

Choice of Interpolation Points Interpolation points may be chosen in several different ways out of the set of splitpoints $\{a_k\}_{k=0}^N$. We compute four different polynomials. First, we may use "inner" polynomial with $N-1$ points $\{a_k\}_{k=1}^{N-1}$, so without taking into account the first and the last splitpoints that are the bounds of the implementation domain $a_0 = a, a_N = b$. 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 the $N+1$ splitpoints. When *a posteriori* conditions are not verified for all the four polynomials (2.9)-(2.10), it is a symptom of failure. We may add some interpolation points and check polynomials computed for the enlarged set of 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 [5], which means that the conditions (2.9)-(2.10) are rarely verified. We

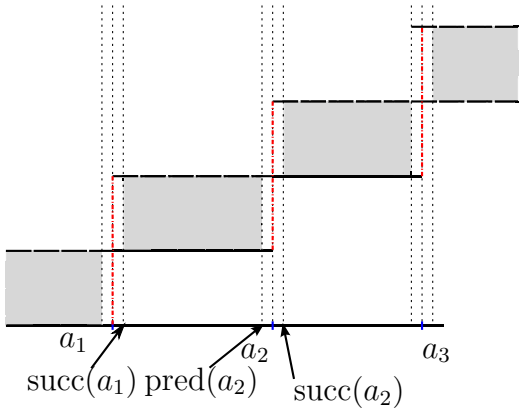


Figure 2.12: Modified floating-point conditions for polynomial.

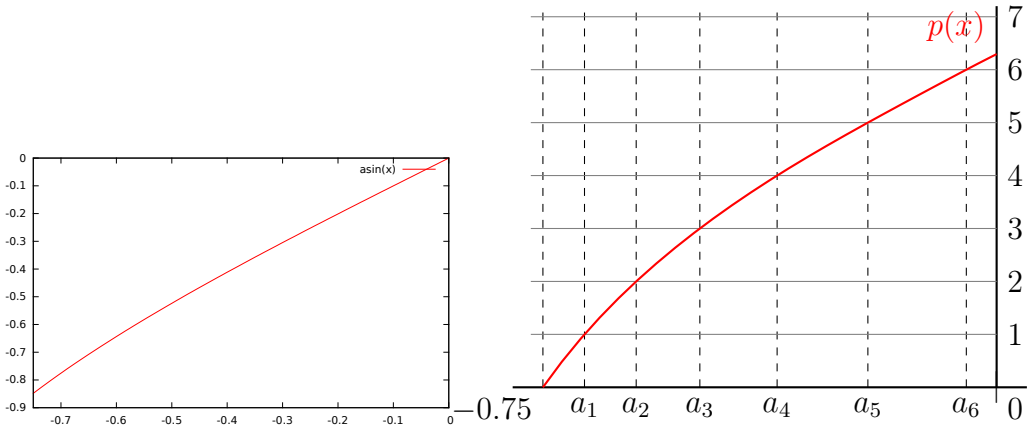


Figure 2.13: Example for asin(x) flavor and its polynomial for mapping function (left)

also add a parameter to limit polynomial degree for this mapping function. When this mapping function is not needed we may initialize it with a small value (one for example) to prevent Metalibm of unnecessary computations.

Examples Here we show several examples of successful computation of polynomial for mapping function M. The function to be generated is asin(x) on $[-0.75, 0]$ with required accuracy $\bar{\varepsilon} \leq 2^{-48}$, limit for approximation polynomial degree is $d_{\max} = 10$. The conditions (2.9)-(2.10) are verified for the "left polynomial" of degree six. Plots of generated function and of the polynomial for mapping are on Figure 2.13.

Another successful example may be computed for generation of asin(x) on $[-0.8, 0]$ with required accuracy $\bar{\varepsilon} \leq 2^{-45}$ and for approximation polynomial degree not larger than $d_{\max} = 10$. For this example the "inner" interpolation is used, so degree of polynomial for mapping function is four (see Figure 2.14).

For error function erf(x) on the domain $[-0.9, 0]$ with target accuracy 2^{-45} and

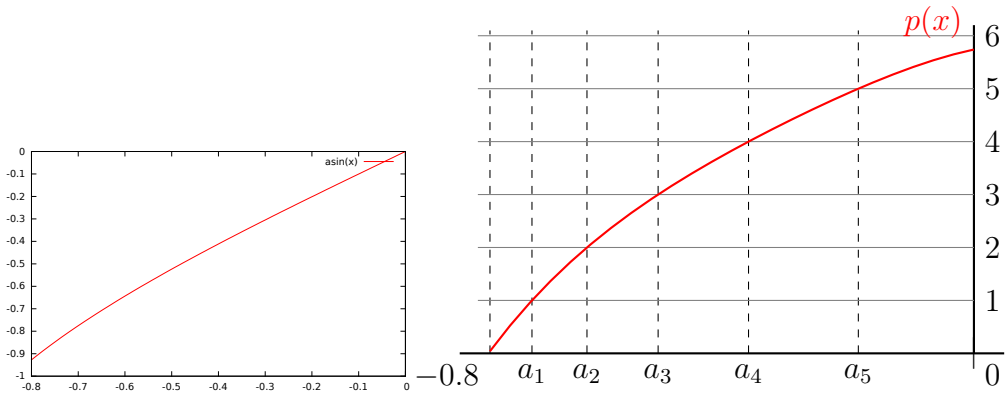


Figure 2.14: Example for asin(x) flavor and its polynomial for mapping function (inner)

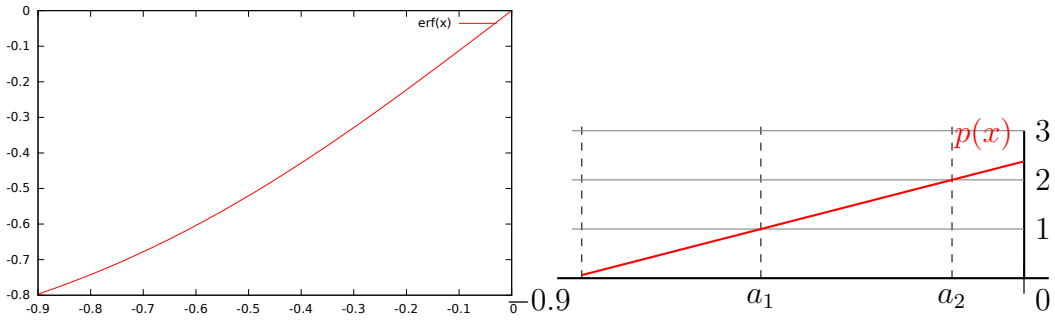


Figure 2.15: Example for erf(x) flavor and its polynomial for mapping function (inner)

approximating polynomial of degree not larger then 10, "inner" interpolation is used. After symmetry detection, the domain was reduced to $[-0.9, 0]$ and then it was split into three subdomains. The polynomial for mapping function p is a linear function shown on Figure 2.15.

Conditions (2.9)-(2.10) are essential for our polynomial and as we are checking them 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.

For example, for atan flavor on $[0, \pi/2]$ with accuracy bounded by $\bar{\varepsilon} \leq 2^{-40}$ with maximum degree of the approximation polynomial $d_{\max} = 8$ it is not possible to find a polynomial mapping function for reconstruction. The domain is split into seven subdomains; even the polynomial passing through all these splitpoints does not verify the conditions (2.9)-(2.10). It is illustrated on Figure 2.16, we see that it crosses two lines in the first subdomain, in the second subdomain it crosses the lower border and then decreases. Metalibm tried to add an interpolation point and to recompute the polynomial. It added the point from the first subdomain with the

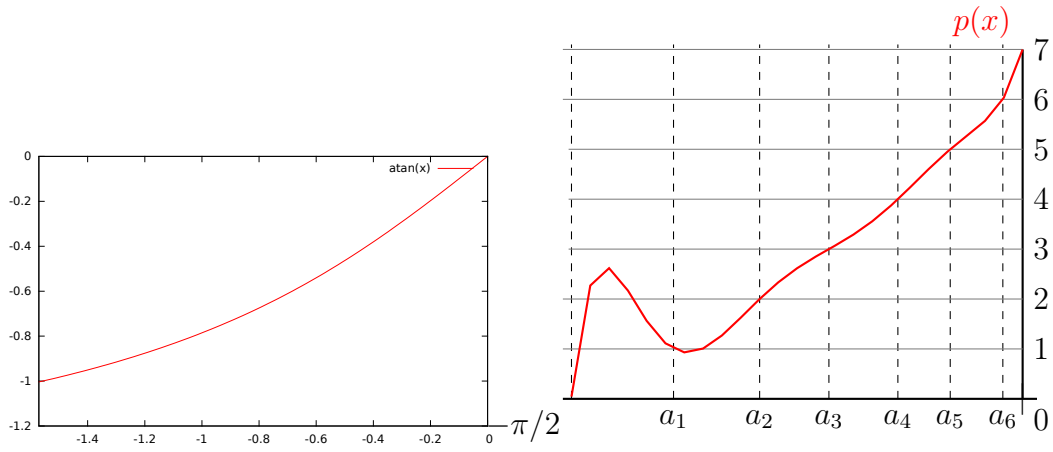


Figure 2.16: Example for atan(x) flavor. Our method fails to find a mapping function.

largest derivative. However, it did not help: polynomial p slightly exceeds the line $y = 1$ in the first subdomain. It cannot be seen on a plot as this extension is too small.

Towards *a priori* Conditions The interpolation problem is formulated with the system of linear algebraic equations (2.8). We solved it for splitpoints and integer numbers that are indexes of the subdomains. The *a posteriori* conditions (2.9)-(2.10) are about the admissible intervals for polynomial values. Thus, we can pass from *a posteriori* check to *a priori* considering intervals instead of points: on abscissas we take subdomains and intervals $[k, \mathrm{pred}(k+1)]$ on ordinates. Then, the task is almost the same: system of linear equations with unknown coefficients $c_0, \ldots, c_N$. Instead of the numbers x_i, y_i we operate intervals in system (2.11).

$$\begin{pmatrix} 1 & \mathbf{x}_0 & \cdots & \mathbf{x}_0^N \\ 1 & \mathbf{x}_1 & \cdots & \mathbf{x}_1^N \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \mathbf{x}_N & \cdots & \mathbf{x}_N^N \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_N \end{pmatrix} = \begin{pmatrix} \mathbf{y}_0 \\ \mathbf{y}_1 \\ \vdots \\ \mathbf{y}_N \end{pmatrix} \tag{2.11}$$

Depending on predicates $\forall$ and $\exists$ there are different tasks to solve with one system of linear interval equations [76]. The two problems should be considered in our case: search for tolerance or united solution set.

Definition 2.1 (Tolerance solution set)**.** *Let be* $\mathbf{X}c = \mathbf{y}$ *an interval linear system, then the following set is called its tolerance solution.*

$$\Xi_{tol} = \left\{ c \in \mathbb{R}^{N+1} \,|\, \forall X \in \mathbf{X}, \quad \forall y \in \mathbf{y}, \quad Xc = y \right\}$$

Definition 2.2 (United solution set)**.** *Let be* $\mathbf{X}c = \mathbf{y}$ *an interval linear system, then the following set is called its united solution.*

$$\Xi_{uni} = \left\{ c \in \mathbb{R}^{N+1} \,|\, \exists X \in \mathbf{X}, \quad \exists y \in \mathbf{y}, \quad Xc = y \right\}$$

In classical approach of interval analysis solution vector has interval elements. By the sense of problem statement, solution vector contains the coefficients of the polynomial for our mapping function M. Therefore, we are not interested in search of all possible values for its coefficients, we need only one vector for its values $c_0, \ldots c_N$.

The tolerance solution set of the system (2.11) may be found in polynomial time, but it can be empty. In this case the united solution set may be found, but this problem is NP-hard [75]. 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.

Connection between Domain Splitting and Reconstruction One can notice that the problems of computing polynomial for this mapping function M come from the fact of arbitrary splitting. We tried to split domain optimally: to maximize the polynomial degree on each of the subdomains and to minimize the quantity of these subdomains. This creates arbitrary splitting and makes polynomial-based reconstruction difficult. This type of reconstruction can be easily made for uniform splitting (a linear function) that creates too many subdomains. Thus, there is a certain connection between splitting and reconstruction. When we cannot find a suitable polynomial for vectorizable reconstruction, we have to return to splitting and recompute it in other way. There is no information on how many of these returns are needed to compute at the same time a quasi-optimal splitting and a polynomial for reconstruction. Interval arithmetic approach could be used here too: instead of the fixed splitpoints we may compute some intervals that contain these splitpoint. Then, moving the splitpoints over such intervals may give us a suitable combination of splitting and reconstruction. However, this does not give strong guarantees of existence of polynomial for reconstruction. Establishing of this connection between splitting and polynomial reconstruction is left for future work on Metalibm. A new parameter might be added too: if users are interested in vectorizable implementations, there is probably no need to find an optimal split. And if there is no need in vectorization, the split should be computed optimally and this complex reconstruction step should be avoided.

Conclusion

The work on generation of vectorizable implementations has started. Our approach of replacing branches by polynomials was already published in [52]. As it does not give any guarantee of successful computation of mapping function, it has to be

improved. There are two main strategies for that. The first one is establishing of the connection between domain splitting and reconstruction procedures. And the second one is to use interval arithmetic in reconstruction and even in splitting. Generation of vectorizable implementations is in priority for Metalibm, so work on improvement of described method will be started in the nearest future.

2.2.6 Several examples of code generation with Metalibm

In this section we illustrate generation process on several examples. These examples illustrate how to fill the rectangle "implementation" on Figure 2.3. Besides producing the implementations Metalibm also runs the generated code and plots the current function flavor as well as the relative error of the implementation.

1. Approximation by one polynomial.
   We try to generate $\exp(x)$ on a small domain $[0, 0.3]$ with accuracy bounded by $\bar{\varepsilon} = 2^{-53}$ and polynomial degree not larger than 9. Metalibm detects that one polynomial will be enough to approximate this function with the specified accuracy on the specified domain. Thus, generated code only consists of polynomial coefficients and polynomial evaluation function. This function flavor is about 1.5 times faster than the standard exp function from the glibc libm.

2. Properties-based reduction and approximation.
   We enlarge domain from the previous example to $[0, 5]$ and set $t = 4$ for table (the table size is 2^t). The family of exponential functions is detected and domain is reduced to $[-\log(2)/32, \log(2)/32]$. Then Metalibm passes to approximation level. We find in the produced code constants, table, polynomial coefficients, routine to reduce domain, to evaluate the polynomial and to reconstruct the function. The obtained code for this function 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. Properties-based reduction, domain splitting and approximation.
   For some function flavors all the three levels of code generation are used. One of the examples is $\sin(x)$ on $[-10, 10]$ with accuracy 2^{-40} approximated by polynomials of degree not larger than 8. Metalibm detects first periodicity and reduces domain to $[-\pi, \pi]$. It detects also the need of triple-double arithmetic [56]. Then it detects odd symmetry and reduces domain even twice more: $[-\pi, 0]$. Afterwards domain splitting procedure starts. This twice reduced domain is split then into 9 smaller subdomains. Our reduced domain is too big for the sin implementation. There are specific property-based argument reduction schemes for sin that allow to reduce the range even more [36, 69]. Thus, while the libm sin is executed within 15 - 40 cycles, our implementation needs more that 1000 cycles.

4. Composite function example.
   We generate code for $\tan(\mathrm{erf}(x))$ on $[-2, 2]$ with polynomial maximum degree 8 and accuracy $\bar{\varepsilon} = 2^{-45}$. We ask Metalibm not to perform function decomposition, therefore the approximation will be computed for the whole function tan(erf). Metalibm detects symmetry and reduces domain to $[-2, 0]$. Then it splits the domain into 16 small subdomains. The corresponding polynomial degrees are almost all equal to eight, except the last one which is five. The libm code is executed within 400-500 cycles for the most cases. Running our code takes between 600 and 700 cycles. In terms of accuracy codes give almost the same result.

5. Sigmoid function. We try to generate code for 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 into 22 subintervals.

Metalibm performs three main steps of function implementation automatically. However, there is the very first step that is not treated by Metalibm for the moment: filtering of special cases. For some function flavors (functions on small domain for instance) it is not needed, therefore the generated code may be used directly. For a complete replacement of implementations from standard libms manual filtering of special cases needs to be added. Automatizing this step is left for future work.

2.2.7 Conclusion and Future Work

In previous sections we discussed the problem of code generation for mathematical function implementations. It was shown that currently available libms should provide users with more choices. As the quantity of all these choices is tremendous the code generator of parametrized function implementations is of big interest. Metalibm generates implementation for mathematical functions automatically. Moreover, functions to be generated are parametrized (specific domain, accuracy, etc). Metalibm is a black-box generator: we can pass an arbitrary function as a parameter, there is no fixed dictionary of available functions to generate. The only requirement for the function to be generated is that it should be continuous with its few derivatives. Accuracy of the produced code is guaranteed by construction.

Metalibm has evolved a lot since the first studies on automatization of function implementations. It detects automatically the needed precision for all inner computations to achieve the specified accuracy. It detects algebraic properties to use specific range reduction procedure, it decides if the further domain splitting is

needed. Domain splitting was improved: the generator tries to split domain optimally reducing the headroom between the given limit on the degree of approximation polynomial $d_{\max}$ and actual degree on the subdomain. This causes memory saves on storing the splitpoints and polynomial coefficients. The work on producing vectorizable implementations has started. It is based on replacement of branching by polynomials. Our method does not guarantee the possibility of vectorizable code generation but there are several ways to change it and improve the method. The two possible ways to improve our vectorization procedure are

1. transition from a posteriori condition check to a priori conditions with the use of interval arithmetic

2. establishing connection between splitting procedure and reconstruction.

Both of them are left for future work. Besides that, there is still no automatic filtering of special cases (infinities, large inputs producing overflows, etc.) that should be added soon. There may be more specific argument reduction procedures. The link between splitting and reconstruction has to be found in the nearest future. The supported parameter list can be enlarged too.

We mentioned that there were two use-cases for Metalibm. Our product is a fully automated generator. However, there exists analogue of Metalibm by N. Brunie and F. de Dinechin. It was developed as an assistant tool for libm programmers. However, it is hard to separate the two approaches distinctly. Based on the same software,

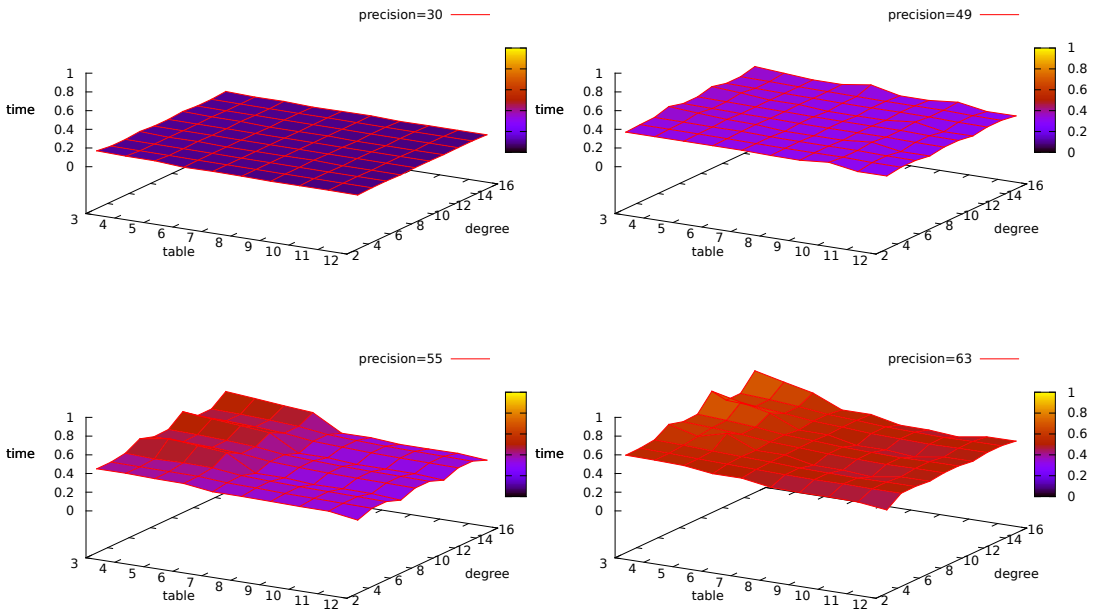


Figure 2.17: Performance measures for exp flavors

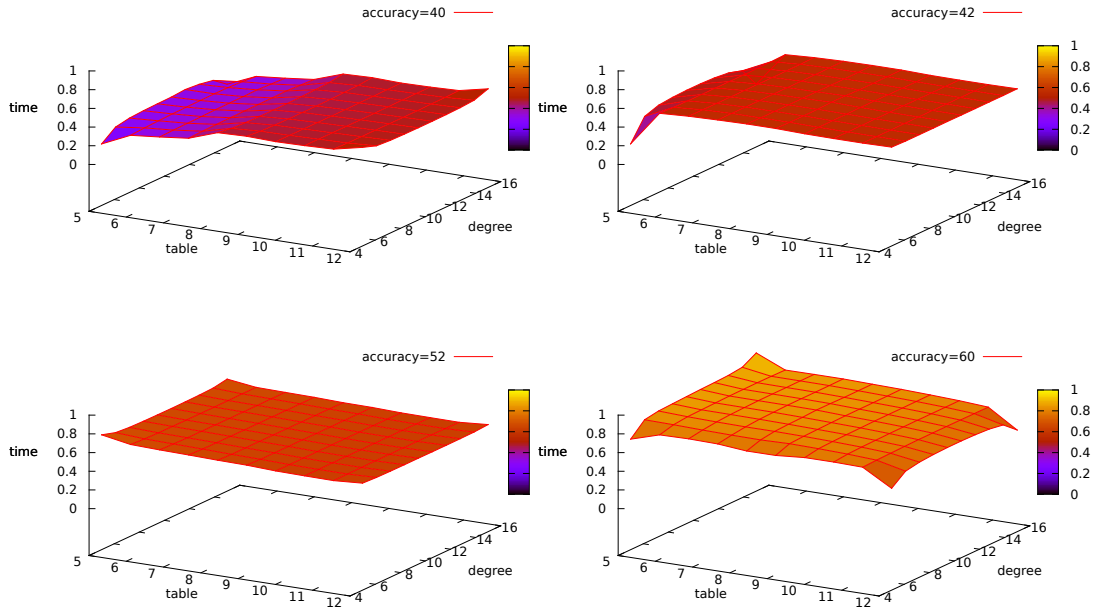


Figure 2.18: Performance measures for log flavors

they contain the same basic bricks, for instance, generator of approximation schemes or of C11 functions. The ambitious goal of the whole ANR project is to integrate the two approaches. Some algorithms from Metalibm can be reused by other code generators. For example, semi-automatic generator of special functions needs to split implementation domains in the same manner as Metalibm [57].

The possibility of automatic generation of different flavors gives an additional bonus. Various flavors of one functions may be generated and measured in performance. Then the generated implementation with the best combination of parameters and performance should be used. For example, on Figure 2.17 there are four plots of performance relatively to the demanded accuracy, maximum degree and table size. Time here is a relative value, it was scaled to fit into $(0, 1]$. On Figure 2.18 there is the same example for logarighm function. Another bonus of Metalibm is generation of composite functions. We may use the only one approximation for a composite function. In standard libms there are several function calls performed in this case.

CHAPTER 3

Mixed-Radix Arithmetic and Arbitrary Precision Base Conversions

A mathematician is a machine for turning coffee into theorems.

Alfréd Rényi[1]

This section is devoted to mixed-radix arithmetic, so to the research on operations that mix the inputs and the output of different radices. For instance, addition of binary and decimal FP number with the result in binary. We present in Section 3.2 an atomic operation for radix conversion [55] with integer computations. Then we provide the novel algorithm to convert a character sequence representing decimal FP number to its binary IEEE754 representation in Section 3.3. Conversion operation will be reused in this algorithm. This is a re-entrant algorithm with precomputed memory consumption. We finish the chapter with the worst cases search for mixed-radix fused multiply-add or FMA(Section 3.4).

3.1 Preface to Mixed-Radix Arithmetic

IEEE754-1985 Standard defined and required only binary arithmetic. The first attempt to standardize decimal arithmetic was done in 1987 with IEEE854 standard. However, it was never implemented and it did not allow to mix radices within one FP operation. The revision of the IEEE754 Standard added decimal FP formats and operations in 2008. However, the worlds of decimal and binary arithmetic are not supposed to be mixed by the Standard. On the junction of human and machine arithmetic there are always decimal-binary and binary-decimal conversions [13, 37, 78].

[1]Alfréd Rényi (1921 - 1970) was a Hungarian mathematician who made contributions in combinatorics, graph theory, number theory but mostly in probability theory. This quotation is often attributed incorrectly to Paul Erdős, but Erdős himself ascribed it to Rényi.

Conversions are inevitable for financial applications too: the inputs are in decimal and the computations may use some often-used constants stored in binary.

FP radix conversion (from binary to decimal and vice versa) is a widespread operation, the simplest examples are the scanf and printf-like functions. It could also exist as an operation for financial applications or as a precomputing step for mixed-radix operations. The radix conversion is used in FP number conversion operations, and also in scanf and printf operations. The current implementations of scanf and printf are correct only for one rounding mode and allocate a lot of memory. In this chapter we develop a unified atomic operation for the conversion, so all the computations can be done in integer with the precomputed memory consumption.

As mixed-radix arithmetic almost does not exist for the moment and as we are going to prove some theorems, we introduce the corresponding notations first. According to Def. 1.2, FP number may be represented as $\beta^E m$ where β is radix and mantissa m is bounded by $\beta^{p-1} \leq m \leq \beta^p - 1$. So, we denote binary FP numbers of precision p_2 as $2^E m$ and decimals with decimal precision p_{10} as $10^F n$. We call binary arithmetic operation $\diamond$ a mixed-radix operation, when the operands x, y and the result z are not all in the same radix:

$$z = x \diamond y.$$

As $\diamond$ is a binary operation, it has eight variants depending on the radix. A ternary operation such as FMA has three inputs, therefore sixteen different variants to implement. As we cannot study such a great number of different cases one by one, we have to find a unified way of handling them.

Mixed-radix operations may be considered as a generalization of the operations defined in IEEE754-2008 Standard. Two variants for each mixed-radix operations are already implemented. These are pure binary or pure decimal versions that do not actually mix the radices. Both binary and decimal FP representations may be unified to a mixed-radix one. Decimal mantissa n can be transformed into a binary FP number $2^E m$ of the form Def. 1.2. And the exponent part 10^F can be factorized as $5^F \cdot 2^F$. Thus, decimal FP number is representable in a form of

$$10^F n = 5^F 2^{F+E} m.$$

Taking $F = 0$ we get a binary FP number. As we are going to deal with bulky formulas, we take $2^E 5^F m$ as a mixed-radix notation with binary mantissa m bounded by one binade $2^{p-1} \leq m \leq 2^p - 1$, $E \in \mathbf{E}$, $F \in \mathbf{F}$. The numerical values of p and intervals $\mathbf{E}, \mathbf{F}$ depend on the formats used and will be given later in this section.

3.2 Radix Conversion

While radix conversion is a very common operation, it comes in different variants that are mostly coded in ad-hoc way in existing code. However, radix conversion always breaks down into two elementary steps: determining an exponent of the output radix and computing a mantissa in the output radix. Section 3.2.1 gives an overview of the 2-steps approach of the radix conversion, Section 3.2.2 contains the algorithm for the exponent computation, Section 3.2.3 presents a novel approach of raising 5 to an integer power used in the second step of the radix-conversion that computes the mantissa. Section 3.2.4 contains accuracy bounds for the algorithm of raising five to a large integer power, Section 3.2.5 describes some implementation tricks and presents experimental results.

3.2.1 Overview of the Two-steps Algorithm

Conversion from a binary FP representation $2^E \cdot m$, where E is the binary exponent and m is the mantissa, to a decimal representation $10^F \cdot n$, requires two steps: determination of the decimal exponent F and computation of the mantissa n. The conversion back to binary is pretty similar except of an extra step that will be explained later. Here and after consider normalized mantissas n and m: $10^{p_{10}-1} \leq n \leq 10^{p_{10}} - 1$ and $2^{p_2-1} \leq m \leq 2^{p_2} - 1$, where p_{10} and p_2 are the decimal and binary precisions respectively. We call the intervals $[2^{p_2-1}; 2^{p_2} - 1]$ and $[10^{p_{10}-1}; 10^{p_{10}} - 1]$ a binade and a decade. The exponents F and E are bounded by some values depending on the IEEE754-2008 format (see Table 3.1 for more details).

In order to enclose the converted decimal mantissa n into one decade, for a certain output precision p_{10}, according to Def. 1.6 the decimal exponent F has to be computed as follows:

$$F = \left\lfloor \log_{10}(2^E \cdot m) \right\rfloor - p_{10} + 1. \tag{3.1}$$

The most difficult thing here is the evaluation of the logarithm: as the function is transcendental, the result is always an approximation and a function call to logarithm evaluation may be expensive. We are going to present an algorithm that computes

format	exponent range	mantissa range
binary 32	$[-172, 104]$	$[2^{23}, 2^{24} - 1]$
binary 64	$[-1126, 971]$	$[2^{52}, 2^{53} - 1]$
binary 128	$[-16606, 16270]$	$[2^{112}, 2^{113} - 1]$
decimal 32	$[-107, 90]$	$[10^6, 10^7 - 1]$
decimal 64	$[-413, 369]$	$[10^{15}, 10^{16} - 1]$
decimal 128	$[-6209, 6111]$	$[10^{33}, 10^{34} - 1]$

Table 3.1: Constraints on variables for radix conversion

this exponent F (3.1) for a new-radix floating-point number only with a multiplication, binary shift, a precomputed constant and a look-up table (see Section 3.2.2). According to Def. 1.2 and Def. 1.6 mantissa computation contains rounding, so the following relation is fulfilled: $10^F \cdot n = 2^E \cdot m \cdot (1 + \varepsilon)$. We are going to consider a value n^* instead, such that $10^F \cdot n^* = 2^E \cdot m$. Thus, we get the following expression for the decimal mantissa:

$$n^* = 2^{E-F} 5^{-F} m \tag{3.2}$$

Multiplication by a power of two 2^{E-F} may be performed with a simple binary shift. Then, as m is small, multiplication by m is easy; therefore the binary-to-decimal mantissa conversion reduces to compute the leading bits of 5^{-F} which is explained in Section 3.2.3.

We explain the algorithm on binary-to-decimal conversion. The same idea applies to decimal-to-binary conversion, however it requires one more normalization step that is explained later. For binary mantissa we get similarly to (3.2):

$$m^* = \frac{10^F \cdot n}{2^E},$$

Thus, for decimal-to-binary conversion computation of the power 5^F is required instead of 5^{-F}. The second step is about computing a power of five 5^B. We are going to consider natural exponents B even while the initial range for exponent might contain negative values. If it is the case, $5^{B+\bar{B}}$ should be computed within our algorithm, where $\bar{B}$ is chosen so that the range for the exponents $B + \bar{B}$ gets nonnegative. We store the leading bits of $5^{-\bar{B}}$ as a constant and after computing $5^{B+\bar{B}}$ with the proposed algorithm, we multiply the result by the constant.

3.2.2 Loop-Less Exponent Determination

The current implementations of the logarithm function are expensive and usually produce approximated values. However, some earlier conversion approaches computed this approximation [37] by Taylor series or using iterations [14, 78]. We explain how to compute the exponent for the both conversion directions exactly neither with libm function call nor any polynomial approximation.

After performing a transformation step based on properties of the logarithm, (3.1) can be rewritten as following:

$$F = \lfloor E \log_{10}(2) + \lfloor \log_{10}(m) \rfloor + \{\log_{10}(m)\} \rfloor - p_{10} + 1, \tag{3.3}$$

where with $\{x\} = x - \lfloor x \rfloor$ we denote the fractional part of the number x. For example, for $x = 3.123$, $\lfloor x \rfloor = 3$, $\{x\} = 0.123$.

As we assumed that the binary mantissa m is normalized in one binade $2^{p_2-1} \leq m < 2^{p_2}$, we can bound it by one decade too.

For example, for binary32 format mantissa m takes values from $[2^{23}, 2^{24} - 1]$. This binade contains 10^7. The "neighbor" powers of ten and our binade are ordered as follows: $2^{23} < 10^7 < 2^{24} < 2^{25} < 10^8$. As we have a power of ten inside the binade, additional scaling is needed: the considered FP number should be $2^{E-1} \cdot 2m$ instead of $2^E \cdot m$. Thus the new mantissa is $2m$ and takes values from the binade $[2^{24}, 2^{25} - 1]$ and therefore is bounded by a decade $[10^7, 10^8 - 1]$. This is an example with "additional scaling" which may be not needed for some other formats. Without loss of generality we stay with the same notations $2^E \cdot m$ knowing that for some formats variables E and m have to be re-assigned.

The inclusion of the mantissa in one decade means that $\lfloor \log_{10}(m) \rfloor$ stays the same for all values of m, we denote it with $\lfloor \log_{10}(m) \rfloor = L$. So, for the given format one can precompute and store this value as a constant. Thus, it is possible to take the integer number $\lfloor \log_{10}(m) \rfloor$ out of the floor operation in the previous equation (3.3). After representing the first summand $E \log_{10}(2)$ as a sum of its integer and fractional parts, we get the following expression for F:

$$F = \lfloor \lfloor E \log_{10}(2) \rfloor + \{E \log_{10}(2)\} + \{\log_{10}(m)\} \rfloor + L - p_{10} + 1.$$

We may take out of the floor another integer number $\lfloor E \log 10(2) \rfloor$, we discuss later how to compute it. Thus, the final formula for the decimal exponent is

$$F = \lfloor \{E \log_{10}(2)\} + \{\log_{10}(m)\} \rfloor + \lfloor E \log_{10}(2) \rfloor + L - p_{10} + 1. \tag{3.4}$$

How to Compute F with Several Additions and a Table

In (3.4) inside the floor operation we have addition of the two fractional parts. Each fractional part $\{\cdot\}$ is in $[0, 1)$ by definition, therefore their sum is inside $[0, 2)$. Roughly speaking, in the expression for F we have $F = \lfloor r \rfloor + \lfloor E \log_{10}(2) \rfloor + L - p_{10} + 1$ where $r \in [0, 2)$. Therefore the result of the floor operation $\lfloor r \rfloor$ may be denoted with γ, where $\gamma \in \{0, 1\}$. So, finally we get the expression for F:

$$F = \gamma + \lfloor E \log_{10}(2) \rfloor + L - p_{10} + 1,\ \gamma \in \{0, 1\}.$$

This correction γ equals to 1 when the sum of two fractional parts from the previous expression exceeds 1 or is equal to 1, or mathematically:

$$r = \{E \log_{10}(2)\} + \{\log_{10}(m)\} \geq 1.$$

This is the same as

$$E \log_{10}(2) - \lfloor E \log_{10}(2) \rfloor + \log_{10}(m) - \lfloor \log_{10}(m) \rfloor \geq 1.$$

As logarithm is an increasing function the left part of this inequality is increasing too. This means that we need only one threshold value $m^*(E)$, such that $\forall m \geq m^*(E)$ the correction $\gamma = 1$. As we know the range for the exponents E beforehand, we can store these critical values in a table:

$$m^*(E) = 10^{1-(E \log_{10} 2 - \lfloor E \log_{10} 2 \rfloor) + \lfloor \log_{10}(m) \rfloor}. \tag{3.5}$$

input : E, m
1 $F \leftarrow E \cdot \lfloor \log_{10}(2) \cdot 2^{\omega} \rfloor$; // multiplication by a constant
2 $F \leftarrow \lfloor F \cdot 2^{-\omega} \rfloor$; // binary right shift
3 $F \leftarrow F + \lfloor \log_{10}(m) \rfloor + 1 - p_{10}$; // addition of a constant
4 **if** $m \geq m^*(E)$ **then**
5 $\quad F \leftarrow F + 1$;
6 **end**

Algorithm 5: The exponent computation in the conversion from binary to decimal

How to Compute $\lfloor E \log_{10}(2) \rfloor$

There is a technique that allows to compute $\lfloor E \log_{10}(2) \rfloor$ for E that takes values from a bounded interval with a multiplication, binary shift and the use of a precomputed constant [10]:

$$\lfloor E \log_{10}(2) \rfloor = \left\lfloor E \left\lfloor \log_{10}(2) \cdot 2^{\omega} \right\rfloor \cdot 2^{-\omega} \right\rfloor$$

for some $\omega \geq \omega^*$

This is not trivial to prove mathematically, but as the range for E is bounded the exhaustive search may be used to check it as well as to find this constant ω^*.

For FP formats ranges for E are always limited. Thus, we may find such suitable ω with brute force. With this transformation we multiply a precomputed constant $\lfloor \log_{10}(2) \cdot 2^{\omega} \rfloor$ by E, and perform binary shift on ω bits.

Finally, putting everything together, the value of the decimal exponent can be obtained as

$$F = \left\lfloor E \left\lfloor \log_{10}(2) \cdot 2^{\omega} \right\rfloor \cdot 2^{-\omega} \right\rfloor + \lfloor \log_{10}(m) \rfloor - p_{10} + 1 + \gamma \tag{3.6}$$

The pseudocode is provided on Algorithm 5.

Adaptation of the Algorithm for Reverse Conversion

Computation of the binary exponent in decimal-to-binary conversion is performed in the same way with the similar reasoning. However,one additional remark has to be clarified. We start with the similar formula

$$E = \lfloor F \log_2(10) + \lfloor \log_2(n) \rfloor + \{\log_2(n)\} \rfloor - p_2 + 1$$

Here we have decimal mantissas n and we consider them bounded by one decade $10^{p_{10}-1} \leq n \leq 10^{p_{10}} - 1$. Even the smallest decade $[1, 10)$ contains three powers of two: 2, 4, 8. As $\lfloor \log_2(10) \rfloor = 3$ it seems that we need three tables, but we may always represent the decimal mantissa n as a binary FP number $n = 2^{\widehat{E}}\widehat{m}$ in some precision $\kappa \geq \lceil \log_2(10^{p_{10}} - 1) \rceil$. Then, for all the possible values $\widehat{m}$ the following holds $\lfloor \log_2(\widehat{m}) \rfloor = \kappa - 1$. This representation can be made exact: we have to shift

$\widehat{m} \leftarrow n$;
$\widehat{E} \leftarrow 0$;
while $\widehat{m} < 2^{63}$ **do**
  $\widehat{m} \leftarrow 2\widehat{m}$;
  $\widehat{E} \leftarrow \widehat{E} - 1$;
end
return $\widehat{m}, \widehat{E}$;

Algorithm 6: Representation of decimal mantissa as a binary FP number

Initial Format	Table size, bytes
binary32	554
binary64	8392
binary128	263024
decimal32	792
decimal64	6294
decimal128	19713

Table 3.2: Table size for exponent computation step

the decimal mantissa n to the left, according to $\widehat{E}$. The further reasoning stays the same. For example, decimal64 numbers can be represented as binary FP numbers $\mathbb{F}_{64}$ from Def. 1.2 with Algorithm 6. So, the unsigned int64 format might be used to store this new mantissa $\widehat{m}$.

A Small Note for Binary-to-Decimal Conversion

The proposed algorithm works for both conversion directions. However, one can notice that for binary-to-decimal conversion the table size can be even reduced by the factor of two. We have used the mantissas from one binade: $2^{p_2-1} \leq m < 2^{p_2}$. However, these mantissas may be scaled to another binade $1 \leq m < 2$. This scaling changes the value of $\lfloor \log_{10} m \rfloor$ in (3.6). Taking mantissas from $[1, 2)$ means that $\lfloor \log_{10} m \rfloor = 0$. This scaling also affects the range for exponents E, but their quantity does not change.

The values for $\lfloor \log_{10} m \rfloor$ stays the same for all m from one decade. Let us see, what happens if we slightly modify the mantissa bounds: $\forall m' : 1 \leq m' < 4$, $\log_{10}(m') = 0$. The new representation of the input is computed out of $2^{E'} m' = 2^E m$. Therefore, we take $E' = E - (E \mod 2)$ and $m' = m \cdot 2^{E \mod 2}$. The value $E \mod 2$ is the last bit in the number E, thus E' is a "shifted version" of E by 1 bit. This means that the range for E' is twice smaller than the range for E. Thus, the table for $m^*(E)$ (3.5) gets twice smaller too.

3.2.3 Computing the Mantissa with the Right Accuracy

Having the exponent computed, the corresponding mantissa must be determined too. We are going to compute it as in (3.2). As we have shown before this task is reduced to the computation of $5^B, B \in \mathbb{N}$. In this Section we propose an algorithm for raising five to a natural power without rational arithmetic or divisions. However, on each computational step we are going to shift to the right λ last bits. The range for these natural exponents B is determined by the input format, e.g. for the conversion from binary64 the range is about 600.

We propose to perform several Euclidean divisions[2] in order to represent the number B in the following way:

$$B = 2^{n_k} \cdot q_k + 2^{n_{k-1}} q_{k-1} + \ldots + 2^{n_1} q_1 + q_0, \tag{3.7}$$

where $0 \leq q_0 \leq 2^{n_1} - 1$, $n_k \geq n_{k-1}$, $k \geq 1$. For example, to convert from binary64 format we used this representation:

$$B = 2^8 q_2 + 2^4 q_1 + q_0.$$

All the quotients q_i are in the same range and we assume that the range for q_0 is the largest one, so we have $q_i \in [0; 2^{n_1} - 1]$, $0 \leq i \leq k$. Once the exponent is represented as in (3.7), computation 5^B is done with the respect to the following expression:

$$5^B = (5^{q_k})^{2^{n_k}} \cdot (5^{q_{k-1}})^{2^{n_{k-1}}} \cdot \ldots \cdot (5^{q_1})^{2^{n_1}} \cdot 5^{q_0}. \tag{3.8}$$

Varying parameters k and n_i the quotients q_i may be made small and the values 5^{q_i} can be stored in a table. These values are normalized so, that they get bounded by one binade: $2^{p-1} \leq 5^{q_i} \leq 2^p - 1$ for some p, for example $p = 64$ for unsigned int64 format used to store these table values. Then, each factor in (3.8) is a table value raised to the power 2^{n_i} which is the same as a table value squared n_i times. So, for our example with binary64 numbers we get

$$5^B = (5^{q_2})^{2^8} \cdot (5^{q_1})^{2^4} \cdot 5^{q_0}.$$

Therefore we get an extended square-and-multiply operation on integers. The value 5^B is huge and we can store only its leading bits. On each multiplication or squaring step, we truncate the last λ bits of the result. So, multiplication of some numbers a, b is executed as $\lfloor a \cdot b \cdot 2^{-\lambda} \rfloor$. There is one more detail to be clarified here. We are going to square values 5^{q_i} that were normalized to fit into $[2^{p-1}, 2^p - 1]$. Therefore after first (pure) multiplication we get

$$2^{2p-2} \leq 5^{q_i} \cdot 5^{q_i} < 2^{2p}.$$

[2]We claimed that the algorithm does not contain divisions. These are done with binary shifts and masks.

input: n_j a power of 2, a value to square $v_j = 5^{q_j}$
1 $\sigma_j \leftarrow 0$;
2 **for** $i \leftarrow 1$ **to** n_j **do**
3 $\quad v_j \leftarrow \lfloor v_j^2 \cdot 2^{-\lambda} \rfloor$;
4 $\quad$ s $\leftarrow 1 - \lfloor v_j \cdot 2^{1-p} \rfloor$ // get the first bit
5 $\quad v_j \leftarrow v_j \ll$ s;
6 $\quad \sigma_j \leftarrow 2 \cdot \sigma_j +$s;
7 **end**
8 result $\leftarrow v_j \cdot 2^{-\sigma_j} \cdot 2^{(2^{n_j}-1)\lambda}$;

Algorithm 7: Squaring with shifting λ last bits

1 $m \leftarrow 1$;
2 **for** $i \leftarrow k$ **to** 1 **do**
3 $\quad m \leftarrow \lfloor (m \cdot v_i) \cdot 2^{-\lambda} \rfloor$;
4 **end**
5 $m \leftarrow \lfloor (m \cdot 5^{q_0}) \cdot 2^{-\lambda} \rfloor$;
6 $m \leftarrow m \cdot 2^{((2^{n_k}-1)+(2^{n_{k-1}}-1)+\cdots+(2^{n_1}-1)+k)\lambda - \sum_{i=k}^{1} \sigma_i}$;
7 $s \leftarrow \sum_{i=k}^{1} (n_i(\lfloor \log_2(5^{q_i}) \rfloor - p + 1)) + \lfloor \log_2(5^{q_0}) \rfloor - p + 1$;
8 result $\leftarrow m \cdot 2^s$;

Algorithm 8: Final multiplication step

This interval is slightly larger than a binade $[2^{2p-1}, 2^{2p})$. When we take $\lambda = p$ it happens that on some squaring steps we may get the loss of precision. Therefore, after truncation of λ last bits on each multiplication step we check the first bit of the obtained number. If it is one, we need to shift the result. This normalization is done with the correction σ_j in Algorithm 7. This algorithm is applied k times to each factor in (3.8). Then the last step is to multiply all the factors starting from the largest power as it is done in Algorithm 8. The line 6 of Algorithm 8 is the compensation of the obtained result. On each step we truncated by λ bits, so we multiply the result by some power of 2^λ. As we normalized the result of each squaring by 2^{σ_j} we multiply the whole result m by $2^{-\sigma_j}$.

The whole algorithm schema is presented on Figure 3.1. Depending on the range of B one can represent it in different manner but for our conversion tasks the ranges for B were not that large, so the numbers n_j were not more than 10 and the loops for squarings can be easily unrolled. For instance, for the conversions from binary32, binary64, decimal32 and decimal64 one can use the expansion of B of the following form:

$$B = 2^8 \cdot q_2 + 2^4 \cdot q_1 + q_0$$

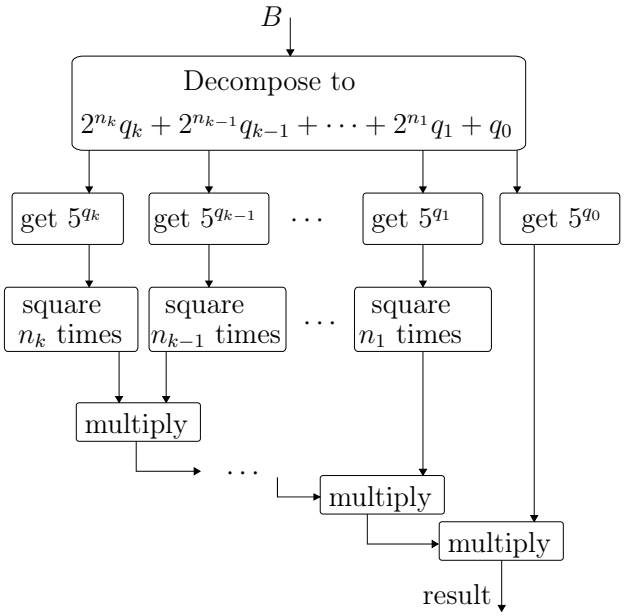


Figure 3.1: Raising 5 to an integer power

3.2.4 Error Analysis for the Powering Step

On the computation of powers of five we store only leading bits. After each multiplication we truncate the result on λ bits. This shifting leading to the truncation is the source of error.

We decomposed the power B into $k+1$ summands and for k of them we execute Algorithm 7 for squaring. With N we denote the total quality of multiplications on the mantissa computation step. For multipliers 5^{q_j} with $1 \leq j \leq k$ we perform n_j squarings. Then, in the end we get $k+1$ multipliers in (3.8), therefore there are k more multiplications. Thus, the total number of multiplications in this algorithm is

$$N = \sum_{j=1}^{k} n_j + k.$$

So, the result is a product of N factors and on each step we have some relative error ε_i. This means that if we define y as the exact product without errors, then what we really compute in our algorithm can be represented as following:

$$\widehat{y} = y \prod_{i=1}^{N} (1 + \varepsilon_i),$$

where all the ε_i are bounded by some $\bar{\varepsilon}$. Thus, the relative error of the computations is

$$\varepsilon = \frac{\widehat{y}}{y} - 1 = \prod_{i=1}^{N} (1 + \varepsilon_i) - 1$$

Let us prove a lemma that will help us to find the bounds for the relative error of the result.

Lemma 3.1. *Let $N \geq 3$, $0 \leq \bar{\varepsilon} < 1$ and $|\varepsilon_i| \leq \bar{\varepsilon}$ for all $i \in [1, N]$. Then the following holds:*

$$\left| \prod_{i=1}^{N} (1 + \varepsilon_i) - 1 \right| \leq (1 + \bar{\varepsilon})^N - 1.$$

Proof. This inequality is equivalent to the following:

$$-(1 + \bar{\varepsilon})^N + 1 \leq \prod_{i=1}^{N} (1 + \varepsilon_i) - 1 \leq (1 + \bar{\varepsilon})^N - 1$$

The proof of the right side is trivial. From the lemma condition we have $-\bar{\varepsilon} \leq \varepsilon_i \leq \bar{\varepsilon}$, which is the same as $1 - \bar{\varepsilon} \leq \varepsilon_i + 1 \leq \bar{\varepsilon} + 1$ for arbitrary i from the interval $[1, N]$. Taking into account the borders for $\bar{\varepsilon}$, we get that $0 < (1 + \varepsilon_i) < 2$ for all $i \in [1, N]$. This means that we can multiply the inequalities $1 + \varepsilon_i \leq \bar{\varepsilon} + 1$ by $1 + \varepsilon_j$ with $j \neq i$. After performing $N - 1$ such multiplications and taking into account that $1 + \varepsilon_i \leq \bar{\varepsilon} + 1$, we get the following:

$$\prod_{i=1}^{N} (\varepsilon_i + 1) \leq (\bar{\varepsilon} + 1)^N.$$

So, the right side is proven.

The same reasoning applies for the left bounds from the lemma condition, and the family of inequalities $1 - \bar{\varepsilon} \leq \varepsilon_i + 1$ leads to the condition:

$$(1 - \bar{\varepsilon})^N - 1 \leq \prod_{i=1}^{N} (1 + \varepsilon_i) - 1.$$

So, in order to prove the lemma we have to prove now that

$$-(1 + \bar{\varepsilon})^N + 1 \leq (1 - \bar{\varepsilon})^N - 1.$$

After regrouping the summands we get the following expression to prove:

$$2 \leq (1 + \bar{\varepsilon})^N + (1 - \bar{\varepsilon})^N.$$

Using the binomial coefficients this trasforms to

$$2 \leq 1 + \sum_{i=1}^{N} \binom{N}{i} \bar{\varepsilon}^i + 1 + \sum_{i=1}^{N} \binom{N}{i} (-\bar{\varepsilon})^i$$

On the right side of this inequality we always have the sum of 2 and some nonnegative terms. So, the lemma is proven. □

The error $\bar{\varepsilon}$ is determined by the basic multiplication algorithm. It takes two input numbers (each of them is bounded between 2^{p-1} and 2^p), multiplies them and cuts off λ last bits, see line 3 of Algorithm 7 and Algorithm 8. Thus, instead of v_j^2 on each step we get $v_j^2 2^{-\lambda} + \delta$, where $-1 < \delta \leq 0$. So, the relative error of the multiplication is bounded by $|\bar{\varepsilon}| \leq 2^{-2p+2+\lambda}$.

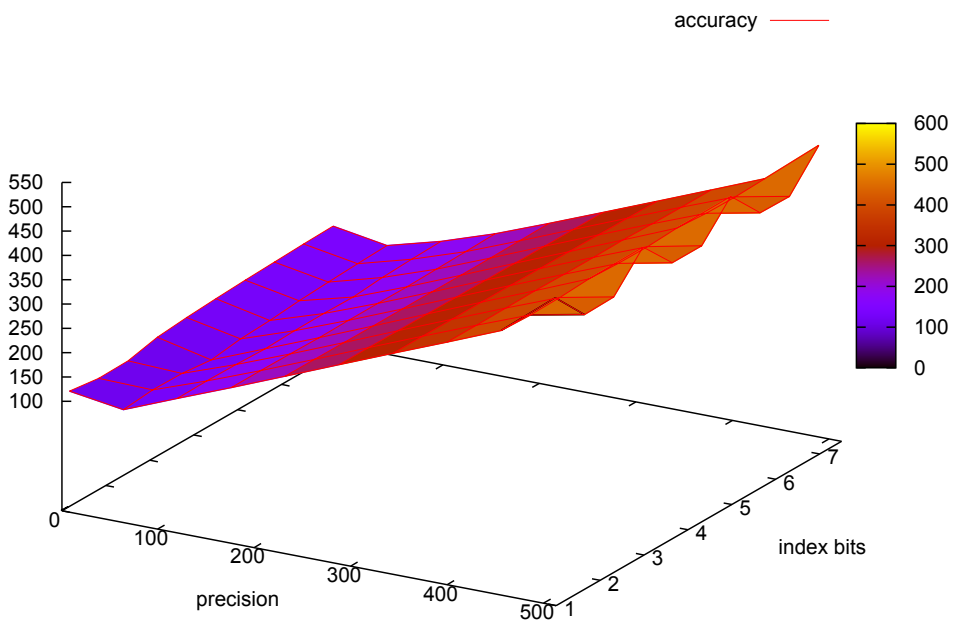


Figure 3.2: Accuracy as a function of precision and table index size

3.2.5 Implementation Details

While the implementation of the first step is relatively simple, we need to specify some parameters and techniques that we used to implement raising 5 to an integer power.

The used computational precision p was equal to 128 bits. The standard C integer types give us either 32 or 64 bits, so for the implementation we used the `uint128_t` type from GCC that is realised with two 64-bit numbers. As a shifting parameter λ we took 64, so getting most or least 64 bits out of `uint128_t` number is easy and fast. Squarings and multiplications can be easily implemented using typecastings and appropriate shifts.

The described conversion approach was used in the implementation of the `scanf` analogue (see Section 3.3) in `libieee754` library [51].

We have implemented an run parametrized algorithm for computation of 5^B, as the parameter we took the table index size (for entries 5^{q_i}) and the working precision p. We see (Fig. Figure 3.2) that the accuracy depends almost linearly on the precision.

3.2.6 Conclusion

A novel algorithm for conversion between binary and decimal floating-point representations has been presented. All the computations are done in integer arithmetic, so no FP flags or modes can be influenced. This means that the corresponding code can be made reentrant. The exponent determination is exact and can be done with several basic arithmetic operations, stored constants and a table. The mantissa computation algorithm uses a small exact table. The error analysis is given and it corresponds to the experimental results. The accuracy of the result depends on the computing precision and the table size. The conversions are often used and the tables are multipurpose, so they can be reused by dozens of algorithms. As this conversion scheme is used everywhere and the tables are not large, they might be integrated in hardware. The implementation of the proposed algorithm can be done without loops, so it reduces the instructions that control the loop, optimizes and therefore accelerates the code.

3.3 Conversion from Decimal Character Sequence

The elegance of a mathematical theorem is directly proportional to the number of independent ideas one can see in the theorem and inversely proportional to the effort it takes to see them

George Pólya[3]

This section is devoted to the conversion from a sequence of decimal characters to a binary FP number. This is an operation similar to a scanf version in C and to its analogues in other languages that read a floating-point number. As we mentioned, the discrete grids of decimal FP numbers and binary ones do not match, that is why this operation generally returns a rounded result. When we are interested in correctly-rounded results, the TMD occurs. The key point here is that the length of the users input may be arbitrarily long, therefore worst cases cannot be precomputed. Thus, our goal was to develop an algorithm that returns correctly-rounded results without divisions or memory allocation; all the rounding modes and floating-point flags have to be maintained correctly. Division is one of the most expensive arithmetic operations, so should be avoided. Memory allocation is also an expensive operation and furthermore may not be available on some architectures (embedded systems). The proposed algorithm is a part of `libieee754-2008` compliance library [51].

3.3.1 Notations and General Idea

We are going to read the decimal string input up to some digit with a finite-state machine. Infinities and NaNs can be also passed as input to our algorithm. So, these textual values have to be filtered out first. After this filtering we assume that the input is some number x. The goal is to get the binary FP number $2^{F_1}n_1$ in precision κ, therefore according to Def. 1.2 we consider its mantissa n_1 bounded as $2^{\kappa-1} \leq n_1 \leq 2^{\kappa} - 1$. We focus on conversion to binary64, so $\kappa = 53$ here. The decimal input is read up to some decimal digit, therefore decimal representation of x is its rounded toward zero version. The scheme of the algorithm is presented on Figure 3.3.

We are going to read two decimal representations of the input x: the first one is "short" and is denoted with $10^E m$, having mantissa in decimal precision r and the second one is a "long" representation $10^{\bar{E}}\bar{m}$ with decimal precision $\bar{r}$. Their mantissas are bounded in these decades $10^{r-1} \leq m \leq 10^r - 1$ and $10^{\bar{r}-1} \leq \bar{m} \leq 10^{\bar{r}} - 1$. The names come from the formats used to store their values: for "short" mantissa one

[3]George Pólya (1887-1985) was a Hungarian mathematician, a professor of mathematics at ETH Zürich and then at Stanford University; also known for science popularization.

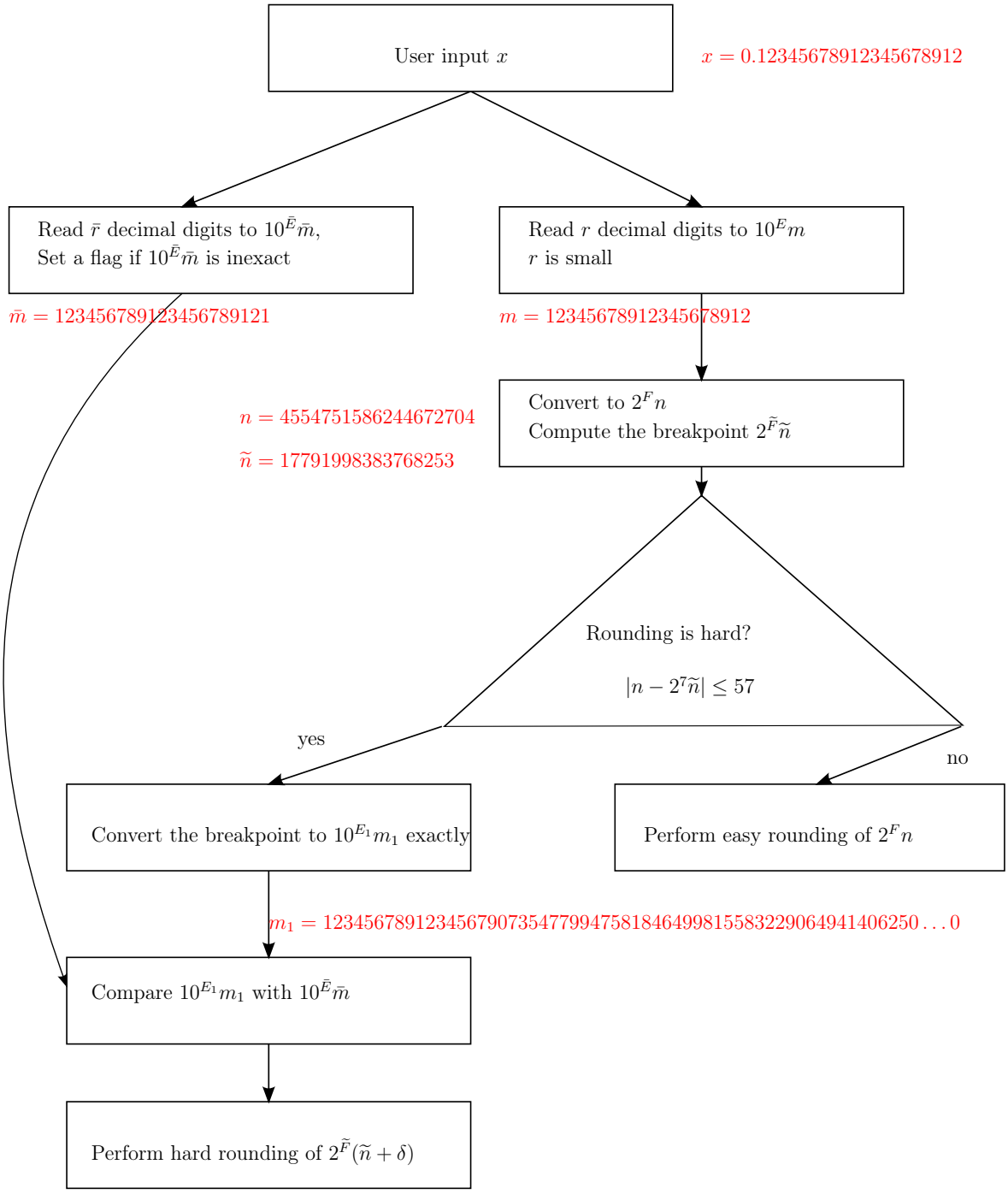


Figure 3.3: Scheme for conversion of decimal input of arbitrary length to binary FP

variable of 64-bit unsigned integer will be enough, for "long" mantissas we will use an array of such variables. Correctly-rounded result is obtained in two ways: the short path for easy rounding and the long path for hard rounding. On the short path only the short decimal representation is used: we convert it to a binary FP number $2^F n$ of higher precision k and compute the binary midpoint $2^{\widetilde{F}}\widetilde{n} \in \mathbb{F}_{54}$. If rounding is hard we use the long path which requires conversion of binary midpoint to a long decimal FP number. This conversion is exact due to the choice of long decimal precision $\bar{r}$ of the new number. Thus, for hard rounding we can compare the binary midpoint correctly represented in decimal with the long version of the input $10^{\bar{E}}\bar{m}$. The result of this comparison gives us the needed rounding direction.

3.3.2 Determine Needed Decimal Precisions

Here we show how to determine the precisions r and $\bar{r}$ for decimal representations of the input. The precision $\bar{r}$ for the large accumulator is determined by the exactness of conversion the binary midpoint to the decimal FP number on long accumulator. The idea of determination of the small decimal precision r is the following. We read the input x into a decimal FP number with r digits, then we transform this decimal number to a binary one in higher-than-double precision k. Therefore, the input x is somehow transformed to binary. We evaluate the relative error of this double transformation knowing the binary precision k.

Determine the Large Decimal Precision $\bar{r}$

We aim to convert midpoints for binary64 numbers to decimal without error, thus we need to determine the corresponding decimal precision. We start from the bounds for the exponent of the midpoint $2^{\widetilde{F}}\widetilde{n}$. We may write the largest and the least FP number in binary64 format according to format specifications given in Table 1.1 and Def. 1.2 of Chapter 1. We suppose that mantissas of the result $2^{F'}n'$ are normalized in one binade $2^{\kappa-1} \leq n' \leq 2^{\kappa}-1$. Therefore, mantissas for midpoints are in the next binade: $2^{\kappa} \leq \widetilde{n} \leq 2^{\kappa+1}-1$. The largest number in binary64 format may be computed as [43]

$$S = 2^{2^{w-1}-\kappa-1} \cdot (2^{\kappa+1}-2),$$

where w is the length of the exponent field. Therefore, we get the upper bound for $\widetilde{F}$ as follows

$$\widetilde{F} \leq 2^{w-1} - \kappa - 1.$$

The smallest FP number is a subnormal one with the minimal exponent $-2^{w-1}+2$ and mantissa $0.\underbrace{0\ldots01}_{\kappa-1}$. Thus, its value may be written as

$$s = 2^{-2^{w-1}-2\kappa+3} \cdot 2^{\kappa}.$$

Then, the midpoint for s is $\frac{1}{2}s$ and therefore, the lower bound for $\widetilde{F}$ may be found as $\widetilde{F} \geq -2^{w-1} - 2\kappa + 2$.

We remind that we use Def. 1.2 to get the FP representation from an arbitrary real number. Thus, transformation of $2^{\widetilde{F}}\widetilde{n}$ to some number $10^{E_1}m_1$ with decimal precision $\bar{r}$ may be formalized as

$$E_1 = \left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor - \bar{r} + 1$$
$$m_1 = \left\lfloor 10^{-\left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor + \bar{r} - 1} 2^{\widetilde{F}}\widetilde{n} \right\rfloor$$

In order to make this conversion exact we should guarantee that the following value is integer:

$$10^{-\left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor + \bar{r} - 1} 2^{\widetilde{F}}\widetilde{n} \in \mathbb{N}.$$

For the previously obtained lower and upper bounds for $\widetilde{F}$ and $\widetilde{n}$ we may find some values of $\bar{r}$, where the previous expression gets only natural values. This means that the decimal precision $\bar{r}$ should be $\bar{r} \geq \bar{r^*}$, where

$$\bar{r^*} = \min\Big\{\bar{r} \,|\, 10^{-\left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor + \bar{r} - 1} 2^{\widetilde{F}}\widetilde{n} \in \mathbb{N}, \widetilde{n} \in [2^{\kappa}, 2^{\kappa+1} - 1], \widetilde{F} \in [-2^{w-1} - 2\kappa + 2, 2^{w-1} - \kappa - 1]\Big\}.$$

By the definition of a FP number (Def. 1.2) $\widetilde{n}, \widetilde{F} \in \mathbb{Z}$. With the range for $\widetilde{F}$ there are two possible situations to be considered: $\widetilde{F} \geq 0$, and $\widetilde{F} < 0$.

1. $\widetilde{F} \geq 0$. Trivially, the number $2^{\widetilde{F}}\widetilde{n} \in \mathbb{N}$. There is a need to guarantee only that

   $$10^{-\left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor + \bar{r} - 1} \in \mathbb{N}.$$

   This is possible, when the power of 10 is positive, therefore when

   $$-\left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor + \bar{r} - 1 \geq 0.$$

   Thus, we get the constraint for $\bar{r}$:

   $$\bar{r} \geq \lfloor \log_{10} 2^{\widetilde{F}}\widetilde{n} \rfloor + 1.$$

2. $\widetilde{F} < 0$. The number $2^{\widetilde{F}}\widetilde{n}$ is not integer. As the mantissa $\widetilde{n} \in \mathbb{N}$ then we require

   $$10^{-\left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor + \bar{r} - 1} 2^{\widetilde{F}} \in \mathbb{N}.$$

   This is equivalent to the requirement for these two numbers $5^{-\left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor + \bar{r} - 1}$ and $2^{-\left\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \right\rfloor + \bar{r} - 1} 2^{\widetilde{F}}$ to be integer. Thus, we get two conditions

   (a) $\lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \rfloor - \bar{r} + 1 \leq 0$. This case gives the same bound for $\bar{r}$ as previously obtained.

(b) $\widetilde{F} - \lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \rfloor - \bar{r} + 1 \geq 0$. Here, for negative exponents $\widetilde{F}$ we get the following:

$$\bar{r} \geq \lfloor \log_{10}(2^{\widetilde{F}}\widetilde{n}) \rfloor + 1 - F, \ (\widetilde{F} < 0). \tag{3.9}$$

Thus, requiring (3.9) we get the two cases verified.

To find the numerical value of $\bar{r}$, we will require stronger inequalities to be satisfied. For example, in order to guarantee that $y \geq \lfloor x \rfloor$ we can require that $y \geq x$ as for all $x \geq 0$, $x \geq \lfloor x \rfloor$. The implication chain on profs is usually built from left to right, but here we try to do in reverse direction.

The value of $\bar{r}$ is non-negative as it is the precision, thus we get

$$\bar{r} \geq \log_{10}(2^{\widetilde{F}}\widetilde{n}) + 1 - F, \ (\widetilde{F} < 0)$$

and it transforms with the use of logarithm properties to

$$\bar{r} \geq \widetilde{F}(\log_{10}(2) - 1) + \log_{10}(\widetilde{n}) + 1, \ (\widetilde{F} < 0).$$

We insert the lower bound for $\widetilde{F}$ and the upper bound for $\widetilde{n}$ in the previous expression to compute the numerical value of $\bar{r}$:

$$\bar{r} \geq (-2^{w-1} - 2\kappa + 2) \cdot (\log_{10}(2) - 1) + \log_{10}(2^{\kappa+1} - 1) + 1.$$

As the expression for the bound of $\bar{r}$ is positive, we use one more transformation with the property $x \leq \lceil x \rceil$. Requirement that $\bar{r} \geq \lceil x \rceil$ guarantees that $\bar{r} \geq x$. Thus, requiring

$$\bar{r} \geq \left\lceil (-2^{w-1} - 2\kappa + 2) \cdot (\log_{10}(2) - 1) + \log_{10}(2^{\kappa+1} - 1) + 1 \right\rceil, \tag{3.10}$$

makes the previous two inequalities satisfied. We get the final expression to compute $\bar{r}$ in (3.10). For binary64 format the corresponding variables are $w = 11$ and $\kappa = 53$ (see Table 1.1), therefore $\bar{r} \geq 806$. So, the long decimal mantissa will have $\bar{r} = 806$ decimal digits, therefore we need 2678 bits to store their values, which is feasible with an array of 42 64-bit integers or array of 84 elements of 32-bit integers.

Determine the small decimal precision r

As mentioned, this precision is determined with evaluation of relative error for binary FP representation of the input number x. So, we arranged to read r decimal digits from x to get its decimal representation $10^E m$. Thus, according to Def. 1.2 we have the following expressions for E and m:

$$E = \lfloor \log_{10}(x) \rfloor - r + 1$$
$$m = \left\lfloor \frac{x}{10^E} \right\rfloor$$

Thus, we may write the following:

$$10^E m = x(1+\varepsilon_1),$$

and the next lemma finds the bound for this relative error.

Lemma 3.2 (About ε_1)**.** *Let be* $x \in \mathbb{R}$, $E \in \mathbb{Z}$, $m \in \mathbb{N}$, $r \in \mathbb{N}$ *such that* $E = \lfloor \log_{10}(x) \rfloor - r + 1$, $m = \left\lfloor \frac{x}{10^E} \right\rfloor$, *then* m *is bounded by one decade*

$$10^{r-1} \leq m \leq 10^r - 1$$

and the following holds

$$10^E \cdot m = x(1+\varepsilon_1), \text{ where } -10^{-r+1} < \varepsilon_1 \leq 0$$

Proof. By the definition $m = \left\lfloor \frac{x}{10^E} \right\rfloor = \left\lfloor x \cdot 10^{-\lfloor \log_{10} x \rfloor + r - 1} \right\rfloor = \left\lfloor 10^{r-1+\delta_{\lfloor\cdot\rfloor}} \right\rfloor$, where $0 \leq \delta_{\lfloor\cdot\rfloor} < 0$. Hence, the bounds for m are $10^{r-1} \leq m < 10^r$. Thus, the first part of the lemma is proven and we have to find the bounds for ε_1 now. From the definition of m and the $\lfloor\cdot\rfloor$ operation we have

$$m = \frac{x}{10^E} - \delta_{\lfloor\cdot\rfloor}, \text{ where } 0 \leq \delta_{\lfloor\cdot\rfloor} < 1.$$

So, we take $\varepsilon' = \frac{\delta_{\lfloor\cdot\rfloor}}{m}$ satisfying $0 \leq \varepsilon' < 10^{1-r}$, then the previous statement gives an expression for x:

$$x = 10^E \cdot m(1+\varepsilon').$$

Which is the same as $10^E \cdot m = x(1+\varepsilon_1)$, where $\varepsilon_1 = \frac{1}{1+\varepsilon'} - 1$. Now we need to find the bounds for ε_1. We use Taylor's development $\frac{1}{1+\varepsilon'}$.

$$\varepsilon_1 = -\varepsilon' + (\varepsilon')^2 - (\varepsilon')^3 + \ldots$$

Then, we get the converging alternating series and its sum is negative and may be bounded by its first summand, so by -10^{1-r}. □

After reading the input, the obtained decimal number $10^E m$ is converted to a binary one $2^{F'} n'$ with mantissa n' on k bits. This conversion from decimal to binary is done with some error ε_2. We may write

$$2^{F'} n' = x(1+\varepsilon_1)(1+\varepsilon_2) = x(1+\varepsilon_1+\varepsilon_2+\varepsilon_1\varepsilon_2). \tag{3.11}$$

For binary FP number $a = 2^{F'} n$ representing the input x we can find the next one binary FP number $a_s = 2^{F'}(n'+1)$. Then, the relative error is computed as

$$\frac{a_s - a}{a} = \frac{1}{n'} \leq 2^{-k+1}.$$

Thus, we can use this bound for relative error $\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2$ in (3.11). We require that

$$|\varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2| \leq 2^{-k+1} \tag{3.12}$$

We want the errors ε_1 and ε_2 be of the same order. We may use the property of absolute value $|a+b| \leq |a| + |b|$. Thus, the following requirement will satisfy (3.12):

$$|\varepsilon_1| + |\varepsilon_2| + |\varepsilon_1\varepsilon_2| \leq 2^{-k+1}.$$

Trivially, $|\varepsilon_1| + |\varepsilon_2| + |\varepsilon_1\varepsilon_2| \geq |\varepsilon_1| + |\varepsilon_2|$. Thus, if we require that $2|\varepsilon_1| \leq 2^{-k+1}$ then (3.12) is satisfied. We may even ask for $|\varepsilon_1| \leq 2^{-k'-g}$, where $g \geq 1$ is some quantity of guard bits. Then,

$$\begin{aligned} 10^{-r+1} &\leq 2^{-k-g} \\ -r+1 &\leq \log_{10}(2^{-k-g}) \\ r &\geq 1 + (k'+g)\log_{10}(2) \end{aligned}$$

If we take $k = 54$ and $g = 5$, then we get $r = 19$ which means that decimal mantissa m may be stored in 64 bits as $\lceil 19 \log_2 10 \rceil = 64$. In this case we get $|\varepsilon_2| \leq 10^{-r+1}$.

3.3.3 Notes on Conversion from Decimal FP Number $10^E m$ to a Binary One $2^F n$

First we convert $10^E m$ to a number $2^{F'} n'$ with mantissa n' and we show further that it is bounded by $2^{60} - 1 \leq n' \leq 2^{61}$. As these bounds do not correspond to one binade as in Def. 1.2 we pass then to a number $2^F n$ with mantissa n on $k = 61$ bits and therefore $2^{60} \leq n \leq 2^{61} - 1$.

The conversion operation was detailed in Section 3.2, we compute the exponent F' as it is explained there, however for the mantissa n' we use a bit different representation that is reused in the further theorems. Thus, the expression for the exponent F' is the following:

$$F' = \left\lfloor \log_2(10^E m) \right\rfloor - k + 1,$$

We consider a mantissa n^* such that

$$2^{F'} n^* = 10^E m.$$

We are going to use several new variables that are defined as follows.

$$\Delta = 4 + \lfloor E \log_2 10 \rfloor - F', \Delta \in \mathbb{Z}$$

$$\varphi(E) = 10^E \cdot 2^{-\lfloor E \log_2 10 \rfloor - 4}$$

With these expressions n^* may be written as

$$n^* = \varphi(E) 2^{\Delta} m.$$

As the mantissa n^* is not computable with pure FP arithmetic we consider other numbers $\bar{n}$ and n' that are defined as follows:

$$\begin{aligned} \bar{n} &= \lfloor \varphi(E) \cdot 2^{\alpha} \rceil \, 2^{\Delta} 2^{-\alpha} m, \\ n' &= \lfloor \bar{n} \rfloor. \end{aligned}$$

These new variables are both some approximations of the input x. Their bounds and relative errors of approximation may be found from the following lemma.

Lemma 3.3 (About the range and error for $\bar{n}, n'$)**.** *Let be* $x \in \mathbb{R}$, $E \in \mathbb{Z}$, $m \in \mathbb{N}$, $r = 19$ *such that*
$E = \lfloor \log_{10}(x) \rfloor - r + 1$, $m = \left\lfloor \frac{x}{10^E} \right\rfloor$,
$n' \in \mathbb{N}, F', \Delta \in \mathbb{Z}, k = 61, \alpha = 64$
$F' = \left\lfloor \log_2(10^E \cdot m) \right\rfloor - k + 1$, $\Delta = 4 + \lfloor E \log_2 10 \rfloor - F'$
$\varphi(E) = 10^E \cdot 2^{-\lfloor E \log_2 10 \rfloor - 4}$, $n^* = 2^{-F'} 10^E m$
$\bar{n} = \lfloor \varphi(E) \cdot 2^\alpha \rceil \, 2^\Delta 2^{-\alpha} m$, $n' = \lfloor \bar{n} \rfloor$
Then, the following holds:

$$\bar{n} = n^*(1 + \bar{\varepsilon}),\ 2^{60} - 1/4 < \bar{n} < 2^{61} + 1,\ |\bar{\varepsilon}| < 2^{-61} \tag{3.13}$$

$$n' = \bar{n}(1 + \varepsilon_n),\ 2^{60} - 1 \le n' \le 2^{61}, |\varepsilon_n| \le \frac{1}{2^{60} - 1/4} \tag{3.14}$$

$$n' = 2^{-F'} x (1 + \varepsilon_2),\ |\varepsilon_2| \le 2^{-58.59} \tag{3.15}$$

Proof. From the definition of n^* we may easily find its bounds.

$$n^* = 10^E 2^{-F'} m = 10^E 2^{-\left\lfloor \log_2(10^E \cdot m) \right\rfloor + k - 1} m = 2^{k - 1 + \delta_{\lfloor \cdot \rfloor}},\ \text{where } 0 \le \delta_{\lfloor \cdot \rfloor} < 1,$$

so $2^{k-1} \le n^* < 2^k$. To find the bounds for $\bar{n}$ we have to develop its expression.

$$\begin{aligned}
\bar{n} &= \lfloor \varphi(E) \cdot 2^\alpha \rceil \, 2^\Delta 2^{-\alpha} m = \varphi(E) 2^\Delta m + \delta_{\lfloor \cdot \rceil} \cdot 2^{\Delta - \alpha} \cdot m \\
&= 10^E 2^{-\lfloor E \log_2 10 \rfloor - 4} \cdot 2^{4 + \lfloor E \log_2 10 \rfloor - F'} \cdot m + \delta_{\lfloor \cdot \rceil} \cdot 2^{\Delta - \alpha} \cdot m \\
&= 10^E 2^{-F'} m + \delta_{\lfloor \cdot \rceil} \cdot 2^{\Delta - \alpha} \cdot m = n^* + \delta_{\lfloor \cdot \rceil} \cdot 2^{\Delta - \alpha} \cdot m = \\
&= n^* + \delta_{\lfloor \cdot \rceil} \cdot 2^{4 + \lfloor E \log_2 10 \rfloor - F' - \alpha} m = n^* + \delta_{\lfloor \cdot \rceil} \cdot 2^{4 + E \log_2 10 + \delta^I_{\lfloor \cdot \rfloor} - \left\lfloor \log_2(10^E \cdot m) \right\rfloor + k - 1 - \alpha} m \\
&= n^* + \delta_{\lfloor \cdot \rceil} \cdot 2^{4 + E \log_2 10 + \delta^I_{\lfloor \cdot \rfloor} - E \log_2 10 - \log_2 m - \delta^{II}_{\lfloor \cdot \rfloor} + k - 1 - \alpha} m \\
&= n^* + \delta_{\lfloor \cdot \rceil} \cdot 2^{3 - \alpha + k + \delta^I_{\lfloor \cdot \rfloor} - \delta^{II}_{\lfloor \cdot \rfloor}},\ \text{with } -1 < -\delta^I_{\lfloor \cdot \rfloor} \le 0,\ -1 < -\delta^{II}_{\lfloor \cdot \rfloor} \le 0, |\delta_{\lfloor \cdot \rceil}| \le 1/2.
\end{aligned}$$

Thus, after the substitution of all the bounds in the expression for $\bar{n}$ we get

$$2^{60} - 1/4 < \bar{n} < 2^{61} + 1.$$

Assuming $\bar{\varepsilon} = \frac{\delta_{\lfloor \cdot \rceil}}{n^*} \cdot 2^{3 - \alpha + k + \delta^I_{\lfloor \cdot \rfloor} - \delta^{II}_{\lfloor \cdot \rfloor}}$ with $-1 < -\delta^I_{\lfloor \cdot \rfloor} \le 0$, $-1 < -\delta^{II}_{\lfloor \cdot \rfloor} \le 0$, $|\delta_{\lfloor \cdot \rceil}| \le 1/2$ we get $|\bar{\varepsilon}| < 2^{-61}$ and

$$\bar{n} = n^*(1 + \bar{\varepsilon}).$$

The number n' is defined as $n' = \lfloor \bar{n} \rfloor$, which means that n' is an integer and as the function $\lfloor \cdot \rfloor$ is an increasing function the bounds for n' are easily determined from these for $\bar{n}$:

$$2^{60} - 1 \le n' \le 2^{61}.$$

By its definition, $n' = \lfloor \bar{n} \rfloor = \bar{n} + \delta_{\lfloor \cdot \rfloor}$ with $-1 < \delta_{\lfloor \cdot \rfloor} \leq 0$, so $\varepsilon_n = \frac{\delta_{\lfloor \cdot \rfloor}}{\bar{n}}$. Thus, $|\varepsilon_n| \leq \frac{1}{2^{60}-1/4}$. The last thing to prove in this lemma is the bound for ε_2. We develop the expression for n' to get the relation between n' and x in one equation.

$$n' = \bar{n}(1+\varepsilon_n) = n^*(1+\varepsilon_n)(1+\bar{\varepsilon}) = 2^{-F'}x(1+\varepsilon_1)(1+\varepsilon_n)(1+\bar{\varepsilon})$$

Thus, we may write $n' = 2^{-F'}x(1+\varepsilon_2)$ with $\varepsilon_2 = (1+\varepsilon_1)(1+\varepsilon_n)(1+\bar{\varepsilon})-1$. So, after substitution of all needed bounds for errors in the last formula, we get the bound for ε_2.

$$|\varepsilon_2| \leq 2^{-58.59}$$

□

The number $2^F n$ with mantissa n on 61 bits is deduced from $2^{F'}n'$ exactly, so $2^F n = 2^{F'}n'$. Thus, there are three conditions:

1. $n' = 2^{61}$. In this case we take $n = n'/2$ and $F = F' + 1$. Division by two is exact as n' is a power of two.
2. $n' = 2^{60} - 1$. We take $n = 2n'$ and $F = F' - 1$.
3. In all other cases we take $n = n'$ and $F = F'$.

Thus, the binary FP number $2^F n$ with mantissa on 61 bits approximates x with the following: $n = 2^{-F}x(1+\varepsilon_2)$ with $|\varepsilon_2| \leq 2^{-58.59}$. This means that the number n has 58 correct bits out of its 61.

The breakpoint mantissa can be computed as

$$\widetilde{n} = \left\lfloor (n + 2^6)2^{-7} \right\rfloor.$$

We compute not only the midpoints for $\mathbb{F}_{53}$ which are rounding bounds for RN, but also the numbers from $\mathbb{F}_{53}$ themselves, which are rounding bounds for the directed rounding modes.

3.3.4 Easy Rounding

The rounding is easy if $2^F n$, the converted version of the input x is far from the midpoint $2^{\widetilde{F}}\widetilde{n}$ (see Chapter 1). Thus, we try to estimate the distance between $2^F n$ the high-precision representation of x and the midpoint $2^{\widetilde{F}}\widetilde{n}$. The most important is to compare the mantissas. For the case of easy rounding, the number $2^F n$ rounds the same as x.

From the results of Lemma 3.3 we can find the absolute error of n approximating $2^{-F}x$. As $n = 2^{-F}x(1+\varepsilon_2)$, then

$$\frac{2^{60}}{1+|\varepsilon_2|} \leq 2^{-F}x \leq \frac{2^{61}-1}{1-|\varepsilon_2|}.$$

Thus, if we take $\delta_n = 2^{-F} \cdot \varepsilon_2$, then $n = 2^{-F}x + \delta_n$ with

$$|\delta_n| \leq \frac{2^{61}-1}{1-|\varepsilon_2|}\,|\varepsilon_2| = \frac{2^{61}-1}{1-2^{-58.59}}2^{-58.59} \approx 5.31.$$

The case of the easy rounding occurs when $|n - 2^7\widetilde{n}| \geq 58$, and the following lemma proves it.

Lemma 3.4 (Easy rounding)**.** *Let $n, \widetilde{n} \in \mathbb{Z}$, $\widetilde{n} = \lfloor (2^6+n)\cdot 2^{-7} \rfloor$, $|n - 2^7\widetilde{n}| \geq 58$, $n \in \mathbb{Z}$, $2^{60} \leq n \leq 2^{61}-1$, $n = 2^{-F}x + \delta$, $|\delta| < 6$ then the following holds:*

$$\left\lfloor 2^{-F-8}x \right\rceil = \left\lfloor 2^{-8}n \right\rceil,$$
$$\left\lceil 2^{-F-8}x \right\rceil = \left\lceil 2^{-8}n \right\rceil,$$
$$\left\lfloor 2^{-F-8}x \right\rfloor = \left\lfloor 2^{-8}n \right\rfloor$$

Proof. We are going to make general judgments from the hypothesis first.

By the definition $\widetilde{n} = \lfloor (2^6+n)\cdot 2^{-7} \rfloor$, which means that $\widetilde{n} = (2^6+n)\cdot 2^{-7} + \delta_{\lfloor\cdot\rfloor}$, with $-1 < \delta_{\lfloor\cdot\rfloor} \leq 0$ or $2^7\widetilde{n} = (2^6+n) + 2^7\delta_{\lfloor\cdot\rfloor}$. From this we get $|2^7\widetilde{n} - n| \leq 2^6 = 64$. From the hypothesis we know that $|n - 2^7\widetilde{n}| \geq 58$, which means that

$$2^7\widetilde{n} - n \in [-64, -58] \cup [58, 64]$$

or after factoring by 2^8,

$$\frac{1}{2}\widetilde{n} - 2^{-8}n \in \left[-\frac{64}{2^8}, -\frac{58}{2^8}\right] \cup \left[\frac{58}{2^8}, \frac{64}{2^8}\right].$$

As the intervals are symmetric, after rephrasing the previous statement we get the expressions for $2^{-8}n$ and $\frac{1}{2}\widetilde{n}$:

$$\frac{1}{2}\widetilde{n} \in 2^{-8}n + \left[-\frac{64}{2^8}, -\frac{58}{2^8}\right] \cup \left[\frac{58}{2^8}, \frac{64}{2^8}\right]; \tag{3.16}$$

$$2^{-8}n \in \frac{1}{2}\widetilde{n} + \left[-\frac{64}{2^8}, -\frac{58}{2^8}\right] \cup \left[\frac{58}{2^8}, \frac{64}{2^8}\right]. \tag{3.17}$$

From the hypothesis we get the following expression for $2^{-8}n$:

$$2^{-8}n = 2^{-F-8}x + 2^{-8}\delta,\ |\delta| < 6.$$

We represent δ values as interval in order to get the interval expression for $2^{-8}n$:

$$2^{-8}n \in 2^{-8-F}x + \left[-\frac{6}{2^8}, \frac{6}{2^8}\right], \tag{3.18}$$

or due to the symmetry of the interval

$$2^{-8-F}x \in 2^{-8}n + \left[-\frac{6}{2^8}, \frac{6}{2^8}\right]. \tag{3.19}$$

Thus, from (3.19) and (3.17) we get

$$2^{-8-F}x \in \frac{1}{2}\widetilde{n} + \left[-\frac{70}{2^8}, -\frac{52}{2^8}\right] \cup \left[\frac{52}{2^8}, \frac{70}{2^8}\right]. \tag{3.20}$$

From the statements (3.20) and (3.17) the theorem may be proven by applying the corresponding rounding operations to the left-hand sides. However, as the both terms contain $\frac{1}{2}\widetilde{n}$, we should consider two cases: when $\widetilde{n}$ is even and when it is odd.

1. Suppose that $\widetilde{n}$ is even. In this case $\frac{1}{2}\widetilde{n} \in \mathbb{Z}$. We start from rounding to the nearest ($\lfloor\cdot\rceil$). The rounding bound for this mode is 1/2, so all the numbers from the interval $\frac{1}{2}\widetilde{n} + \left[-\frac{1}{2}, \frac{1}{2}\right)$ round the same. As $\frac{1}{2} = \frac{128}{256} > \frac{70}{256} > \frac{64}{256}$ the left parts of (3.20) and (3.17) round to the number $\frac{1}{2}\widetilde{n} \in \mathbb{Z}$. Thus, $\lfloor 2^{-8}n \rceil = \left\lfloor \frac{1}{2}\widetilde{n} \right\rceil = \left\lfloor 2^{-8-F}x \right\rceil$, and in this case the theorem is proven for rounding to the nearest mode.

   Consider now rounding to zero ($\lfloor\cdot\rfloor$), which has integer numbers as rounding bounds. We use the proof by contradiction here. So, we suppose that $\lfloor 2^{-8}n \rfloor \neq \lfloor 2^{-8-F}x \rfloor$, which is possible in one of the next two cases:

   (a) $2^{-8-F}x \in \frac{1}{2}\widetilde{n} + \left[-\frac{70}{2^8}, -\frac{52}{2^8}\right]$ and $2^{-8}n \in \frac{1}{2}\widetilde{n} + \left[\frac{58}{2^8}, \frac{64}{2^8}\right]$,

   (b) $2^{-8-F}x \in \frac{1}{2}\widetilde{n} + \left[\frac{52}{2^8}, \frac{70}{2^8}\right]$ and $2^{-8}n \in \frac{1}{2}\widetilde{n} + \left[-\frac{64}{2^8}, -\frac{58}{2^8}\right]$.

   However, from the both cases we get $\left|2^{-8}n - 2^{-8-F}x\right| \leq \frac{110}{2^8}$ which is a contradiction with (3.18). Hence, the assumption was false and $\lfloor 2^{-8}n \rfloor = \lfloor 2^{-8-F}x \rfloor$. The proof for rounding to infinity ($\lceil\cdot\rceil$) is similar as it also has integer numbers as rounding bounds.

2. Suppose that $\widetilde{n}$ is odd, which means that it can be represented as $\widetilde{n} = 2K + 1$, with some integer K. Thus, we may rewrite the statements (3.20) and (3.17) with $\frac{1}{2}\widetilde{n} = K + \frac{1}{2}$:

$$2^{-8}n \in K + \left[\frac{64}{2^8}, \frac{70}{2^8}\right] \cup \left[\frac{186}{2^8}, \frac{192}{2^8}\right], \tag{3.21}$$

$$2^{-8-F}x \in K + \left[\frac{58}{2^8}, \frac{76}{2^8}\right] \cup \left[\frac{180}{2^8}, \frac{198}{2^8}\right]. \tag{3.22}$$

   For round to the nearest mode we are going to use proof by contradiction. Thus, we suppose that $\lfloor 2^{-8}n \rceil \neq \lfloor 2^{-8-F}x \rceil$ which is possible in one of two cases:

   (a) $2^{-8-F}x \in K + \left[\frac{180}{2^8}, \frac{198}{2^8}\right]$ and $2^{-8}n \in K + \left[\frac{64}{2^8}, \frac{70}{2^8}\right]$

   (b) $2^{-8-F}x \in \frac{1}{2}\widetilde{n} + \left[\frac{52}{2^8}, \frac{70}{2^8}\right]$ and $2^{-8}n \in \frac{1}{2}\widetilde{n} + \left[-\frac{64}{2^8}, -\frac{58}{2^8}\right]$.

Once again, from both cases we get $\left|2^{-8}n - 2^{-8-F}x\right| \leq \frac{110}{2^8}$ which is a contradiction with (3.18). Thus, the theorem is proven for rounding to the nearest.

As the rounding bounds for both $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ are the integer numbers and the values from (3.21) and (3.22) are strictly between two integer numbers, the left parts of the mentioned expressions round the same.

Thus, we have proven the theorem for the three rounding modes and for all the subcases. □

Therefore, for easy rounding we get the result, rounding $2^F n$ which is a binary representation of x and this method works for all rounding modes.

3.3.5 How to Implement Easy Rounding

The Lemma 3.4 gives us the result that rounding of the input x is the same as rounding the number $2^F n$. We round the number $2^F n$ with mantissa n on 61 bits to double format, so to 53-bit mantissas. There are several cases to consider: overflow, underflow, normal rounding and subnormal rounding.

How to Produce Over/Underflow

By its definition $n \in [2^{60}, 2^{61}-1]$, therefore the condition for overflow is $F > 1023-61$ and for underflow it is $F < -1074 - 61$. They are found from the expression of the largest and smallest FP number provided in the beginning of this section. Our algorithm used integer computations in order to not affect the rounding modes and FP flags. So, we store a huge and a small exact FP number, e.g. 2^{600} and 2^{-600} and its squaring will set all the needed flags and produce the needed result (NaN or infinity). After this filtering if $F \geq -1022 - 61$ it is a normal rounding.

How to Perform Normal Rounding

The task is to produce a binary64 FP number, so a binary number in $\mathbb{F}_{53}$ from $2^F n$ where n is an integer on 61 bits. We will use memory representation of FP numbers [27], thus we define the following type:

```
typedef union {
uint64_t i;
double f;
} bin64wrapper;
```

Listing 3.1: Wrapper for FP memory representation

Thus, to get bits representation of the FP number x we write it in `bin64wrapper.f` field and read the `bin64wrapper.i` field. To get the FP representation of the number $2^F n$ we start with representing an integer mantissa n in FP and then we will multiply

it by 2^F which is itself a FP number. The way of representing n in FP format is based on Lemma 3.5 and Lemma 3.6 [27].

Lemma 3.5 (Representable integers)**.** *Let $z \in \mathbb{Z}$ be an integer such that $|z| \leq 2^k$. For precision k, $k \geq 2$*

$$z \in \mathbb{F}_k$$

Proof. Consider such cases:

a) $|z| = 0$. Trivial case.

b) $|z| = 2^k$. Trivial case, we have $z = 2^1 \cdot 2^{k-1}$, so according to Def. 1.2 of the floating-point numbers $z \in \mathbb{F}_k$.

c) $1 \leq |z| \leq 2^{k-1} - 1$. We have $z = 2^{-k+1} \cdot (2^{k-1}z)$. If we take $E = -k + 1$ and $m = 2^{k-1} \cdot z$, we get an FP number with mantissa in range $2^{k-1} \leq m \leq 2^{k-1} - 1$.

d) $2^{k-1} \leq |z| \leq 2^k - 1$. The FP number is $z = 2^0 z$, that is $E = 0$, $m = z \in \mathbb{Z}$, $2^{k-1} \leq m \leq 2^k - 1$.

□

Lemma 3.6 (The shift trick)**.** *If y is an integer value in floating-point format $\mathbb{F}_k$ such that $y \leq 2^p - 1 < 2^{k-1}$, the last p significand bits of the floating-point number $z = 2^{k-1} + y$ give us a signed integer number, representing y.*

Proof. The integer number $z = 2^{k-1} + y \in \mathbb{F}_k$ according to Lemma 3.5. Consider first the case when $y \geq 0$. Let us remember how the floating-point numbers in $\mathbb{F}_k$ are stored [42], Figure 1.1. We have 1 bit for the sign; w bits for the exponent with a hidden mantissa first bit and $k - 1$ bits for the mantissa trailing part. We know that $y \leq 2^p - 1$ and $y \in \mathbb{Z}$, so we want to "shift" it in such a way, that it occupies the last p mantissa bits. According to Lemma 3.5, 2^{k-1} is a number in $\mathbb{F}_k$. The mantissa of the value 2^{k-1} in floating-point format is $1.0 \ldots 0$. So, the first 'one' will be the hidden bit stored in exponent field and the mantissa field will be filled with zeros. Thus, if we add to such number an integer value y that is strictly less than 2^{k-1}, y will occupy the least significant bits of mantissa. Thus, the value of z must be more than 2^{k-1}. □

To represent 61-bit integer n in binary64 FP we cut the number n into two parts n_h and n_l so that $n = 2^{32}n_h + n_l$ and then apply the shifting trick adding 2^{52} to n_h and n_l. Listing 3.2 shows in the details how to do this: we use previously described wrapper and memory representation instead of FP numbers to avoid FP operations that may bring the errors and therefore signal the inexact exception and raise the corresponding flag. We use this path in subnormal rounding for the sake of easiness.

```
uint64_t nh, nl;
bin64wrapper nhw, nlw, mantw;
...
nh = n >> 32;
nl = n & 0x00000000ffffffffull;

/* Produce 2^32 * nh and nl as binary64 numbers */
/* 0x4330000000000000ull is a bit representation of FP number 2^52 */
nhw.i = 0x4330000000000000ull | nh;
nhw.f -= 4503599627370496.0; /*2^52*/
nhw.f *= 4294967296.0; /* 2^32 */
nlw.i = 0x4330000000000000ull | nl;
nlw.f -= 4503599627370496.0;

/* Round 2^32 * nh + nl to a binary64 number */
mantw.f = nhw.f + nlw.f;
```

Listing 3.2: Represent 61-bit integer n as a FP number

3.3.6 When Rounding is Hard

When $2^F n$, the binary representation of the input, is close to breakpoint we have the case of hard rounding. We already mentioned, that the binary breakpoint $2^{\widetilde{F}}\widetilde{n}$ will be converted to a decimal FP number $10^{E_1}m_1$ with a long mantissa (see Figure 3.3). This mantissa has $\bar{r}$ decimal digits. The conversion is exact due to the choice of $\bar{r}$. For the hard rounding we may also establish relation between the input x and the midpoint mantissa $\widetilde{n}$ with the following lemma.

Lemma 3.7. *Let be* $n, \widetilde{n} \in \mathbb{Z}$, $\widetilde{n} = \lfloor (n + 2^6)2^{-7} \rfloor$ *and for* $\mu = 57$ *we have* $|n - 2^7\widetilde{n}| \leq \mu$. *Besides that, we use the result from Lemma 3.3:*

$$2^{60} \leq n \leq 2^{61} - 1, n = 2^{-F}x(1 + \varepsilon_2),\ |\varepsilon_2| \leq 2^{-58.59},$$

Then, it may be shown that $\widetilde{n} = \left\lfloor 2^{-7-F}x \right\rceil$.

Proof. We use the absolute error of n representing $2^{-F}x$:

$$n = 2^{-F}x + \delta_n,\ |\delta_n| \leq 5.31$$

From the condition $|n - 2^7\widetilde{n}| \leq \mu$ we get $|2^{-7}n - \widetilde{n}| \leq 2^{-7}\mu$. It means that there is some $\widetilde{\delta}$ such that $|\widetilde{\delta}| \leq 2^{-7}\mu$, that holds $\widetilde{n} = 2^{-7}n + \widetilde{\delta}$. Then,

$$\widetilde{n} = 2^{-7}n + \widetilde{\delta} = 2^{-7}(2^{-F}x + \delta_n) + \widetilde{\delta} = 2^{-F-7}x + 2^{-7}\delta_n + \widetilde{\delta}.$$

Denoting $\widehat{\delta} = 2^{-7}\delta_n + \widetilde{\delta}$ we may show that $\widetilde{n} = 2^{-F-7}x + \widehat{\delta}$ and $|\widehat{\delta}| < 1/2$. Indeed, using the values of δ_n and $\widehat{\delta}$,

$$|\widehat{\delta}| \leq 2^{-7}|\delta_n| + |\widetilde{\delta}| < 2^{-7} \cdot 5.31 + 2^{-7} \cdot 57 < 1/2$$

We get that the difference between $\widetilde{n}$ and $2^{-F-7}x$ is less than one half. By the definition $\widetilde{n} \in \mathbb{Z}$, which means that $\widetilde{n} = 2^{-F-7}x + \widehat{\delta} = \lfloor 2^{-F-7}x + \widehat{\delta} \rceil = \lfloor 2^{-F-7}x \rceil$ by the definition of rounding to the nearest. □

In hard rounding there are also subcases for over/underflow result, normal rounding and subnormal rounding. The first two of them can be filtered out by the value of the exponent $\widetilde{F}$. For normal and especially subnormal rounding some extra actions are needed. The common thing for these both subcases is the exact conversion of binary midpoint to a decimal number. The number $2^{\widetilde{F}}\widetilde{n}$ is converted to $10^{E_1}m_1$ with mantissa m_1 containing $r = 806$ decimal digits. This conversion is performed according to the algorithm explained in Section 3.2, so we do not focus on the details. Once the new decimal number $10^{E_1}m_1$ with long decimal mantissa m_1 is computed it may be compared with $10^{\bar{E}}\bar{m}$, the long decimal representation of the input x. After scaling the number with the least exponent to make $E_1 = \bar{E}$ this comparison may be done lexicographically. Therefore, we have three possibilities:

1. $10^{E_1}m_1 > 10^{\bar{E}}\bar{m}$
2. $10^{E_1}m_1 = 10^{\bar{E}}\bar{m}$
3. $10^{E_1}m_1 < 10^{\bar{E}}\bar{m}$

We take an indicator δ from the result of comparison that will be reused in subcases of normal and subnormal rounding $\delta \in \left\{-\frac{1}{4}, 0, \frac{1}{4}\right\}$. We substitute rounding $\star_\kappa(2^{\widetilde{F}}\widetilde{n})$ by $\star_\kappa(2^{\widetilde{F}}(\widetilde{n} + \delta))$.

Producing Over/Underflow

When $|2^{\widetilde{F}}\widetilde{n}|$ is larger than the largest binary64 floating-point number we get an overflow. Thus, *overflow* exception must be signaled and we need to produce Infinity or the largest binary64 number according to the input sign [43]. In order to do this we execute multiplication of two floating-point numbers for positive input $2^{1000} \times (2^{53} - 1)$ and $(-1)^s \times 2^{1000} \times (2^{53} - 1)$. This case is possible when $\widetilde{F} > 2^{w-1} - \kappa$, or in our binary64 case $\widetilde{F} > 2^{10} - 53 = 971$.

When $|2^{\widetilde{F}}\widetilde{n}|$ is less than the smallest binary64 floating-point number, it is an underflow. As in the previous case we execute multiplication of 2 floating-point numbers, that will certainly give us *underflow* exception with zero value. We use the multiplication of 2^{-1000} and $(-1)^s \times 2^{-1000}$. The conditions on $\widetilde{F}$ to get to this case are $\widetilde{F} < -2^{w-1-\kappa+3-\kappa}$, or for our format $\widetilde{F} < -2^{10} - 106 + 3 = -1127$.

Normal Rounding in the Hard Case

The normal rounding has to be performed when $2^{-2^{w-1}+2} \leq |2^{\widetilde{F}}\widetilde{n}| \leq 2^{2^{w-1}}$, so the midpoint is between the smallest and the largest normal numbers.

We need to perform the rounding $\star_{53}(2^{\widetilde{F}}\widetilde{n})$ for an unknown rounding mode $\star \in \{\circ, \Delta, \nabla, \bowtie\}$. Due to the properties of all the rounding operations we can perform

$$\star_{53}\left(2^{\widetilde{F}}\widetilde{n}\right) = 2^{\widetilde{F}+1} \star_{53} \left(\frac{\widetilde{n}}{2}\right) \tag{3.23}$$

In the interior of the intervals between $\mathbb{F}_{54}$ numbers all user inputs x round the same. So, we can substitute the rounding $\star_{53}(x)$ by $2^{\widetilde{F}+1}\star_{53}(\frac{1}{2}(\widetilde{n}+\delta))$, where $\delta \in \{-\frac{1}{4}, 0, \frac{1}{4}\}$ is an indicator which shows the comparison result from the previous step.

We concentrate now on $\star_{53}(\frac{1}{2}(\widetilde{n}+\delta))$ and introduce a new variable $\nu = \left\lfloor \frac{1}{2}\widetilde{n} \right\rfloor$. Then,

$$\star_{53}\left(\frac{1}{2}(\widetilde{n}+\delta)\right) = \star_{53}\left(\nu + \frac{1}{2}(\widetilde{n}+\delta) - \nu\right) = \star_{53}(\nu+\mu),$$

where $\mu = \frac{1}{2}(\widetilde{n}+\delta) - \nu$. We can deduce all the possible values for μ and it can be computed at the beginning of this step, using the parity of $\widetilde{n}$ and the value of δ.

$$\mu \in \left\{-\frac{1}{8}, 0, \frac{1}{8}, \frac{3}{8}, \frac{1}{2}, \frac{5}{8}\right\} \subset \mathbb{F}_{53}$$

By the definition, $\nu \in \mathbb{N}$, and as $2^{53} \leq \widetilde{n} \leq 2^{54}-1$, we get that $2^{52} \leq \nu \leq 2^{53}-1$, hence $\nu = 2^0\nu \in \mathbb{F}_{53}$. Thus, as $\nu \in \mathbb{F}_{53}$ and $\mu \in \mathbb{F}_{53}$ the rounding $\star_{53}(\nu+\mu)$ can be obtained by executing an addition on the machine. Due to (3.23) we need to multiply this by $2^{\widetilde{F}+1}$. In general, the value $2^{\widetilde{F}+1} \notin \mathbb{F}_{53}$, thus we perform this multiplication in two steps: we take $F_1 = \left\lfloor \frac{\widetilde{F}+1}{2} \right\rfloor$ and $F_2 = \widetilde{F} + 1 - F_1$. Thus, $2^{\widetilde{F}+1} = 2^{F_1}2^{F_2}$. As in the case with ν and μ, $F_1, F_2 \in \mathbb{F}_{53}$ trivially. Perform the final multiplication and rounding as

$$\star_{53}\left(2^{F_1} \cdot \star_{53}(2^{F_2} \cdot \star_{53}(\nu+\mu))\right) \tag{3.24}$$

Subnormal Rounding in the Hard Case

This case occurs when $|2^{\widetilde{F}}\widetilde{n}| < 2^{-2^{w-1}+2}$, i.e. the rounding boundary is less than the smallest normal binary64 number. The subnormal rounding can be divided into 2 cases

1. $2^{\widetilde{F}}\widetilde{n} \leq \frac{1}{4}2^{-2^{w-1}+3-\kappa}$, so the number to be rounded is less than $\frac{1}{4}$ of the smallest subnormal. It is clear that we should set *underflow* flag here and return 0. It can be done by executing

$$\star_{53}(2^{-1000} \cdot 2^{-1000})$$

Let us deduce the bounds for $2^{\widetilde{F}}(\widetilde{n}+\delta)$ for this case. If

$$2^{\widetilde{F}}\widetilde{n} \leq \frac{1}{4}\cdot 2^{-2^{w-1}-\kappa+3}$$

then according to the bounds for $\widetilde{n}$

$$2^{\widetilde{F}} \leq \frac{1}{4}\cdot\frac{2^{-2^{w-1}-\kappa+3}}{2^{54}-1}$$

This leads to

$$2^{\widetilde{F}}(\widetilde{n}+\delta) \leq \frac{1}{4}\cdot 2^{-2^{w-1}-\kappa+3}\left(1+\frac{1}{4}\frac{1}{2^{54}-1}\right)$$

If we demand that $2^{\widetilde{F}}(\widetilde{n}+\delta) \leq \frac{1}{2}\eta$, where $\eta = 2^{-2^{w-1}-\kappa+2}$, then the previous inequation would be also satisfied.

2. $\frac{1}{4}\,2^{-2^{w-1}+3-\kappa} < 2^{\widetilde{F}}\widetilde{n} < 2^{-2^{w-1}+2}$.

We need to perform subnormal rounding and produce a subnormal result in this case. First, we need to make a new definition and prove a theorem.

Definition 3.1. *Let a new operation be* $\langle\cdot\rangle : \mathbb{R}\to\mathbb{Z}$ *as*

$$\langle x\rangle = \begin{cases} x, & if\ x\in\mathbb{Z} \\ \lfloor x\rfloor, & if\ x\notin\mathbb{Z}\ and\ \lfloor x\rfloor\ is\ odd \\ \lceil x\rceil, & otherwise \end{cases} \tag{3.25}$$

Theorem 3.1 (About the operation $\langle\cdot\rangle$)**.** *Let be* $\theta\in\mathbb{Z}:\ 2^{\kappa-1}\leq\theta\leq 2^{\kappa}-1$ *and* $0\leq\rho<1$, $t\in\mathbb{N}$, $t\geq 2$. *Then it can be shown, that*

$$\star_\kappa(\theta+\rho) = \star_\kappa(\theta+2^{-t}\left\langle 2^t\rho\right\rangle) \tag{3.26}$$

Proof. If $2^t\rho\in\mathbb{Z}$, formulation (3.26) is trivial. $\star_\kappa(\theta+\rho)=\star_\kappa(\theta+2^{-t}\left\langle 2^t\rho\right\rangle)$

So, consider the case when $2^t\rho\notin\mathbb{Z}$. For such numbers $\langle 2^t\rho\rangle$ is always odd.

The operation $\langle\cdot\rangle$ is always odd. By definition (3.25), if $\lfloor 2^t\rho\rfloor$ is odd, then $\langle 2^t\rho\rangle = \lfloor 2^t\rho\rfloor$, so $\langle 2^t\rho\rangle$ is odd. Otherwise, if $\lfloor 2^t\rho\rfloor$ is even, then $\langle 2^t\rho\rangle = \lceil 2^t\rho\rceil = \lfloor 2^t\rho\rfloor + 1$, which means that the function value $\langle 2^t\rho\rangle$ is odd. Therefore,

$$\exists\, m\in\mathbb{Z}:\ \left\langle 2^t\rho\right\rangle = 2m+1$$

Let us compute the bounds for $\langle 2^t\rho\rangle$. From the theorem clause we have the bounds for θ and ρ: $2^{k-1}\leq\theta\leq 2^{\kappa}-1$ and $0\leq\rho<1$. Thus, we may get bounds for $\lfloor 2^t\rho\rfloor$ and $\lceil 2^t\rho\rceil$.

$$0\leq 2^t\rho < 2^t,$$

Thus,

$$0 \leq \left\lfloor 2^t \rho \right\rfloor \leq 2^t - 1$$
$$1 \leq \left\lceil 2^t \rho \right\rceil \leq 2^t$$

Therefore we get $0 \leq \langle 2^t \rho \rangle \leq 2^t$. The trivial case when $2^t \rho \in \mathbb{Z}$ was already considered and now we focus only on $2^t \rho \notin \mathbb{Z}$. Thus, $2^t \rho \neq 0$, which means that the lower bound 0 cannot be attained. As we proved, $\langle 2^t \rho \rangle$ is always odd, so the upper bound 2^t also cannot be attained. Finally

$$1 \leq \left\langle 2^t \rho \right\rangle \leq 2^t - 1$$

This result leads to:

$$\begin{aligned} 2^{-t} \leq & 2^{-t} \left\langle 2^t \rho \right\rangle \leq 1 - 2^{-t} \\ \Rightarrow 2^{\kappa-1} + 2^{-t} \leq & \theta + 2^{-t} \left\langle 2^t \rho \right\rangle \leq 2^\kappa - 2^{-t} < 2^\kappa \\ \Rightarrow & \left\lfloor \log_2 \left(\theta + 2^{-t} \left\langle 2^t \rho \right\rangle \right) \right\rfloor = \kappa - 1 \end{aligned} \tag{3.27}$$

Let $[\![\cdot]\!] \in \{\lfloor \cdot \rfloor, \lceil \cdot \rceil, \lfloor \cdot \rceil\}$. We will use it just to generalize the rounding expressions. Thus, according to definition of the roundings Def. 1.2 and (3.27)

$$\begin{aligned} & \star_\kappa \left(\theta + 2^{-t} \left\langle 2^t \rho \right\rangle \right) = \\ & \quad 2^{\left\lfloor \log_2 \left(\theta + 2^{-t} \left\langle 2^t \rho \right\rangle \right) \right\rfloor - \kappa + 1} \cdot \left[\!\!\left[2^{-\left\lfloor \log_2 \left(\theta + 2^{-t} \left\langle 2^t \rho \right\rangle \right) \right\rfloor + \kappa - 1} \cdot \left(\theta + 2^{-t} \left\langle 2^t \rho \right\rangle \right) \right]\!\!\right] = \\ & \qquad [\![\theta + \rho']\!], \text{ where } \rho' = 2^{-t} \left\langle 2^t \rho \right\rangle \text{ and } 0 < \rho' < 1 \end{aligned} \tag{3.28}$$

Thus, we proved that $\star_\kappa(\theta + \rho') = [\![\theta + \rho']\!]$ for $\rho' = 2^{-t} \langle 2^t \rho \rangle$ for the generalized notation of rounding. We are going to show now that $\star_\kappa(\theta + \rho) = [\![\theta + \rho]\!]$ for $0 \leq \rho < 1$, then considering each rounding from $\{\lfloor \cdot \rfloor, \lceil \cdot \rceil, \lfloor \cdot \rceil\}$ we prove that $\star_\kappa(\theta + \rho') = \star_\kappa(\theta + \rho)$ which is the relation we need to prove the theorem.

From the theorem hypothesis $\theta \in \mathbb{Z}$, $2^{\kappa-1} \leq \theta \leq 2^\kappa - 1$ and $0 \leq \rho < 1$, therefore $\lfloor \log_2 (\theta + \rho) \rfloor = \kappa - 1$. Thus,

$$\star_\kappa (\theta + \rho) = [\![\theta + \rho]\!] \tag{3.29}$$

Now with the use of (3.28) and (3.29) we will prove that $\star_\kappa(\theta + \rho) = \star_\kappa(\theta + \rho')$ for $\rho \in [0, 1)$ and $\rho' \in (0, 1)$ and for the three basic roundings from Def. 1.2 RN or $\star = \circ$, RD or $\star = \nabla$, RU or $\star = \Delta$.

1) $\star = \nabla$. According to Def. 1.2 we use here $\lfloor \cdot \rfloor$ instead of $[\![\cdot]\!]$.

$$\begin{aligned} \star_\kappa(\theta + \rho') &= \lfloor \theta + \rho' \rfloor = \theta \\ \star_\kappa(\theta + \rho) &= \lfloor \theta + \rho \rfloor = \theta \\ \Rightarrow \star_\kappa(\theta + \rho') &= \star_\kappa(\theta + \rho) \end{aligned}$$

2) $\star = \Delta$. Analogically with the use of $\lceil\cdot\rceil$,

$$\begin{aligned}\star_\kappa(\theta+\rho') &= \lceil\theta+\rho'\rceil = \theta+1\\ \star_\kappa(\theta+\rho) &= \lceil\theta+\rho\rceil = \theta+1\\ \Rightarrow \star_\kappa(\theta+\rho') &= \star_\kappa(\theta+\rho)\end{aligned}$$

3) $\star = \circ$. This case must be divided into 2 subcases: with $\rho < 1/2$ and with $\rho > 1/2$. If $\rho = 1/2$, then $2^t\rho = 2^{t-1} \in \mathbb{Z}$, but we excluded integer numbers in the beginning of the proof.

a) $0 \le \rho < 1/2$. In this case we have $0 < \rho' < 1/2$. Thus, we can do as earlier,

$$\begin{aligned}\star_\kappa(\theta+\rho') &= \lfloor\theta+\rho'\rceil = \theta\\ \star_\kappa(\theta+\rho) &= \lfloor\theta+\rho\rceil = \theta\\ \Rightarrow \star_\kappa(\theta+\rho') &= \star_\kappa(\theta+\rho)\end{aligned}$$

b) $1/2 < \rho < 1$. In this case we have $1/2 < \rho' < 1$. Thus, similarly to the previous cases,

$$\begin{aligned}\star_\kappa(\theta+\rho') &= \lfloor\theta+\rho'\rceil = \theta+1\\ \star_\kappa(\theta+\rho) &= \lfloor\theta+\rho\rceil = \theta+1\\ \Rightarrow \star_\kappa(\theta+\rho') &= \star_\kappa(\theta+\rho)\end{aligned}$$

Thus, as we proved for each of the 3 possible roundings, the theorem is proven for $\star_\kappa(\theta+\rho') = \star_\kappa(\theta+\rho)$.

The result of this theorem will be used later in our reasoning. □

In order to get the final result we perform rounding $\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta))$. However, we will substitute it by rounding the sum of FP numbers $\xi+\zeta$ such that $2^{52} \le \xi \le 2^{53}-1,\ 0 \le \zeta < 1$. We need to get the bounds for $2^{\widetilde{F}}(\widetilde{n}+\delta)$. We are in the subcase where $\frac{1}{4}\,2^{-2^{w-1}+3-\kappa} < 2^{\widetilde{F}}\widetilde{n} < 2^{-2^{w-1}+2}$. Consider the left part of the inequality in this case condition:

$$\frac{1}{4}\,2^{-2^{w-1}+3-\kappa} < 2^{\widetilde{F}}\widetilde{n}$$

This leads to

$$\frac{1}{4}\,2^{-2^{w-1}+3-\kappa}\,2^{-\widetilde{F}} < \widetilde{n}.$$

After the substitution $\widetilde{n}$ by its upper bound we get

$$\frac{1}{4}\,2^{-2^{w-1}+3-\kappa}\,2^{-\widetilde{F}} < 2^{54}-1.$$

From the last expression we can define bounds for $\widetilde{F}$ in the current case

$$\begin{aligned}\widetilde{F} &\geq \left\lceil -\left(2^{w-1}+\kappa-1+\log_2\left(2^{54}-1\right)\right)\right\rceil \\ &= -\left(2^{w-1}+\kappa-1\right)+\left\lceil -\log_2\left(2^{54}-1\right)\right\rceil \\ &= -2^{w-1}-\kappa+1-53, \quad \text{as } \left\lceil -\log_2\left(2^{54}-1\right)\right\rceil = -53.\end{aligned}$$

Thus, we can obtain the lower bound for $2^{\widetilde{F}}\widetilde{n}$:

$$2^{\widetilde{F}}\widetilde{n} > 2^{-2^{w-1}+1-\kappa-53}\cdot 2^{53}$$
$$2^{\widetilde{F}}\widetilde{n} > \frac{1}{4}\cdot 2^{-2^{w-1}+3-\kappa}\cdot 2^{53}$$

As we operate the integer values of $\widetilde{n}$, we can easily get the inequality like '$\geq$' by increasing $\widetilde{n}$ by 1. Thus,

$$2^{F}\widetilde{n} \geq \frac{1}{4}\cdot 2^{-2^{w-1}+3-\kappa}\cdot(1+2^{53}). \tag{3.30}$$

The same way, to get the upper bound consider the right part of the inequality from the case condition:

$$2^{\widetilde{F}}\widetilde{n} < 2^{-2^{w-1}+2}$$

After dividing the both parts by $2^{\widetilde{F}}$, we get the following:

$$\widetilde{n} < 2^{-\widetilde{F}}\cdot 2^{-2^{w-1}+2}$$

And after substitution the lower boundary for $\widetilde{n}$:

$$2^{-\widetilde{F}} > 2^{53-2^{w-1}+2}$$

Now we can define the upper bound for $\widetilde{F}$ and as earlier we added 1 to lower bound, now we will subtract 1 from the strict upper bound in order to get the inequality like '$\leq$'.

$$\widetilde{F} < -(53+2^{w-1}-2)$$
$$\widetilde{F} < -53-2^{w-1}+2$$
$$\widetilde{F} \leq -53-2^{w-1}+1$$

So, the upper bound for the midpoint $2^{\widetilde{F}}\widetilde{n}$ is

$$2^{\widetilde{F}}\widetilde{n} \leq 2^{-2^{w-1}+2}\left(1-2^{-54}\right) \tag{3.31}$$

As in the case with normal rounding we will compute $\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta))$.

The bounds for $2^{\widetilde{F}}(\widetilde{n}+\delta)$ are now easy to compute:

$$\frac{1}{4}\cdot 2^{-2^{w-1}-\kappa+3}\cdot\left(1+\frac{3}{4}\cdot 2^{-53}\right) \leq 2^{\widetilde{F}}(\widetilde{n}+\delta) \leq 2^{-2^{w-1}+2}\left(1-\frac{3}{4}\cdot 2^{-54}\right)$$

In order to perform the rounding $\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta))$ as summation of FP numbers, we introduce another variable. Let be $q \in \mathbb{Z},\ q = 4\widetilde{n} + 4\delta$. Then,

$$\begin{aligned}\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta)) &= 2^{-2^{w-1}-\kappa+3} \cdot \star_\kappa \left(2^{2^{w-1}+\kappa-3} \cdot 2^{\widetilde{F}}(\widetilde{n}+\delta)\right) \\ &= 2^{-2^{w-1}-\kappa+3} \cdot \star_\kappa \left(2^{2^{w-1}+\kappa-3} 2^{\widetilde{F}-2} q\right)\end{aligned} \tag{3.32}$$

Now, let us focus on what the machine does with rounding $\xi + \zeta$ such that: $2^{52} \leq \xi \leq 2^{53} - 1,\ 0 \leq \zeta < 1$. As we want to consider all the possible roundings we use again the notation $[\![\cdot]\!]$, where $[\![\cdot]\!] \in \{\lfloor\cdot\rfloor, \lceil\cdot\rceil, \lfloor\cdot\rceil\}$. So, after all the declarations we have:

$$\begin{aligned}&\star_\kappa\,(\xi+\zeta) = 2^{\lfloor \log_2|\xi+\zeta| - \kappa + 1\rfloor} \cdot \left[\!\left[2^{-\lfloor \log_2|\xi+\zeta| - \kappa + 1\rfloor} \cdot (\xi+\zeta)\right]\!\right], \\ &\star_\kappa\,(\xi+\zeta) = 2^{52-\kappa+1}\left[\!\left[2^{-52+\kappa-1}(\xi+\zeta)\right]\!\right], \\ &\text{as } \lfloor \log_2|\xi+\zeta| - \kappa + 1\rfloor = \lfloor \log_2|\xi+\zeta|\rfloor - \kappa + 1 = 52 - \kappa + 1\end{aligned}$$

Let ξ and ζ be so, that performs the following:

$$2^{-52+\kappa-1}(\xi+\zeta) = 2^{2^{w-1}+\kappa-3} \cdot 2^{\widetilde{F}-2} \cdot q.$$

Now, after substitution it to (3.32) and taking $2^{-52+\kappa-1}$ out of the rounding operation, we get

$$\begin{aligned}\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta)) &= 2^{-2^{w-1}-\kappa+3} \star_\kappa \left(2^{2^{w-1}+\kappa-3} \cdot 2^{\widetilde{F}}(\widetilde{n}+\delta)\right) \\ &= 2^{-2^{w-1}-\kappa+3} \star_\kappa \left(2^{-52+\kappa-1}(\xi+\zeta)\right) \\ &= 2^{-2^{w-1}-\kappa+3}\, 2^{-52+\kappa-1} \star_\kappa (\xi+\zeta) \\ &= 2^{-2^{w-1}-50} \star_\kappa (\xi+\zeta)\end{aligned}$$

Now, we can finally determine the values of ξ and ζ:

$$\begin{aligned}\xi &= 2^{52}\left\lfloor 2^{-52} \cdot 2^{52-\kappa+1} \cdot 2^{2^{w-1}+\kappa-3} \cdot 2^{\widetilde{F}-2} \cdot q \right\rfloor \\ \zeta &= 2^{52-\kappa+1} \cdot 2^{2^{w-1}+\kappa-3} \cdot 2^{\widetilde{F}-2} \cdot q - \xi\end{aligned}$$

After the simplifications we get

$$\begin{aligned}\xi &= 2^{52}\left\lfloor 2^{-4+\widetilde{F}+2^{w-1}} \cdot q \right\rfloor \\ \zeta &= 2^{48+\widetilde{F}+2^{w-1}} \cdot q - \xi \\ \xi + \zeta &= 2^{48+\widetilde{F}+2^{w-1}} \cdot q.\end{aligned}$$

Let us remember the technique from Lemma 3.6. Let be $0 \leq \tau < 2^{\kappa-1}$, then we have $2^{\kappa-1} \leq 2^{\kappa-1} + \tau < 2^\kappa \ \Rightarrow\ \lfloor \log_2(2^{\kappa-1}+\tau)\rfloor = \kappa - 1$. According to definition of the FP rounding Def. 1.2:

$$\begin{aligned}\star_{53}(2^{\kappa-1}+\tau) &= 2^{\lfloor \log_2(2^{\kappa-1}+\tau)\rfloor} \cdot \left[\!\!\left[2^{-\lfloor \log_2(2^{\kappa-1}+\tau)\rfloor}(2^{\kappa-1}+\tau)\right]\!\!\right] \\ &= \left[\!\left[(2^{\kappa-1}+\tau)\right]\!\right] \\ &= 2^{\kappa-1} + [\![\tau]\!]\end{aligned}$$

So, as in Lemma 3.6, we got the following result

$$\star_{53}(\tau) = \star_{53}(2^{\kappa-1} + \tau) - 2^{\kappa-1}.$$

If we can prove that

$$0 \leq 2^{2^{w-1}+\kappa-3} \cdot 2^{\widetilde{F}-2} \cdot q < 2^{\kappa-1} \tag{3.33}$$

we may compute our rounding as follows:

$$\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta)) = \star_\kappa(2^{\widetilde{F}-2}\cdot q) = 2^{-2^{w-1}-\kappa+3}\cdot(\star_\kappa(2^{\kappa-1}+2^{2^{w-1}+\kappa-3}\cdot 2^{\widetilde{F}-2}\cdot q)-2^{\kappa-1})) \tag{3.34}$$

Let us prove (3.33). First of all we need to normalize q. So, $2^{F'}q' = 2^{\widetilde{F}-2}q$. Then, q' is normalized and $2^{55} \leq q' \leq 2^{56}-1$. Thus, proving (3.33) is the same as proving

$$0 \leq 2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q' < 2^{\kappa-1}.$$

From (3.30) and (3.31) we had the bounds for $\widetilde{F}$:

$$-2^{w-1} - \kappa - 53 + 1 \leq \widetilde{F} \leq -53 - 2^{w-1} + 1$$

As $F' = \widetilde{F} - 2$ we get the following bounds for F':

$$-2^{w-1} - \kappa - 53 - 1 \leq F' \leq -53 - 2^{w-1} - 1 \tag{3.35}$$

Thus, (3.33) is proven with the following reasoning:

$$\begin{aligned}
2^{-2^{w-1}-\kappa-54} &\leq 2^{F'} \leq 2^{-52-2^{w-1}-2} \\
2^{-2^{w-1}-\kappa-54} \cdot 2^{55} &\leq 2^{F'} \cdot q \leq 2^{-52-2^{w-1}-2} \cdot (2^{56}-1) < 2^{-52-2^{w-1}-2} \cdot 2^{56} \\
2^{-2^{w-1}-\kappa+1} &\leq 2^{F'} \cdot q < 2^{2-2^{w-1}} \\
2^{2^{w-1}+\kappa-3} \cdot 2^{-2^{w-1}-\kappa+1} &\leq 2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q < 2^{2^{w-1}+\kappa-3} \cdot 2^{2-2^{w-1}} \\
\frac{1}{4} &\leq 2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q < 2^{\kappa-1}
\end{aligned}$$

We proved even the more strict condition than in (3.33), so (3.33) is also satisfied and we can use now (3.34) with one detail: we change $2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q' = \sigma + \rho$, where

$$\begin{aligned}
\sigma &= \lfloor 2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q' \rfloor \\
\rho &= 2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q' - \sigma
\end{aligned}$$

and let be $\theta = 2^{\kappa-1} + \sigma$. Now, (3.34) takes the following form:

$$\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta)) = 2^{-2^{w-1}-\kappa+3}\cdot(\star_\kappa(2^{\kappa-1}+\sigma+\rho)-2^{\kappa-1}) = 2^{-2^{w-1}-\kappa+3}\cdot(\star_\kappa(\theta+\rho)-2^{\kappa-1})$$

Defined earlier $\theta \in \mathbb{F}_\kappa$, it is easy to prove. It is trivial that $\theta \in \mathbb{Z}$. We need to check if $2^{\kappa-1} \leq \theta \leq 2^\kappa - 1$. It is possible to do with (3.35) and bounds for q'. So,

$$\theta = 2^{\kappa-1} + \lfloor 2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q' \rfloor \geq 2^{\kappa-1}.$$

Additionally:

$$2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q \leq 2^{2^{w-1}+\kappa-3} \cdot 2^{-2^{w-1}-53+1-2} \cdot (2^{56}-1) \leq 2^{\kappa-57} \cdot (2^{56}+1) < 2^{\kappa-1},$$
$$\lfloor 2^{2^{w-1}+\kappa-3} \cdot 2^{F'} \cdot q \rfloor \leq 2^{\kappa-1} - 1$$

The previous inequations give us the upper bound for θ: $\theta \leq 2^\kappa - 1$. Hence we get that $\theta \in \mathbb{F}_\kappa$. Now, finally we can use the theorem 3.1 to compute $\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta))$, substituted ρ by $\rho' = 2^{-\kappa} < 2^\kappa \rho >$.

$$\begin{aligned}\star_\kappa(2^{\widetilde{F}}(\widetilde{n}+\delta)) &= 2^{-2^{w-1}-\kappa+3} \cdot (\star_\kappa(\theta+\rho) - 2^{\kappa-1}) \\ &= 2^{-2^{w-1}-\kappa+3} \cdot (\star_\kappa(\theta+\rho') - 2^{\kappa-1})\end{aligned}$$

Therefore, now all the steps of the algorithm shown on Figure 3.3 are detailed, proven and implemented. This long theory gives the way of getting the final result which is then easy to implement.

3.3.7 Conclusion and Discussion

In this section an algorithm for conversion of decimal string representing a FP number to a binary number was developed. The algorithm performs integer inner computations and is therefore independent of currently set rounding mode. The memory consumption of the algorithm is known beforehand, so there is no memory allocations and it could be used in embedded systems. The result may be produced via short or long path in terms of easy or hard rounding. However, there is a pitfall in choosing the short or long path. When the input number x is a binary FP number itself, the binary version and the breakpoint are the representations of the result. For the moment, our algorithm detects that such an input is too close to the midpoint and performs hard rounding with unnecessary computations. The FP numbers $\mathbb{F}_{53}$ are the subset of midpoints, or numbers from $\mathbb{F}_{54}$. For directed roundings (RU, RD, RZ) the rounding bounds are the FP numbers, for RN rounding mode rounding bounds are the numbers from $\mathbb{F}_{54} \setminus \mathbb{F}_{53}$. Thus, for efficient conversion of string representation of FP numbers the new direct path should be added: after the test whether x was a FP number it is returned as the result. This is left for the future work. Implementation of this direct path would increase the performance of our code and make it comparable with the current version of the scanf operations for FP numbers. This comparison as well as possible optimizations of the algorithm is of great interest for future.

```
double res, theta;
uint64_t q, qprime, sigma, rhoTimes2ToTheG, rhoPrimeTimes2ToK, rest, rhoPrime, temp;
int64_t Fprime, G, Erho;
binary64wrapper wrapper;
...
qprime = 4 * n_tild + 4 * delta;
  Fprime = F - 2;
  qprime = q;
  if (qprime < TWO_TO_55) {
    qprime = qprime << 1;
    Fprime--;
  }
  G = Fprime + 1024 + 50;
  sigma = qprime >> G;
  rhoTimes2ToTheG = qprime & ((1ull << G) - 1);
  rhoPrimeTimes2ToK = rhoTimes2ToTheG >> (G - 53);
  rest = rhoTimes2ToTheG & (1ull << (G - 53) - 1);
  if (rest !=0) {
    rhoPrimeTimes2ToK |= 1;
  }
  wrapper.i = ((1023+52) << 52) | sigma;
  theta = wrapper.f;
  rhoPrime = rhoPrimeTimes2ToK >> 53;
  Erho = -53;
  temp = rhoPrimeTimes2ToK;
  while (temp < (1ull << 52)) {
    temp = temp << 1;
    Erho--;
  }
  wrapper.i = ((Erho + 1024) << 52) | (temp & ((1ull << 52) - 1));
  rhoPrime = wrapper.f;
  if (rhoPrime != 0.0) {
    res = underflowinexact();
  }
  wrapper.f = theta + rhoPrime;
  wrapper.f = wrapper.f - 4503599627370496.0;
  wrapper.i |= (1023 - 1024 - 50) << 52;
  res = wrapper.f;
  return res;
```

Listing 3.3: Subnormal rounding for hard case

3.4 Mixed-radix FMA

An FMA operation means execution of multiplication and addition (or subtraction) within only one rounding:

$$\mathrm{fma}(x, y, z) = \star_k(xy \pm z),$$

$x, y, z \in \mathbb{F}_k, \star_k \in \{\circ_k, \nabla_k, \Delta_k \bowtie_k\}$. Thus, the operation gives a "better" result, than usual multiplication and addition operations, which will produce a double rounding. Several issues with double rounding were discussed in [7,68]. For mixed-radix version of FMA we accept any possible combination of inputs and output radix. As there are two operations within one rounding, implementation of mixed-radix FMA gives already addition (subtraction) and multiplication. It is also a base for division and square root [46, 61, 84]. Thus, implementation of mixed-radix FMA is the base in the research on mixed-radix arithmetic operations.

The IEEE754 Standard requires correctly-rounded implementation of the basic arithmetic operations ($+, -, \times, /, \sqrt{}$, FMA). Getting the correctly-rounded result for a mixed-radix FMA is not easy due to the TMD (reviewed in Chapter 1). Conversion between binary and decimal is rarely errorless: FP numbers of one radix rarely match the values from the discrete set of FP numbers of other radix [62]. The TMD may be solved with precomputing the worst cases, or the cases when the exact results of the operation are so close to the rounding bound that it is not possible to choose the rounded result. The following theory is applied to binary64 and decimal64 formats. We estimate the size of the problem later in this section and as it is tremendous our theory is unfortunately not scalable to 128-bit formats.

This section is not about the implementation of mixed-radix FMA operation, it is a beginning of the research on this topic that contains only worst cases search. We start with a brief overview of the problem in Section 3.4.1 and a short survey on the used technique with continued fractions in Section 3.4.2, then we continue with detailed algorithm and proofs. We finish this section with our algorithm, results and short conclusion on mixed-radix FMA (Section 3.4.9). Actually implementing a mixed-radix FMA is left to the future work.

3.4.1 Brief Overview of the Problem

We start with a mathematical formalization of the mixed-radix FMA operation and the worst cases search. There are three inputs and an output in FMA operation and we consider here any possible combination of input and output radices. Thus, we investigate an expression $d = \star_k(a \cdot b \pm c)$, where a, b, c, d are some binary or decimal FP numbers and $\star_k \in \{\circ_k, \nabla_k, \Delta_k \bowtie_k\}$ is a rounding mode. As we know, floating-point numbers may have different signs, so when performing the worst cases search, we have to pay attention on the signs of inputs, too (the sign of the output

is determined by the signs and absolute values of inputs). FMA operation has 16 variants depending on the sign of operation and on the signs of inputs but we consider first only positive inputs. The impact of all these signs is investigated later in Section 3.4.7.

We are going to use mixed-radix notation, shown in preface to the current Chapter: we represent each number as $2^E5^F m$ with the mantissa m scaled in one binade. Thus our mixed-radix FMA may be written as

$$2^{E_0}5^{F_0}m_0 = \star_k(2^{E_a}5^{F_a}m_a \cdot 2^{E_b}5^{F_b}m_b \pm 2^{E_2}5^{F_2}m_2), \tag{3.36}$$

where k is the precision of the result. We discuss the bounds for all the used parameters later, for the moment we can note them as $m_0, m_a, m_b, m_2 \in [2^{k-1}, 2^k - 1]$, $E_0, E_a, E_b, E_2 \in \mathbf{E}$, and $F_0, F_a, F_b, F_2 \in \mathbf{F}$, where $\mathbf{E}, \mathbf{F}$ are intervals of integers.

The key point of the FMA operation is to perform multiplication and addition in one rounding. The first operation (multiplication) may be performed exactly with the use of a higher-precision mixed-radix number: $2^{E_1}5^{F_1}m_1 = 2^{E_a}5^{F_a}m_a \cdot 2^{E_b}5^{F_b}m_b$, for example $2^{2k-1} \le m_1 \le 2^{2k}$. And then, after replacement of the multiplication our FMA operation is reduced to

$$2^{E_0}5^{F_0}m_0 = \star_k(2^{E_1}5^{F_1}m_1 \pm 2^{E_2}5^{F_2}m_2). \tag{3.37}$$

with the constraint $2^{E_1}5^{F_1}m_1 \ge 2^{E_2}5^{F_2}m_2$, which might require summands swap.

As the result of addition or subtraction may be not representable in the results format due to the hidden radix conversion, we write the following, which is the essential question of TMD:

$$2^{E_1}5^{F_1}m_1 \pm 2^{E_2}5^{F_2}m_2 \approx 2^{E_0}5^{F_0}m_0.$$

This transforms into the following fraction after division by $m_0, 2^{E_1}$ and 5^{F_1}:

$$\frac{m_1 \pm 2^{E_2-E_1}5^{F_2-F_1}m_2}{m_0} \approx \frac{2^{E_0-E_1}}{5^{F_1-F_0}}.$$

To make this formula look more compact, we introduce new variables: $T = E_2 - E_1$, $S = F_2 - F_1$, $B = E_0 - E_1$, $A = F_1 - F_0$. Then our FMA is transformed to

$$\frac{m_1 \pm 2^T5^S m_2}{m_0} \approx \frac{2^B}{5^A}. \tag{3.38}$$

To find the worst cases of an operation or a function we have to find the smallest nonzero distance between the function value and the FP midpoint [9]:

$$\min_{m_0,m_1,m_2,A,B,S,T} \left| \frac{m_1 \pm 2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right|. \tag{3.39}$$

All the parameters for minimum search are discrete and this minimum may be found brute-force. However, the ranges are large (see Section 3.4.4 for numerical values)

which means that this search is hard in combinatoric sense: there are too many variants to be considered. Considering the expression inside minimum in details, we may notice that it is a rational approximation of a rational number. Moreover, as we are interested in minimum, this is the so-called best rational approximation: it is closer to the considered number than other approximations. Continued fractions are used to find the best rational approximation for the given number. We will consider the fraction $2^B/5^A$ as a given rational number and with continued fractions we are going to find its best approximation. The advantage here, is that we get rid of brute-force iteration over the ranges for m_0, m_1, m_2. However, the application of continued fractions does not take into account this specific form of the numerator, so there is an extra step needed. All this is discussed later.

So, for the moment for all positive inputs and output there are two subtasks for this minimum search: the case with "+" sign and with "−" sign in numerator $m_1 \pm 2^T 5^S m_2$ that we call later *addition case* and *subtraction case*. The algorithm of the best rational approximation looks for fractions with positive bounded numerator and denominator, therefore it might be applied to the addition case straightforward.

For the addition case we aim to find

$$\min_{m_0,m_1,m_2,A,B,S,T} \left| \frac{m_1 + 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A} \right|. \tag{3.40}$$

Then, trivially for the subtraction case it is

$$\min_{m_0,m_1,m_2,A,B,S,T} \left| \frac{m_1 - 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A} \right|. \tag{3.41}$$

The next subsection contains a short survey on the application of continued fractions for such rational approximations. Then in Section 3.4.7 we provide discussion on considering the signs of the inputs and the subtraction case. Without loss of generality we add a condition

$$m_1 \geq \frac{1}{2} 2^T 5^S m_2. \tag{3.42}$$

This avoids cancellations in numerator for subtraction case.

3.4.2 Short Survey on Continued Fractions

This section contains several definitions on continued fractions and an overview of the algorithm used further for best rational approximation. More details on this topic may be found in Khinchin's book [50] and in paper of Cornea et al. [22].

Definition 3.2 (Continued fraction)**.** *An expression of the form*

$$a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \ldots}}$$

is called a continued fraction.

Definition 3.3 (Convergent)**.** *Every finite continued fraction with numerical elements $a_1, a_2, \ldots, a_n$ is represented as an ordinary fraction p/q, which is called a convergent.*

Definition 3.4 (Mediant)**.** *A mediant of two fractions a/b and c/d is called a fraction*

$$\frac{a+c}{b+d}.$$

Mediants of convergents are convergents too.

One of the important properties of the continued fractions is that they may represent each real number. For rational numbers these fractions are finite, for irrational infinite. The most important application of continued fractions is representation of numbers with some predefined accuracy (see theorems 9 and 13 in [50]). Continued fractions are used to compute the best approximation of some number x and it is proven in [50] that each best approximation of x is its convergent.

Definition 3.5 (Best approximation)**.** *A rational fraction a/b is called a best approximation of a real number x if every other rational fraction c/d with a denominator not larger than b differs from x by larger amount. In other words, $0 < d \leq b$, $\frac{a}{b} \neq \frac{c}{d}$ implies*

$$\left|x - \frac{c}{d}\right| > \left|x - \frac{a}{b}\right|.$$

So, we try to find a fraction a/b that minimizes the value $\left|x - \frac{a}{b}\right|$. The standard approach from [50] assumes x to be real, the algorithm from [22] is applied only to rational numbers x. It considers positive numerators a and denominators b upper-bounded by some values. Besides that, the modified version looks for the fractions different from x. We try to find an approximation for x on the left and on the right. For each of them the two convergents are found on each step a_1/b_1 and a_2/b_2. Then, the closest one is chosen from the best left and the best right approximation. Classical algorithm makes one step at a time, while modification form the paper of Cornea et al. skips several steps. Let us detail the classical algorithm first. The convergents are initialized as follows:

1. for *left* approximation: $a_1/b_1 = \lceil x \rceil - 1$, $a_2/b_2 = \lceil x \rceil$. So, left approximations are always smaller than x.

2. for *right* approximation: $a_1/b_1 = \lfloor x \rfloor$, $a_2/b_2 = \lfloor x \rfloor + 1$. Right approximations are always larger than x.

On the initialization step it is clear that the best left approximation is a_2/b_2 and the best right is a_1/b_1. Then the mediants are computed iteratively until $b_1 + b_2 < N$. As soon as $b_1 + b_2 > N$ is reached, a_1/b_1 is taken as the best left approximation, and a_2/b_2 as the best right approximation of x. So, when $b_1 + b_2 < N$ on each step we compute mediants $a/b = \frac{a_1+a_2}{b_1+b_2}$ and the new pairs of convergents are chosen then.

For *left* approximation: if mediant $a/b < x$ the next pair of convergents is $a/b, a_2/b_2$. If $x \leq a/b$ then the next pair of convergents is $a_1/b_1, a/b$.

For *right* approximation: if mediant $a/b \leq x$, we consider a/b and a_2/b_2. If $x < a/b$ we choose $a_1/b_1, a/b$.

Cornea et al. noticed [22] that this classical approach means computing a tremendous number of mediants, therefore they proposed to skip several steps computing other convergents instead of mediants. They compute some integer numbers k_{left} and k_{right} and depending on their values the following convergents are computed instead of mediants: $a = a_1 k_{left} + a_2$ and $b = b_1 k_{left} + b_2$ or $a = a_1 + a_2 k_{right}$ and $b = b_1 + b_2 k_{right}$.

This means that the best rational approximation may be useful to estimate or bound the minimum from (3.40): for each combination of parameters A and B we find a fraction n/m that approximates the number $2^B/5^A$. Best approximation search does not take into account the specific form of numerator in (3.40), thus some additional transformations are required.

3.4.3 General Idea for the Algorithm

We start the essential part with general explanation of the algorithm. For the moment we consider only the addition case from (3.40), supposing that the inputs were all positive. We give all the details for this case. The support of negative inputs or any other combination of signs will be discussed later.

We remind the problem statement once again, putting all the conditions together. Let be $A, B, S, T \in \mathbb{Z}$; $m_0, m_1, m_2 \in \mathbb{Z}$,

$$A \in \mathbf{A}, \quad B \in \mathbf{B}, \quad S \in \mathbf{S}, \quad T \in \mathbf{T} \tag{3.43}$$

Numerical ranges for all these and other values are discussed in Section 3.4.4.

$$\begin{aligned} 2^{k-1} &\leq m_1 \leq 2^k - 1 \\ 2^{k'-1} &\leq m_2 \leq 2^{k'} - 1 \\ 2^{k''-1} &\leq m_0 \leq 2^{k''} - 1 \end{aligned} \tag{3.44}$$

with the assumption that $m_1 \geq \frac{1}{2} 2^T 5^S m_2$ and that $\frac{m_1 + 2^T 5^S m_2}{m_0} \neq \frac{2^B}{5^A}$ we are looking for

$$\min_{m_0, m_1, m_2, A, B, S, T} \left| \frac{m_1 + 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A} \right|. \tag{3.45}$$

The previous section explained how to find the best rational approximation for number $x \in \mathbb{R}$, which means to find a fraction a/b such that the value $\left|\frac{a}{b} - x\right|$ is minimal. For the moment we assume that for each combination of parameters A and B we find the best fraction a/b that approximates the number $2^B/5^A$.

Algorithm to find this best rational approximation takes upper bounds for positive integer numerator a and denominator b, thus we have to find these bounds first.

The numerator a in our case is represented as $m_1+2^T5^Sm_2$. As it has to stay integer, depending on the signs of S and T there are four ways to represent it and therefore to transform the task.

We are going to iterate the ranges for A, B, S, T and to search for the best rational approximation on each iteration. Then we find the global minimum after all the loops. Thus, the scheme of the algorithm is simple: four nested loops for A, B, S, T and the best rational approximation algorithm in the innermost one. The order of the loops does not matter for the moment but will be fixed later. Let us consider in detail the ways to represent numerator a.

1. $T \geq 0, S \geq 0$. The powers are non-negative, therefore no divisions are needed and the numerator $a = m_1 + 2^T5^Sm_2$ is integer. Its bounds are determined from (3.44), therefore

$$2^{k-1} + 2^T5^S2^{k'-1} \leq a \leq 2^k - 1 + 2^T5^S(2^{k'} - 1).$$

   Denominator b is the same as in the task (3.40) $b = m_0$ in this and all other cases. Thus, we are going to search for a fraction a/m_0 that minimizes the following expression:

$$\min \left| \frac{a}{m_0} - \frac{2^B}{5^A} \right|.$$

2. $T \geq 0, S < 0$. The number 5^S is not integer, therefore we cannot take numerator a as in previous case. Therefore, we represent it as $a = 5^{-S}m_1 + 2^Tm_2$. Then, according to (3.44), it is bounded with

$$5^{-S}2^{k-1} + 2^T2^{k'-1} \leq a \leq 5^{-S}(2^k - 1) + 2^T(2^{k'} - 1).$$

   As we factorized fraction by 5^S, we should do the same for the known number $2^B/5^A$. Therefore, the sought-for minimum transforms into

$$5^S \min \left| \frac{a}{m_0} - \frac{2^B}{5^{A+S}} \right|.$$

3. $T < 0, S \geq 0$. We avoid division by 2^T in order to get an integer number a, therefore we factorize by 2^T and the considered numerator is $a = 2^{-T}m_1+5^Sm_2$. It is bounded by

$$2^{-T}2^{k-1} + 5^S2^{k'-1} \leq a \leq 2^{-T}(2^k - 1) + 5^S(2^{k'} - 1).$$

   Similarly to the previous case, factorization of the fraction leads to

$$2^T \min \left| \frac{a}{m_0} - \frac{2^{B-T}}{5^A} \right|.$$

4. $T < 0, S < 0$. As both parameters are negative we factorize the fraction as well as the whole expression by $2^T 5^S$. Numerators a to be considered for the best rational approximation take the form of $a = 2^{-T}5^{-S}m_1 + m_2$ and take values from

$$2^{-T}5^{-S}2^{k-1} + 2^{k'-1} \leq a \leq 2^{-T}5^{-S}(2^k - 1) + 2^{k'} - 1.$$

Our task is therefore in searching for

$$2^T 5^S \min \left| \frac{a}{m_0} - \frac{2^{B-T}}{5^{A+S}} \right|.$$

In the continued fraction theory there is no constraint on special form of numerator or denominator. Thus, we do not take into account the special form of a. So, we can denote with a^* the value that the algorithm gives us as the best approximation. Then we may use a representation $a^* = 2^{T_1}5^{S_1}m_1 + 2^{T_2}5^{S_2}m_2 \pm r$, where T_1 and T_2

```
1  Procedure leftExpansion(a, S, T):
2  if T ≥ 0 then
3      if S ≥ 0 then
4          α ← 2^T 5^S ;
5          β ← 1;
6      else
7          α ← max{5^{-S}, 2^T} ;
8          β ← min{5^{-S}, 2^T} ;
9      end
10 else
11     if S ≥ 0 then
12         α ← max{2^{-T}, 5^S} ;
13         β ← min{2^{-T}, 5^S} ;
14     else
15         α ← 2^{-T} 5^{-S} ;
16         β ← 1 ;
17     end
18 end
19 a_1 ← ⌊a/α⌋ ;
20 r_1 ← a − a_1 α ;
21 a_2 ← ⌊r_1/β⌋; // a_2 ← ⌈r_1/β⌉ in expansion to the right
22 r ← r_1 − a_2 β ;
23 return a_1 α + a_2 β;
```

Algorithm 9: Expansion of a to the left

cannot be both zeros (the same applies for S_1 and S_2) and $r > 0$. Then, assuming that $a^* = a \pm r$, we may compute the needed minima. This transformation may be done within Algorithm 9. We get then the two new approximations of a^*. Similarly to the continued fraction theory, we call the one that is less then a^* its left expansion and the other right. The difference in sign for $a^* = a \pm r$ influences only one line (line 20) in the algorithm. Therefore, after the best rational approximation, we perform the expansion of the numerator to the left and to the right. Thereby, we take into account the specific form of the numerator. Having two new fractions, we can easily compute the minima and choose the best one.

On Algorithm 10 we illustrate the described four cases for rational approximation. Depending on the signs of the exponents T, S, we approximate different values. This algorithm will be used later in the innermost loop.

3.4.4 Estimation of the Parameter Ranges

To estimate the quantity of iterations in our minimum search the bounds for all the parameters (3.43)-(3.44) have to be determined. Mantissa m_1 was the exact of a multiplication of two FMA parameters. According to [43] binary64 mantissas may be normalized so that they fit into $[2^{52}, 2^{53} - 1]$, decimal64 mantissas may be scaled to 54-bit integers. Thus, we may scale a bit the range for mantissas of the two formats, so that mantissas of both formats are representable. Thus, we represent the mantissas of the inputs m_a and m_b as 55-bit integers (3.36). Therefore, the result of their multiplication, m_1 is on 110 bits. To include the guard bits [38], we suppose that the mantissas of the result and another input are 60-bit integers. Therefore, all the unknowns in (3.44) are now determined:

$$k = 110, \quad k' = 60, \quad k'' = 60 \tag{3.46}$$

The choice of 60-bit integers may be criticized here as a waste; however as we use the algorithm of best rational approximation that skips several convergents at a time, we assume that this is not a remarkable overhead.

The bounds for A, B, S, T are determined according to [43] and scaling of the mantissas done previously. We consider slightly enlarged intervals so that the corresponding numbers occupy a certain quantity of bits. Thus,

$$\begin{aligned} \mathbf{A} &= [-2^{11} + 1; 2^{11} - 1] \\ \mathbf{B} &= [-2^{12} + 1; 2^{12} - 1] \\ \mathbf{S} &= [-2^{11} + 1; 2^{11} - 1] \\ \mathbf{T} &= [-2^{12} + 1; 2^{12} - 1] \end{aligned} \tag{3.47}$$

As mentioned, we are searching for best rational approximation of $2^B/5^A$ in four nested loops, so for all the combinations of the parameters A, B, S, T. This means

Procedure `bestRationalApproximation`$(a_{\min}, a_{\max}, S, T, A, B, k, k', k'')$:
Input : $a_{\min}, a_{\max}, S, T, A, B$, precisions: k for m_1, k' for m_2, k'' for m_0
Output: current minimum m_c
if $T \geq 0$ **then**
  if $S \geq 0$ **then**
    $a_{\min} \leftarrow 2^{k-1} + 2^T 5^S 2^{k'-1}$;
    $a_{\max} \leftarrow 2^k - 1 + 2^T 5^S (2^{k'} - 1)$;
    $\kappa \leftarrow 1$;
    $\alpha \leftarrow \frac{2^B}{5^A}$;
  else
    $a_{\min} \leftarrow 5^{-S} 2^{k-1} + 2^T 2^{k'-1}$;
    $a_{\max} \leftarrow 5^{-S}(2^k - 1) + 2^T (2^{k'} - 1)$;
    $\kappa \leftarrow 5^S$;
    $\alpha \leftarrow \frac{2^B}{5^{A+S}}$;
  end
else
  if $S \geq 0$ **then**
    $a_{\min} \leftarrow 2^{-T} 2^{k-1} + 5^S 2^{k'-1}$;
    $a_{\max} \leftarrow \leq 2^{-T}(2^k - 1) + 5^S (2^{k'} - 1)$;
    $\kappa \leftarrow 2^T$;
    $\alpha \leftarrow \frac{2^{B-T}}{5^A}$;
  else
    $a_{\min} \leftarrow 2^{-T} 5^{-S} 2^{k-1} + 2^{k'-1}$;
    $a_{\max} \leftarrow 2^{-T} 5^{-S}(2^k - 1) + 2^{k'} - 1$;
    $\kappa \leftarrow 2^T 5^S$;
    $\alpha \leftarrow \frac{2^{B-T}}{5^{A+S}}$;
  end
end
$\frac{a_{left}}{b_{left}} \leftarrow$ `bestLeftApprox(`$expr, a_{\min}, a_{\max}, 2^{k''-1}, 2^{k''} - 1$`)`;
$\frac{a_{right}}{b_{right}} \leftarrow$ `bestRightApprox(`$expr, a_{\min}, a_{\max}, 2^{k''-1}, 2^{k''} - 1$`)`;
$a_{left} \leftarrow$ `leftExpansion(`a_{left}, S, T`)`;
$a_{right} \leftarrow$ `rightExpansion(`a_{right}, S, T`)`;
$m_c \leftarrow \kappa \cdot \min\left\{\left|\frac{a_{left}}{b_{left}} - \alpha\right|, \left|\frac{a_{right}}{b_{right}} - \alpha\right|\right\}$;
return m_c;

Algorithm 10: The algorithm to compute the appropriate rational approximation

that the total number of iterations is about $2^{12+12+13+13} > 10^{15}$ which is extremely large. In Section 3.4.1 it was shown how to get each of these parameters. Having some more constraints on the task we can establish certain connection between A, B, S, T

that allow to reduce their ranges.

3.4.5 How to Reduce the Ranges for the Exponents A, B, S, T

The task is to find the minimum of the following expression:

$$\left|\frac{m_1 + 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A}\right|.$$

We may formally divide the range of its values with 1/2: the task will be to find the two minima of the same expression. In the first case this expression takes values less than 1/2 and in the second one it takes values larger than 1/2. Thus, the first one is

$$\min\left|\frac{m_1 + 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A}\right|, \text{ when } \left|\frac{m_1 + 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A}\right| \leq \frac{1}{2}.$$

And the second one is

$$\min\left|\frac{m_1 + 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A}\right|, \text{ when } \left|\frac{m_1 + 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A}\right| > \frac{1}{2}.$$

It is clear that the global result will be among the results of the first task. Therefore, we add a following constraint to our task of FMA

$$\left|\frac{m_1 + 2^T 5^S m_2}{m_0} - \frac{2^B}{5^A}\right| \leq \frac{1}{2} \tag{3.48}$$

From (3.48) we may find new bounds for the parameter B which do not have to exceed the initial bounds of **B** from (3.47). We remind another useful constraint here:

$$m_1 \geq \frac{1}{2} 2^T 5^S m_2. \tag{3.49}$$

From the condition (3.49) we get

$$m_1 + 2^T 5^S m_2 \leq 3m_1.$$

And trivially $m_1 + 2^T 5^S m_2 \geq m_1$ as we add a positive value to m_1. Thus, the fraction $\frac{m_1}{m_0}$ is bounded by

$$\left[\frac{2^{k-1}}{2^{k''} - 1}; 3\frac{2^k - 1}{2^{k''-1}}\right].$$

Therefore from (3.48) we get

$$\frac{m_1}{m_0} - \frac{1}{2} \leq \frac{2^B}{5^A} \leq \frac{1}{2} + 3\frac{m_1}{m_0}$$

Which transforms to the following after a multiplication by 5^A:

$$5^A\left(-\frac{1}{2} + \frac{m_1}{m_0}\right) \leq 2^B \leq 5^A\left(\frac{1}{2} + 3\frac{m_1}{m_0}\right)$$

And finally,

$$5^A\left(-\frac{1}{2}+\frac{2^{k-1}}{2^{k''}-1}\right)\le 2^B\le 5^A\left(\frac{1}{2}+3\frac{2^k-1}{2^{k''-1}}\right)$$

Thus, we get the new bounds for B after taking a logarithm of the last bounds and taking into account that B is an integer. As the goal of all these computations was to reduce the range for B, we take the smallest values from the new bound and the initial one in $\mathbf{B}$.

$$\begin{aligned}B_{\min}&=\max\left\{\inf(\mathbf{B}),\left\lceil A\log_2 5+\log_2\left(\frac{2^{k-1}}{2^{k''}-1}-\frac{1}{2}\right)\right\rceil\right\}\\ B_{\max}&=\min\left\{\sup(\mathbf{B}),\left\lfloor A\log_2 5+\log_2\left(3\frac{2^k-1}{2^{k''-1}}+\frac{1}{2}\right)\right\rfloor\right\}\end{aligned}\tag{3.50}$$

We get the new bounds for B that depend on A. Thus, we determine the loop order too: the outer loop is the iteration on the full range $\mathbf{A}$, the next one is for B. Numerical experiments have shown that the new range for B contains up to three values for each $A\in\mathbf{A}$. We have two more loops: on S and on T and the range for T can be reduced too. The new upper bound for T may be found from (3.49):

$$2^T\le 2\cdot 5^{-S}\frac{m_1}{m_2}$$

Therefore, taking into account the bounds for m_1 and m_2 (3.44) and that $T\in\mathbb{Z}$ we get the new upper bound

$$T_{\max}=\min\left\{\sup(\mathbf{T}),\left\lfloor -S\log_2 5+2-k'+\log_2(2^k-1)\right\rfloor\right\}\tag{3.51}$$

To find the new lower bound for T we need to prove a lemma.

Lemma 3.8. *Let $m_1,m_2,m_0,A,B,T,S\in\mathbb{Z}$. All the variables are bounded by some integer numbers, numerical values for these bounds are not important for this lemma. When $\max_{S,T}\left|2^T5^S\frac{m_2}{m_0}\right|\le\frac{1}{4}\min_{A,B}\left|\frac{m_1}{m_0}-\frac{2^B}{5^A}\right|$, the following holds:*

$$\min_{A,B,S,T}\left|\frac{m_1+2^T5^Sm_2}{m_0}-\frac{2^B}{5^A}\right|\ge\frac{3}{4}\min_{A,B}\left|\frac{m_1}{m_0}-\frac{2^B}{5^A}\right|$$

Proof. Let us rearrange the fractions in the expression we are interested in:

$$\begin{aligned}\left|\frac{m_1+2^T5^Sm_2}{m_0}-\frac{2^B}{5^A}\right|&=\left|\frac{m_1}{m_0}-\frac{2^B}{5^A}+\frac{2^T5^Sm_2}{m_0}\right|\\ &\ge\left|\frac{m_1}{m_0}-\frac{2^B}{5^A}\right|-\left|2^T5^S\frac{m_2}{m_0}\right|\ge\left|\frac{m_1}{m_0}-\frac{2^B}{5^A}\right|-\max_{S,T}\left|2^T5^S\frac{m_2}{m_0}\right|\\ &\ge\frac{3}{4}\min_{A,B}\left|\frac{m_1}{m_0}-\frac{2^B}{5^A}\right|\end{aligned}$$

We used the property of absolute value $|a|-|b|\le|a+b|$ to prove the lemma. □

The new lower bound for T may be found from not fulfilling conditions of the lemma. We denote the minimum from the lemma as

$$M = \min_{A,B} \left| \frac{m_1}{m_0} - \frac{2^B}{5^A} \right|$$

This minimum may be found with best rational approximations for the fraction $\frac{2^B}{5^A}$. When the lemma conditions are not fulfilled, we get

$$\left| 2^T 5^S \frac{m_2}{m_0} \right| > \frac{1}{4} M.$$

So, after rearranging the products we get

$$2^T > \frac{1}{4} 5^{-S} M \cdot \frac{m_0}{m_2}.$$

An finally, after using the bounds for m_0 and m_2:

$$2^T > \frac{1}{4} 5^{-S} M \cdot \frac{2^{k''-1}}{2^{k'} - 1}$$

So, as soon as we get the value of M with best rational approximations, we can compute the new lower bound for T within the following formula. We take into account that T has to be integer and the new lower bound should be not less than the previous one.

$$T_{\min} = \max \left\{ \inf(\mathbf{T}), \left\lceil -S \log_2 5 + \log_2 \left(\frac{M}{4} \frac{2^{k''-1}}{2^{k'} - 1} \right) \right\rceil \right\} \tag{3.52}$$

Therefore, with the new ranges for the exponent, we established also the loop order: we start with A iterating the whole range $\mathbf{A}$, then goes a small loop on B, then the full range for S and iteration over the reduced range for T.

After the described reduction, the following holds:

$$\max(E_{\max} - E_{\min}) = 3$$
$$\max(T_{\max} - T_{\min}) = 185$$

The total number of iterations is reduced from about 2^{50} to $4406504932 \approx 2^{32}$ or by 99.9996%. However, this number is still quite huge and as on each iteration some computations are performed it is not feasible to find this minimum on one machine in less than two months. As we remember, this is the theory only the addition case, therefore there are even more computations needed to solve the whole problem.

3.4.6 The Full Algorithm

For the addition case with positive inputs all the needed theory is provided. Thus, we may put the final algorithm for this case. The other cases are quite similar with

Input : ranges $\mathbf{A}, \mathbf{B}, \mathbf{S}, \mathbf{T}$, precisions k, k', k''
Output: minimum m, set of the parameters A^*, B^*, S^*, T^* where m is reached

for $A \in \mathbf{A}$ // 1st loop is long
do
  $B_{\min} \leftarrow \max\left\{\inf(\mathbf{B}), \left\lceil A\log_2 5 + \log_2\left(\frac{2^{k-1}}{2^{k''-1}} - \frac{1}{2}\right)\right\rceil\right\}$;
  $B_{\max} \leftarrow \min\left\{\sup(\mathbf{B}), \left\lfloor A\log_2 5 + \log_2\left(3\frac{2^{k-1}}{2^{k''-1}} + \frac{1}{2}\right)\right\rfloor\right\}$;
  for $B \in [B_{\min}, B_{\max}]$ // 2nd loop is short
  do
    $M_{left} \leftarrow$ `bestLeftApproximation`$(2^B/5^A, 2^{k-1}, 2^k - 1, 2^{k''-1}, 2^{k''} - 1)$;
    $M_{right} \leftarrow$ `bestRightApproximation`$(2^B/5^A, 2^{k-1}, 2^k - 1, 2^{k''-1}, 2^{k''} - 1)$;
    if $|M_{left} - \frac{2^B}{5^A}| < |M_{right} - \frac{2^B}{5^A}|$ **then**
      $M \leftarrow M_{left}$;
    else
      $M \leftarrow M_{right}$;
    end
    for $S \in \mathbf{S}$ // 3rd loop is long
    do
      $T_{min} = \max\left\{\inf(\mathbf{T}), \left\lceil -S\log_2 5 + \log_2\left(\frac{M}{4}\frac{2^{k''-1}}{2^{k'}-1}\right)\right\rceil\right\}$;
      $T_{\max} \leftarrow \min\left\{\sup(\mathbf{T}), \left\lfloor -S\log_2 5 + 2 - k' + \log_2(2^k - 1)\right\rfloor\right\}$;
      for $T \in [T_{\min}, T_{\max}]$ // 4th loop is short
      do
        $m_c \leftarrow$ `bestRationalApproximation`$(a_{\min}, a_{\max}, S, T, A, B, k, k', k'')$;
        if $m_c < m$ **then**
          $m \leftarrow m_c$;
          $A^* \leftarrow A$; $B^* \leftarrow B$;
          $S^* \leftarrow S$; $T^* \leftarrow T$;
        end
      end
    end
  end
end
return m,set;

Algorithm 11: Full algorithm for worst cases search in mixed-radix FMA.

some differences that will be discussed in the following sections. The essential part of the algorithm is presented on Algorithm 11. There is a function call to Algorithm 10, that handles the four cases described in Section 3.4.3. We remind, that depending on the signs of T and S numerators for best rational approximation are computed differently as well as the expression to be approximated.

3.4.7 How to Take into Account the Signs of the Inputs

The FMA operation may contain addition $\mathrm{fma}(x, y, z) = \star_k(xy + z)$ or subtraction $\mathrm{fma}(x, y, z) = \star_k(xy - z)$. We considered the case with addition assuming that all the inputs were positive. However, to finish the worst-cases search, the case with subtraction has to be considered too as well as the impact of all the signs. We mentioned earlier that taking into account all the signs there are 16 variants of FMA to be considered: the three inputs and the operation signs may be different. We have reduced the ternary FMA operation to a binary one in (3.37). To remind, this may be written as

$$2^{E_0}5^{F_0}m_0 = \star_k(2^{E_1}5^{F_1}m_1 \pm 2^{E_2}5^{F_2}m_2)$$

Mantissas m_i are positive, thus the sign of the input (if it was negative) has to be written implicitly. Therefore, the complete research of the worst cases should consider the following operation

$$\pm 2^{E_0}5^{F_0}m_0 = \star_k(\pm 2^{E_1}5^{F_1}m_1 \pm \pm 2^{E_2}5^{F_2}m_2)$$

with the constraint $2^{E_1}5^{F_1}m_1 \geq 2^{E_2}5^{F_2}m_2$. The sign of the output $2^{E_0}5^{F_0}m_0$ is determined with the signs of inputs, constraint on their magnitudes, and the operation sign. Therefore, we have to consider now 8 variants of mixed-radix FMA. For the variants with all positive inputs for both addition and subtraction we reduced the problem to minimum search (3.40) and (3.41). For each combination of inputs and operation signs there is one of these two minima to find.

Let us consider an example of FMA when the operation sign is "+", and the inputs are negative, therefore the output's sign is thus negative too:

$$-2^{E_0}5^{F_0}m_0 = \star_k(-2^{E_1}5^{F_1}m_1 + (-2^{E_2}5^{F_2}m_2))$$

Therefore, with the similar reasoning we get

$$-2^{E_0}5^{F_0}m_0 \approx -2^{E_1}5^{F_1}m_1 - 2^{E_2}5^{F_2}m_2$$

which is the same as

$$2^{E_0}5^{F_0}m_0 \approx 2^{E_1}5^{F_1}m_1 + 2^{E_2}5^{F_2}m_2.$$

Thus, this case is similar to the detailed one, with positive inputs and addition. We search for minimum (3.40) here.

Consider an example with "−" sign in FMA and negative inputs:

$$-2^{E_0}5^{F_0}m_0 = \star_k(-2^{E_1}5^{F_1}m_1 - (-2^{E_2}5^{F_2}m_2)).$$

This expression may be rewritten as

$$-2^{E_0}5^{F_0}m_0 \approx -2^{E_1}5^{F_1}m_1 + 2^{E_2}5^{F_2}m_2$$

Thus, the reasoning from Section 3.4.1 brings us to

$$\frac{-m_1 + 2^T 5^S m_2}{m_0} \approx -\frac{2^B}{5^A},$$

which leads to minimum search for subtraction from (3.41).

Similarly, all other variations of signs lead to the two problems of minimum search: the one for additional case and the one for subtraction (3.40)-(3.41). To summarize, we include all the cases in Table 3.3.

	m_1 Sign	m_2 Sign	Operation Sign	Result Sign	Expression for min Search
1	+	+	+	+	$\left\lvert \frac{m_1+2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right\rvert$
2	+	+	−	+	$\left\lvert \frac{m_1-2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right\rvert$
3	+	−	+	+	$\left\lvert \frac{m_1-2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right\rvert$
4	+	−	−	+	$\left\lvert \frac{m_1+2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right\rvert$
5	−	−	+	−	$\left\lvert \frac{m_1+2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right\rvert$
6	−	−	−	−	$\left\lvert \frac{m_1-2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right\rvert$
7	−	+	+	−	$\left\lvert \frac{m_1-2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right\rvert$
8	−	+	−	−	$\left\lvert \frac{m_1+2^T5^S m_2}{m_0} - \frac{2^B}{5^A} \right\rvert$

Table 3.3: FMA variants with taking into account inputs and output signs

3.4.8 What Is Different for the Subtraction Case

The algorithm for best approximations takes positive bounds for numerator and denominator. In subtraction case there is a "−" in the numerator, which may make it negative. Thus, the same reasoning as for addition case cannot be applied straightforward. However, splitting the subtraction case into three subcases allows to establish new bounds for all the variables and thus to solve the problem.

1. $m_1 - 2^T 5^S m_2 \geq \alpha m_1$
2. $0 < m_1 - 2^T 5^S m_2 < \alpha m_1$,
3. $m_1 - 2^T 5^S m_2 < 0$

where $|\alpha| < 1$, e.g. $\alpha = 1/4$.

As well as for the addition case there will be four nested loops. Not only the new bounds for variables change, for some cases the order of loop nesting will be different too. There will be also four ways to compute the numerator for best approximation search, like it was described in Section 3.4.3 and in Algorithm 10. The difference is for the bounds on numerator a.

For the first subcase the loop order stays the same as for addition case: long loop on A, then short one on B, long loop on S and the innermost short on T; the new bounds for B and T are found in the same manner as for the addition case. For two other cases the loop order should be changed but the idea is the same: the first loop iterates over the whole interval for S, the second one is for small range T, then the third loop for A is long again and the fourth for B is short.

3.4.9 Results, Conclusion and Future Work

In this section we have shown how to compute the worst cases for mixed-radix FMA operation. As the operation takes three inputs, the quantity of computations is enormous. Even though the number of iterations is reduced on about 99%, it stays huge. As we need reliable and correct results, all the scripts were written in Sollya [18]. To speed up the whole algorithm, the part with best rational approximations was written directly in C with the use of mpfr [33] and mpz libraries. However, execution of the easiest addition case required more than three months on a standard PC. We used a naive approach to parallel the computations: iterations of the nested loops are independent one from another. Thus, we can split the minimum search into several subtasks: we split the range for the outermost loop (which is A for the addition case) into several non-overlapping subdomains and perform the minimum search on each on these subdomains. The advantage here is that the search of these minima on smaller ranges may be done in parallel. We split the interval for A into 100 equal subintervals and solved a hundred smaller problems. This number was

chosen randomly, such splitting means creation of 100 subtasks of smaller dimension. For the outermost loop we got about 45 iterations. To get the final minimum the minimal answer of these hundred was chosen. This is not an optimal split in terms of iterations: the number of iterations is not the same for each subdomain of A and for some values of A the range for B is empty. As the worst cases search is done only once, this non-optimality is admissible.

After splitting the whole task into 100 smaller tasks, we ran each on the node of cluster BIG in LIP6. The results were already obtained for the addition case and for the first subcase of subtraction. Other scripts are still running. The number of iterations executed for the addition case is 4406504932, for the subtraction case is 4495112310. The result for addition case is about $2.84 \cdot 10^{-80}$ and is reached on the set of parameters $A^* = 96$, $B^* = 273$, $S^* = 2$, $T^* = -132$. The result for the first subcase of subtraction minimum search is about $2.15 \cdot 10^{-80}$ and is reached on the following set of the exponents $A^* = 119, B^* = 326, S^* = -24, T^* = -80$. Performing backward transformations, we may get the binary and decimal FP values for hard roundings, that have to be taken into account during the implementation of mixed-radix FMA. Therefore, we should obtain conditions on parameter ranges when roundings are easy and when they are hard. We may start its implementation when all the results are obtained.

Conclusion and Perspectives

Every human activity, good or bad, except mathematics, must come to an end.

Paul Erdős[4]

In this thesis, we investigated two ways to improve and enlarge the floating-point (FP) environment. One considered the implementation of several different variations for mathematical functions. Another way to enlarge the FP environment is to develop mixed-radix operations. Today it becomes possible to generate implementations for black-box specifications of mathematical function in several minutes. The accuracy of the obtained code is guaranteed by construction, performance is comparable to glibc libm or even better. Till today it is impossible to mix FP numbers of different radices within one operation, except a recent work on comparison. However, this is the natural direction for evolution of the IEEE754 Standard and FP environment. We started research on mixed-radix arithmetic operations from the FMA as its implementation would give addition, subtraction, multiplication and may be reused in certain algorithms for division or square root. Thus, the research on mixed-radix FMA paves the way to mixed-radix arithmetic operations.

Do not Write the Code, Generate It!

Mathematical functions are commonly used but are not required by the IEEE754 Standard as their correctly-rounded results are hard to obtain because of the Table Maker's Dilemma. Recently there is a growing interest in non-standard implementations of mathematical functions: less accurate implementations are usually better in performance. There are some other parameters that may influence performance of the mathematical functions, e.g. final accuracy, implementation domain, degree of polynomial approximation. The state of the art shows that modern mathematical libraries (libms) cannot stay static. They should contain several implementations for each function to provide users with more choices. Implementation of a large quantity

[4]Paul Erdős(1913-1996) was a Hungarian mathematician, known not only for his outstanding scientific results but also for inventing so-called "Erdős number" measure.

of such choices or function flavors as we called them is tedious as well as a choice of flavors to maintain. Metalibm addresses this problem: it gives a possibility to specify the function to be implemented to the user and then generates code for the needed function flavor.

Today there is no more need to write mathematical functions implementations manually. Moreover, there is a possibility to get the code for some specific set of parameters: e.g. non-standard accuracy, or domain smaller than the one defined by the format. This generated code is correct by construction, not in the sense of correctly-rounded result, but in the sense of guarantee for the final accuracy. Metalibm produces generic code, there is no special optimization for some particular hardware, and no parameters for hardware specifications. Therefore, for plain function flavors found in every libm on particular architectures Metalibm cannot outperform the libraries written by the corresponding processor manufacturer teams. However, for "exotic" flavors Metalibm is at least of comparable performance as the standard libms.

The working precision is chosen in order to guarantee the demanded final accuracy of the result. Besides that, Gappa proofs are provided for each generated implementation. Metalibm decides automatically which steps it needs to execute for function implementation: argument reduction and domain splitting, polynomial approximation and reconstruction. Our code generator detects essential algebraic properties that allow it to reduce the domain with some well-known techniques. The list of such properties is not fixed, it may be easily enlarged to support more functions.

We optimized the domain splitting algorithm [53] in order to save memory to store the polynomial coefficients and get the polynomials of maximum possible degree. The new splitting algorithm produces less subdomains and the degrees of the corresponding polynomials are more uniformed. Research on generation of vectorizable implementations has started [52]. Difficulties occur for those function flavors that require domain splitting. The key point of vectorizable code generation is to avoid branching, therefore to avoid *if-else* statements used to determine the right polynomial coefficients for the input values. The proposed technique replaces this branching by a polynomial function. However, it uses a posteriori condition checks and we cannot know beforehand if this procedure finishes with success.

Mix the Floating-Point Numbers of Different Radices

The second direction in enlarging the FP environment is research on mixed-radix operations. The 2008 version of the IEEE754 Standard required operations that mix different formats of the same radix, so it is quiet natural to evolve to the idea of mixing radices. A novel algorithm of radix conversion was developed: the computations are done in integer arithmetic, so no FP flags are affected. To determine the FP

number that is the result of this radix conversion we need to determine its two fields: exponent and mantissa. Exponent determination is straightforward and performed with several basic arithmetic operations and a look-up table. Computation of the mantissa uses a small exact table.

These tables are then reused in the proposed algorithm of scanf analogue on FP numbers. This is a conversion operation from decimal character sequence of arbitrary length to a binary FP number. We proposed a novel algorithm that is independent from the current rounding mode. Its memory consumption is known beforehand. Thus, this code is re-entrant and may be used in embedded systems. The research on a mixed-radix version of FMA operation has started with the worst cases search. We have shown how to avoid brute-force searching with the use of continued fractions and establishing relations between some parameters. However, the complete search requires too many computations and cannot be finished on one machine in reasonable time. We obtained the first results of this search recently. I hope that mixed-radix operations will be present in one of the next revisions of IEEE754 Standard.

Perspectives

Metalibm produces flexible implementations for parametrized mathematical functions. However, for the moment it does not generate code to filter out the special cases, e.g. NaNs, infinities or too large inputs that cause overflow. As the complete implementation of a mathematical function always contains this filtering step, this is a short-run goal for future work in code generation direction. Polynomial approach for the vectorization does not work for all the flavors and we discussed the two approaches to improve it in Section 2.2.5. A mid-term goal for the Metalibm project is implementation of these new reconstruction procedures for vectorizable code.

Metalibm generates too generic code that cannot outperform implementations with specific instructions selection. Therefore, an interesting direction is to add hardware specification as a parameter for generation. However, that will make our Metalibm similar with its analogue that we mentioned earlier [11, 12]. This is also a code generator for mathematical functions, the difference is that it does not take black-box functions and as it takes hardware specification as a parameter it optimizes the instruction set for the produced code. Our generator is a "push-button" approach while another one is mostly an assistant tool for function developers. The two projects have a lot of common points, so the strong distinction is hard to be established and is a topic for long discussions. Thus, an interesting and ambitions perspective would be to merge the two approaches for fully-parametrized libm generation.

Metalibm could be used to generate the functions for currently-existing libms and probably to replace the existing implementations. As it does not use any specific

instruction selection for the moment, the generated versions may be too slow in comparison with some particular libms. Metalibm generates code on demand and guarantees accuracy by construction, while the existing mathematical libraries are completely static. However, some integrated version of existing libms and Metalibm can be useful: for slow but accurate implementations Metalibm generated code could be used, and for fast or default versions the current code from libms. It would be difficult to integrate Metalibm to any of the existing libraries: for the moment there is no mechanism to support and to choose among several function implementations. However, inclusion of generated implementations or even the generator to the existing libraries is an interesting future direction. The expertise on this is mostly on libm or compiler developers now.

We did not provide any guidelines on the choice of parameters. For example, the table size for table-driven implementations may depend on some particular architecture. If we are generating a generic flavor that will be run on various machines, how can this value be chosen? The same questions arise for other parameters: degree bound for polynomial approximation or even the final accuracy. The main bonus of the code generator is that we can produce various implementations, measure and compare them in some sense (performance for example). Then the best implementation may be easily chosen. Thus, a tool for Metalibm that helps the users to choose the best parameter set could be useful. Therefore, the users may specify admissible intervals for all the parameters, generate several implementations for all the possible combinations and then pick the best one.

Metalibm generates proof for the polynomial approximations. Specific argument reduction procedures bring their errors too. However, we cannot completely prove the final accuracy for such function implementations. This is another direction in Metalibm development.

The first results for FMA worst cases search were obtained recently, therefore this search has to be finished in the shortest terms. Once we get all the worst cases, the implementation of mixed-radix FMA can be started. We reduced the problem to the minimum search of the expression with several parameters (seven, to be precise). The four of them were the exponents of 2 and 5, that were obtained from the exponents of the input numbers. Therefore, backward transition is also possible and having the set of the exponents for the worst cases, in the implementation of the mixed-radix FMA we can divide the inputs into simple and hard rounding subroutines. The algorithm for the mixed-radix FMA needs to be developed, proven, implemented and thoroughly tested.

As mentioned, FMA is a base in mixed-radix arithmetic research: once implemented, we get immediately multiplication, addition and subtraction. The future goal is to develop algorithms for all the other mixed-radix arithmetic operations. This requires worsts cases search for each operation. In this worst cases search for FMA we used several techniques to reduce the quantity of iterations. However, it

still stays large and the proposed method is not appliable to 128-bit formats. A novel technique should be found for this.

The algorithm for arbitrary precision base conversion is complicated and contains a lot of mathematical deductions, therefore is of great interest to publish too. This is an analogue of scanf function, so its implementation could interest some colleagues from industry. The similar algorithm should be developed for prinf analogue: conversion from binary FP number to decimal character sequence. We assume that this one should be easier to develop that the scanf: binary FP numbers have finite precision. The trick will be to get the identity operation as a superposition of these two conversions.

Developed algorithm for conversion from decimal string representation to a binary FP number is based on lots of theorems proven in this thesis. However, serious testing and comparison with the existing methods is needed. As the length of the user input is arbitrary, the number of inputs tends to infinite, therefore testing all the amount of possible inputs is not feasible. Future work here may consider bringing the formal proofs such as in Coq. There might be added another path for producing the result: when there is no rounding needed, the result should be obtained without extra computations.

Bibliography

[1] Semantics of Floating Point Math in GCC. `https://gcc.gnu.org/wiki/FloatingPointMath`.

[2] *Intel®Math Kernel Library. Reference Manual*. Intel Corporation, 2009.

[3] 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.

[4] ARM. *ARM®Developer Suite, Version 1.2, Compilers and Libraries Guide*, 1999-2001.

[5] K. Atkinson. *An Introduction to Numerical Analysis*. Wiley, 2 edition.

[6] S. Boldo and G. Melquiond. When double rounding is odd. In *Proceedings of the 17th IMACS World Congress on Computational and Applied Mathematics*, Paris, France, 2005.

[7] S. Boldo and G. Melquiond. Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd. *IEEE Trans. Computers*, 57(4):462–471, 2008.

[8] N. Brisebarre and S. Chevillard. Efficient polynomial L∞-approximations. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007), 25-27 June 2007, Montpellier, France*, pages 169–176, 2007.

[9] N. Brisebarre, M. Mezzarobba, J. Muller, and C. Q. Lauter. Comparison between binary64 and decimal64 floating-point numbers. In *21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013*, pages 145–152, 2013.

[10] N. Brisebarre and J.-M. Muller. Correctly rounded multiplication by arbitrary precision constants. *IEEE Transactions on Computers*, 57(2):165–174, Feb. 2008.

[11] N. Brunie. *Contributions to computer arithmetic and applications to embedded systems*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2014.

[12] N. Brunie, F. de Dinechin, O. Kupriianova, and C. Lauter. Code generators for mathematical functions. In *ARITH22 - June 2015, Lyon, France - book of proceedings*, pages 66–73, 2015.

[13] W. Buchholz. Fingers or fists? (the choice of decimal or binary representation). *Commun. ACM*, 2(12):3–11, Dec. 1959.

[14] R. Burger and R. K. Dybvig. Printing floating-point numbers quickly and accurately. In *In Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 108–116, 1996.

[15] P. E. Ceruzzi. The early computers of Konrad Zuse, 1935 to 1945. *IEEE Annals of the History of Computing*, 3(3):241–262, 1981.

[16] E. Cheney. *Introduction to Approximation Theory*. Numerical mathematics and scientific computation. Chelsea Publishing Company, 1982.

[17] S. Chevillard. *Évaluation efficace de fonctions numériques – Outils et exemples*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2009.

[18] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. 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 28–31, Heidelberg, Germany, September 2010. Springer.

[19] S. Chevillard, M. Joldes, and C. Q. Lauter. Certified and fast computation of supremum norms of approximation errors. In *19th IEEE Symposium on Computer Arithmetic, ARITH 2009, Portland, Oregon, USA, 9-10 June 2009*, pages 169–176, 2009.

[20] S. Chevillard, C. Lauter, and M. Joldeş. *User's manual for the `Sollya` tool*.

[21] T. Coe. Inside the Pentium FDIV bug. *Dr. Dobb's Journal of Software Tools*, 20(4):129–135, Apr. 1995.

[22] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev. A software implementation of the IEEE 754r decimal floating-point arithmetic using the binary encoding format. *IEEE Trans. Computers*, 58(2):148–162, 2009.

[23] M. F. Cowlishaw. Decimal floating-point: Algorism for computers. In *16th IEEE Symposium on Computer Arithmetic (Arith-16 2003), 15-18 June 2003, Santiago de Compostela, Spain*, pages 104–111, 2003.

[24] M. F. Cowlishaw, E. M. Schwarz, R. M. Smith, and C. F. Webb. A decimal floating-point specification. In *15th IEEE Symposium on Computer Arithmetic (Arith-15 2001), 11-17 June 2001, Vail, CO, USA*, pages 147–154, 2001.

[25] F. de Dinechin. Générateurs de code pour les fonctions mathématiques et les filtres, `http://www.agence-nationale-recherche.fr/?Projet=ANR-13-INSE-0007`, 2013.

[26] F. de Dinechin, C. Q. Lauter, and G. Melquiond. Certifying the floating-point implementation of an elementary function using gappa. *IEEE Trans. Computers*, 60(2):242–253, 2011.

[27] D. Defour. *Fonctions élémentaires: algorithmes et implémentations efficaes pour l'arrondi correct en double précision*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2003.

[28] D. Defour, F. de Dinechin, C. Lauter, and C. Daramy-Loirat. Crlibm, a library of correctly rounded elementary functions in double-precision.

[29] T. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971/72.

[30] M. D. Ercegovac. A general method for evaluation of functions and computations in a digital computing. In T. R. N. Rao and D. W. Matula, editors, *3rd IEEE Symposium on Computer Arithmetic, ARITH 1975, Dallas, TX, USA, November 19-20, 1975*, pages 147–157. IEEE Computer Society, 1975.

[31] M. D. Ercegovac and T. Lang. *Digital arithmetic*. Morgan Kaufmann Oxford, San Francisco (Calif.), 2004.

[32] R. M. S. et al. *The GNU C Library Reference Manual*. Free Software Foundation, Inc.

[33] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, June 2007.

[34] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[35] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate scientific computations*. Springer, 1986.

[36] S. Gal. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Trans. Math. Softw.*, 17(1):26–45, 1991.

[37] D. M. Gay. Correctly rounded binary-decimal and decimal-binary conversions. Numerical Analysis Manuscript 90-10, nov 1990.

[38] D. Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991.

[39] M. Gouicem. *Conception et implantation d'algorithmes efficaces pour la résolution du dilemme du fabricant de tables sur architectures parallèles.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 2013.

[40] S. Graillat and V. Ménissier-Morain. Accurate summation, dot product and polynomial evaluation in complex floating point arithmetic. *Inf. Comput.*, 216:57–71, 2012.

[41] N. J. Higham. *Accuracy and stability of numerical algorithms (2 ed.).* SIAM, 2002.

[42] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic.* IEEE, New York, Aug. 12 1985.

[43] IEEE Standard for Floating-Point Arithmetic. IEEE 754-2008, also ISO/IEC/IEEE 60559:2011, Aug. 2008.

[44] 854-1987 IEEE Standard for Radix-Independent Floating-Point Arithmetic, 1987.

[45] V. Innocente. Floating point in experimental HEP data processing. In *2nd CERN Openlab/INTEL Workshop on Numerical Computing*, 2012.

[46] C. Jeannerod, N. Louvet, and J. Muller. On the componentwise accuracy of complex floating-point division with an FMA. In A. Nannarelli, P. Seidel, and P. T. P. Tang, editors, *21st IEEE Symposium on Computer Arithmetic, ARITH, Austin, TX, USA, April 7-10, 2013*, pages 83–90. IEEE Computer Society, 2013.

[47] W. Kahan. Why do we need a floating-point arithmetic standard. Technical report, University of California at Berkeley, 1981.

[48] W. Kahan. A logarithm too clever by half. http://www.cs.berkeley.edu/∼wkahan/LOG10HAF.TXT, 2004.

[49] N. Kapre and A. DeHon. Accelerating SPICE model-evaluation using FPGAs. *Field-Programmable Custom Computing Machines*, pages 37–44, 2009.

[50] A. Y. Khinchin. *Continued Fractions.* Courier Dover Publication, 1997.

[51] O. Kupriianova and C. Lauter. The libieee754 compliance library for the IEEE754-2008 Standard. In *15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 2012.

[52] O. Kupriianova and C. Lauter. Replacing branches by polynomials in vectorizable elementary functions. In *16th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 2014.

[53] O. Kupriianova and C. Q. Lauter. A domain splitting algorithm for the mathematical functions code generator. In *2014 Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, November 2-5, 2014*, 2014.

[54] O. Kupriianova and C. Q. Lauter. Metalibm: A mathematical functions code generator. In *Mathematical Software - ICMS 2014 - 4th International Congress, Seoul, South Korea, August 5-9, 2014. Proceedings*, pages 713–717, 2014.

[55] O. Kupriianova, C. Q. Lauter, and J.-M. Muller. Radix conversion for IEEE754-2008 mixed radix floating-point arithmetic. In *2013 Asilomar Conference on Signals, Systems and Computers, Pacific Grove, CA, USA, November 3-6, 2013*, pages 1134–1138, 2013.

[56] C. Lauter. *Arrondi correct de fonctions mathématiques. Fonctions univariées et bivariées, certification et automatisation.* PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2008.

[57] C. Lauter and M. Mezzarobba. Semi-automatic floating-point implementation of special functions. In *ARITH22 - June 2015, Lyon, France - book of proceedings*, pages 58–65, 2015.

[58] D.-U. Lee, P. Cheung, W. Luk, and J. Villasenor. Hierarchical segmentation schemes for function evaluation. *IEEE Transactions on VLSI Systems*, 17(1), 2009.

[59] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable.* PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[60] P. J. L. Lions. Ariane 5 – flight 501 failure. Technical report, Inquiry board, 1996.

[61] N. Louvet, J. Muller, and A. Panhaleux. Newton-raphson algorithms for floating-point division using an FMA. In F. Charot, F. Hannig, J. Teich, and C. Wolinski, editors, *21st IEEE International Conference on Application-specific Systems Architectures and Processors, ASAP 2010, Rennes, France, 7-9 July 2010*, pages 200–207. IEEE, 2010.

[62] D. Matula. A formalization of floating-point numeric base conversion. *IEEE Transactions on Computers*, 19(8):681–692, 1970.

[63] G. Melquiond. *User's Guide for Gappa*.

[64] Michael J. Schulte and Earl E. Swartzlander Jr. Exact rounding of certain elementary functions. In E. S. Jr., M. J. Irwin, and G. A. Jullien, editors, *11th Symposium on Computer Arithmetic, 29 June - 2 July 1993, Windsor, Canada, Proceedings.*, pages 138–145. IEEE Computer Society/, 1993.

[65] C. Mouilleron and G. Revy. Automatic generation of fast and certified code for polynomial evaluation. In E. Antelo, D. Hough, and P. Ienne, editors, *20th IEEE Symposium on Computer Arithmetic, ARITH 2011, Tübingen, Germany, 25-27 July 2011*, pages 233–242. IEEE Computer Society, 2011.

[66] J.-M. Muller. *Elementary Functions: Algorithms and Implementation*. Birkhauser Boston, Inc., Secaucus, NJ, USA, 1997.

[67] J.-M. Muller. On the definition of ulp(x). Technical Report RR-5504, INRIA, Feb. 2005.

[68] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Boston, 2010.

[69] M. H. Payne and R. N. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsl.*, 18(1):19–24, Jan. 1983.

[70] D. Piparo. The VDT mathematical library. In *2nd CERN Openlab/INTEL Workshop on Numerical Computing*, 2012.

[71] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[72] B. Randell. From analytical engine to electronic digital computer: The contributions of ludgate, torres, and bush. *IEEE Annals of the History of Computing*, 4(4):327–341, 1982.

[73] E. Remez. *Sur un procédé convergent d'approximations successives pour déterminer les polynômes d'approximation.* Académie des Sciences, Paris, 1934.

[74] S. Shane. Mechanism to detect IEEE underflow exceptions on speculative floating-point operations. http://www.google.com/patents/WO2001033341A1?cl=en, May 10 2001. WO Patent App. PCT/US2000/025,490.

[75] S. P. Shary. Solving the linear interval tolerance problem. *Mathematics and Computers in Simulation*, 39(1–2):53 – 85, 1995.

[76] S. P. Shary. *Interval algebraic problems and their numerical solution*. PhD thesis, Novosibirsk Institute of Computational Mathematics and Mathematical Geophysics, 2000.

[77] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–368, 1997.

[78] G. L. Steele Jr. and J. L. White. How to print floating-point numbers accurately. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, pages 112–126, 1990.

[79] P. Tang. Binary-integer decimal encoding for decimal floating point. Technical report, Software and Solutions Group, Intel Corporation, 2005.

[80] P.-T. P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Trans. Math. Softw.*, 15(2):144–157, June 1989.

[81] P. T. P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Trans. Math. Softw.*, 16(4):378–400, 1990.

[82] P. T. P. Tang. Table-driven implementation of the expm1 function in IEEE floating-point arithmetic. *ACM Trans. Math. Softw.*, 18(2):211–222, 1992.

[83] T. Viitanen, P. Jaaskelainen, O. Esko, and J. Takala. Simplified floating-point division and square root. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 2707–2711, May 2013.

[84] T. Viitanen, P. Jaaskelainen, and J. Takala. Inexpensive correctly rounded floating-point division and square root with input scaling. In *Signal Processing Systems (SiPS), 2013 IEEE Workshop on*, pages 159–164, Oct 2013.

[85] C. Vinschen and J. Johnston. Newlib. `https://sourceware.org/newlib/`.

[86] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[87] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Softw.*, 17(3):410–423, Sept. 1991.

Abstract

This work investigates two ways of enlarging the current floating-point environment. The first is to support several implementation versions of each mathematical function (elementary such as exp or log and special such as erf or Γ), the second one is to provide IEEE754 operations that mix the inputs and the output of different radices. As the number of various implementations for each mathematical function is large, this work is focused on code generation. Our code generator supports the huge variety of functions: it generates parametrized implementations for the user-specified functions. So it may be considered as a black-box function generator. This work contains a novel algorithm for domain splitting and an approach to replace branching on reconstruction by a polynomial. This new domain splitting algorithm produces less subdomains and the polynomial degrees on adjacent subdomains do not change much. To produce vectorizable implementations, *if-else* statements on the reconstruction step have to be avoided.

Since the revision of the IEEE754 Standard in 2008 it is possible to mix numbers of different precisions in one operation. However, there is no mechanism that allows users to mix numbers of different radices in one operation. This research starts an examination of mixed-radix arithmetic with the worst cases search for FMA.

A novel algorithm to convert a decimal character sequence of arbitrary length to a binary floating-point number is presented. It is independent of currently-set rounding mode and produces correctly-rounded results.

Keywords: Computer arithmetic, floating-point numbers, IEEE754 Standard, elementary functions, code generator, Metalibm, argument reduction, domain splitting, mixed-radix arithmetic, FMA, radix conversion

---

Résumé

Cette thèse fait une étude sur deux moyens d'enrichir l'environnement flottant courant : le premier est d'obtenir plusieurs versions d'implantation pour chaque fonction mathématique (élémentaires comme $\exp, \log$ et spéciales comme erf, Γ), le deuxième est de fournir des opérations de la norme IEEE754, qui permettent de mélanger les entrées et la sortie dans les bases différentes. Comme la quantité de versions différentes pour chaque fonction mathématique est énorme, ce travail se concentre sur la génération du code. Notre générateur de code adresse une large variété de fonctions: il produit les implantations paramétrées pour les fonctions définies par l'utilisateur. Il peut être vu comme un générateur de fonctions boîtes-noires. Ce travail inclut un nouvel algorithme pour le découpage de domaine et une tentative de remplacer les branchements pendant la reconstruction par un polynôme. Le nouveau découpage de domaines produit moins de sous-domaines et les degrés polynomiaux sur les sous-domaines adjacents ne varient pas beaucoup. Pour fournir les implantations vectorisables il faut éviter les branchements *if-else* pendant la reconstruction.

Depuis la révision de la norme IEEE754 en 2008, il est devenu possible de mélanger des nombres de différentes précisions dans une opération. Par contre, il n'y a aucun mécanisme qui permettrait de mélanger les nombres dans des bases différentes dans une opération. La recherche dans l'arithmétique en base mixte a commencé par les pires cas pour le FMA.

Un nouvel algorithme pour convertir une suite de caractères décimaux du longueur arbitraire en nombre flottant binaire est présenté. Il est indépendant du mode d'arrondi actuel et produit un résultat correctement arrondi.

Mots-Clés: Arithmétique des ordinateurs, virgule flottante, norme IEEE754, fonctions élémentaires, variantes de fonctions, générateur de code, Metalibm, réduction d'argument, découpage de domaine, arithmétique en base mixte, FMA, conversion de base