HAL open science

Floating-point arithmetic

Sylvie Boldo, Claude-Pierre Jeannerod, Guillaume Melquiond, Jean-Michel Muller

To cite this version:

Sylvie Boldo, Claude-Pierre Jeannerod, Guillaume Melquiond, Jean-Michel Muller. Floating-point arithmetic. Acta Numerica, 2023, 32, pp.203-290. 10.1017/S0962492922000101 . hal-04095151

HAL Id: hal-04095151

https://hal.science/hal-04095151v1

Submitted on 11 May 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Acta Numerica (2023), pp. 203–290 Printed in the United Kingdom
doi:10.1017/S0962492922000101

Floating-point arithmetic

Sylvie Boldo
Université Paris Saclay, CNRS, ENS Paris Saclay, Inria, LMF,
91190 Gif-sur-Yvette, France
E-mail: sylvie.boldo@inria.fr

Claude-Pierre Jeannerod
Inria, ENS de Lyon, LIP, 69364 Lyon, France
E-mail: claude-pierre.jeannerod@inria.fr

Guillaume Melquiond
Université Paris Saclay, CNRS, ENS Paris Saclay, Inria, LMF,
91190 Gif-sur-Yvette, France
E-mail: guillaume.melquiond@inria.fr

Jean-Michel Muller
CNRS, ENS de Lyon, LIP, 69364 Lyon, France
E-mail: jean-michel.muller@cnrs.fr

Floating-point numbers have an intuitive meaning when it comes to physics-based numerical computations, and they have thus become the most common way of approximating real numbers in computers. The IEEE-754 Standard has played a large part in making floating-point arithmetic ubiquitous today, by specifying its semantics in a strict yet useful way as early as 1985. In particular, floating-point operations should be performed as if their results were first computed with an infinite precision and then rounded to the target format. A consequence is that floating-point arithmetic satisfies the 'standard model' that is often used for analysing the accuracy of floating-point algorithms. But that is only scraping the surface, and floating-point arithmetic offers much more.

© The Author(s), 2023. Published by Cambridge University Press.
This is an Open Access article, distributed under the terms of the Creative Commons Attribution licence (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted re-use, distribution, and reproduction in any medium, provided the original work is properly cited.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

In this survey we recall the history of floating-point arithmetic as well as its specification mandated by the IEEE-754 Standard. We also recall what properties it entails and what every programmer should know when designing a floating-point algorithm. We provide various basic blocks that can be implemented with floating-point arithmetic. In particular, one can actually compute the rounding error caused by some floating-point operations, which paves the way to designing more accurate algorithms. More generally, properties of floating-point arithmetic make it possible to extend the accuracy of computations beyond working precision.

2020 Mathematics Subject Classification: Primary 65Y04
Secondary 65G50, 68M07

CONTENTS

1 Introduction 204
2 Formats, roundings and operations 207
3 Execution environment 231
4 Taming rounding errors 240
5 Extending the precision 267
6 Conclusion 277
References 279

1. Introduction

Floating-point arithmetic has a long history. In one respect, as advocated by Knuth (1998), it can be traced back to the Babylonian base-60 number system, invented around 2000 BC, with which (due to the lack of a digit for zero) only the *floating-point significands* of numbers were represented, that is, the numbers a, $a/60$ and $a \cdot 60^5$ had the same representation (Iffrah 1999). The invention of the exponents took place in several stages. A kind of exponential notation for representing huge numbers was described by Archimedes (c. 287–212 BC) in his treatise *The Sand Reckoner* (Hirshfeld 2009). It seems that the first person to consider zero or negative exponents was Nicolas Chuquet (c. 1450–1488) (Flegg, Hay and Moss 1985). Some of the first computers, designed from the 1940s on, provided a floating-point arithmetic. Notably enough, Konrad Zuse's Z3 computer, completed by the end of 1941, had a binary floating-point arithmetic far ahead of its time, with special representations for undefined results and infinities (Ceruzzi 1981). Many very different solutions (various bases – 2, 8, 16, 10, even 3 – various precisions, various ways of rounding the inexact operations and handling the exceptional cases such as forbidden operations, overflows and underflows, etc.) were introduced in the 1950s, 1960s and 1970s, resulting in the total chaos well described by Kahan (1981) in his paper 'Why do we need a floating-point arithmetic standard?' Sometimes writing portable software required magic tricks only known by old programming

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

wizards, such as inserting, at well-chosen places, multiplications by 1, or adding 0.5 twice instead of adding 1. One could check that a variable z is non-zero and despite this obtain a 'zero divide' error when attempting to divide some number by z.

The IEEE-754 Standard for Floating-Point Arithmetic, adopted in 1985, put an end to the chaos. It greatly facilitated the design and portability of numerical software. The reader interested in the history of the birth of the Standard can read the interview with William Kahan by Severance (1998). A significant revision of IEEE-754 was published in 2008, and a minor revision of the 2008 version was released in 2019 (IEEE 2019). Since 2012, the IEEE Standards have had only a 10-year validity period;[1] a new revision is therefore expected to be published around 2029. It will likely be a thorough revision since the domain has been drastically evolving in recent years.

Today, most FPUs are compliant with IEEE-754. The first generations of GPUs were not compliant, but the situation has improved substantially and one can perform IEEE-754 arithmetic on most of them. In contrast, machine learning cores are nowhere near this state, as the frequent lack of clear documentation on their arithmetic makes it very difficult to prove anything on the behaviour of an algorithm. To mitigate this difficulty and guess what the arithmetic operators do exactly, one must craft carefully designed numerical tests (Fasi, Higham, Mikaitis and Pranesh 2021). This is reminiscent of the first days of the IEEE-754 era, when one had to run software such as PARANOIA (Karpinsky 1985) to check for compliance with the Standard.

It is commonplace that computers are much faster today than 35 years ago. However, the pace of this impressive performance increase has not been the same for all parts of our machines. Between 1986 (i.e. just after the release of the first version of the IEEE Standard for Floating-Point Arithmetic) and 2000, the improvement factor per year has been around 1.52 in processor performance, and 1.07 in memory latency (Hennessy and Patterson 2012, Chapter 2). As a consequence, the ratio

$$\frac{\text{time to read/write in memory}}{\text{time to perform } +, \times, \div, \sqrt{}}$$

has increased by a factor of around 140 between 1986 and 2000. It has continued to increase since 2000, but at a somewhat slower pace. Figure 1.1 presents typical current latencies of arithmetic operations and access to cache or main memory.

Due to this drastic evolution the current situation of numerical computing is quite different from the situation at the time of the birth of IEEE-754. The challenge is no longer (or, at least, not only) to design fast arithmetic operators, but to be able to feed them with data at a very high rate, so that they do not become idle too often. All of this has led to the implementation of many new architectural concepts: multiple levels of cache, pipelining, vector instructions, branch prediction, and so

[1] https://standards.ieee.org/faqs/maintenance/

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

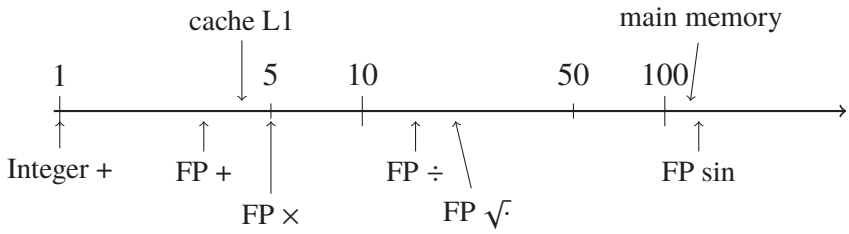

Figure 1.1. Typical current latencies (in number of cycles) of various floating-point (FP) operations and cache/memory access (beware: the scale is logarithmic). These figures vary from one processor to another, but the orders of magnitude remain similar. With a 2 GHz processor, one cycle is 0.5 nanoseconds.

on. The impact of these changes on floating-point computing is very important, as heavy pipelining makes division and square root significantly slower than addition or multiplication, and may make branching very costly, unless some regularity in the branching patterns allows for a very efficient branch prediction. It also makes checking the IEEE-754 flags (overflow, inexact, etc.) without losing too much performance an almost hopeless task. Numbers represented in very small formats (e.g. 16-bit numbers) are processed faster than the 'usual' binary32 or binary64 numbers (in the delay required to transfer one 128-bit number to/from memory, one can transfer eight 16-bit numbers), so one is tempted to use these small formats whenever possible, and hence do *mixed-precision* arithmetic (Higham and Mary 2022). An example is digital neural network training, which requires a huge amount of calculations, that are performed on numbers of very small widths (16-bit, 8-bit or even less). This, however, requires considerable care. Whereas *catastrophic* events such as overflow, underflow and total loss of accuracy are relatively rare when computing in binary64/double-precision arithmetic, they can occur very quickly when using small formats. And, with many different arithmetics emerging, there is the fear of going back to the pre-1985 chaos . . .

Another significant change in the last 40 years lies in the *applications* of floating-point computing. Whereas most users of 1985 were safely running numerical simulation programs on the ground, embedded computing is now ubiquitous, with numerical calculations being performed on board trains, aircraft and partly automated cars, with potential risks to human lives if something goes wrong. Rigorous validation of critical numerical software is now essential. As pointed out by Demmel and Riedy (2021), *at a high level, one significant change is the burgeoning demand for reliability.* This is especially true of the basic arithmetic software, since all of numerical computing is built upon it. The use of formal proof techniques has become more and more important since the years following the Pentium FDIV bug (Moore, Lynch and Kaufmann 1998, Harrison 1999, Cornea-Hasegan, Golliver and Markstein 1999). This is especially true given that the proofs of some arithmetic algorithms are rather long and tedious, with the consequence that they are seldom

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Figure 2.1. The FP numbers between 1/2 and 8 in the toy system $\beta = 2$, $p = 3$ and infinite $e_{\min}$ and $e_{\max}$.

read and therefore an error may remain unnoticed. See the book by Boldo and Melquiond (2017) for a thorough presentation, and the work by Muller and Rideau (2022) for recent examples in double-word arithmetic.

Throughout the history of floating-point arithmetic, useful material has been published describing what needs to be known by an applied mathematician, a computer scientist or an engineer. Still of great interest are the article by Goldberg (1991), 'What every computer scientist should know about floating-point arithmetic', the books by Overton (2001) and Higham (2002), and the notes of Kahan (1997, 2004*a*). More recently, some of us contributed to a *Handbook of Floating-Point Arithmetic* (Muller *et al.* 2018). Much useful information on floating-point arithmetic (especially for C programmers) and IEEE-754 can be found in Beebe (2017, Chapter 4). Finally, a very useful overview was given recently by Higham (2021*a*).

2. Formats, roundings and operations

Let us first define the floating-point numbers. We choose the definition given by Muller *et al.* (2018), inspired by the IEEE-754-2019 Standard for Floating-Point Arithmetic. We assume in the following that $\beta \geq 2$, $p \geq 2$, $e_{\min}$ and $e_{\max}$ are integers, with $e_{\min} < 0 < e_{\max}$ (in all practical cases, $e_{\min} = 1 - e_{\max}$).

Definition 2.1. The set $\mathbb{F}_{\beta,p,e_{\min},e_{\max}}$ of the base-β, precision-p floating-point numbers with extremal exponents $e_{\min}$ and $e_{\max}$ is the set of the real numbers x for which there exists at least one representation (M, e) such that

$$x = M \cdot \beta^{e-p+1}, \tag{2.1}$$

where M and e are integers satisfying

$$|M| \leq \beta^p - 1, \tag{2.2}$$

and

$$e_{\min} \leq e \leq e_{\max}. \tag{2.3}$$

M and e are called the *integral significand* and the *exponent* of the representation of x, respectively.

Figure 2.1 shows where the FP numbers sit on the real line on a toy system. Note that the distance between consecutive FP numbers is multiplied by β each time a power of β is crossed over (see e.g. around the value 4 in the figure).

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

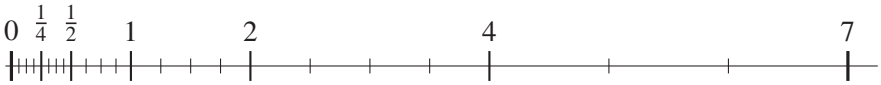

Figure 2.2. The positive FP numbers in the toy system $\beta = 2$, $p = 3$, $e_{\min} = -2$, $e_{\max} = 2$.

In the following, the set $\mathbb{F}_{\beta,p,e_{\min},e_{\max}}$ will be denoted by $\mathbb{F}$ whenever the values of β, p, $e_{\min}$ and $e_{\max}$ are unambiguous. We also define $\overline{\mathbb{F}} = \mathbb{F} \cup \{-\infty, +\infty\}$ and $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$. For a given non-zero floating-point number x, several possible values of M and e may satisfy (2.1), (2.2) and (2.3). The *normalized* representation of x is the one for which $|M|$ is maximal (or equivalently e is minimal). The number zero is rather special, and in practice a special representation is reserved for it. The integral significand of the normalized representation of x is called the integral significand of x, denoted M_x, and the exponent of the normalized representation of x is called the exponent of x, denoted e_x. We say that a non-zero number $x \in \mathbb{F}$ is a *normal number* if $|x| \geq \beta^{e_{\min}}$, and a *subnormal number* otherwise (according to IEEE-754-2019, zero is neither normal nor subnormal). Without difficulty, we find that if $x \in \mathbb{F}$ is normal then $|M_x| \geq \beta^{p-1}$. The number $\beta^{e_{\min}}$ is sometimes called the *underflow threshold.*

The largest element of $\mathbb{F}$ is $\mathbf{\Omega} = \beta^{e_{\max}+1} - \beta^{e_{\max}-p+1}$. The smallest positive element of $\mathbb{F}$ is $\alpha = \beta^{e_{\min}-p+1}$. All elements of $\mathbb{F}$ are integer multiples of α. We call the set $[-\mathbf{\Omega}, -\beta^{e_{\min}}] \cup [\beta^{e_{\min}}, \mathbf{\Omega}]$ the *normal domain* and the set $(-\beta^{e_{\min}}, \beta^{e_{\min}})$ the *subnormal domain*. There are $2 \cdot (\beta^p - \beta^{p-1}) \cdot (e_{\max} - e_{\min} + 1)$ normal numbers and $2 \cdot (\beta^{p-1} - 1)$ subnormal numbers.

Figure 2.2 shows where all the positive FP numbers of a toy binary system sit on the real line. The difference from Figure 2.1 lies in the given values for $e_{\min}$ and $e_{\max}$. The smallest positive value is $\alpha = 2^{e_{\min}-p+1} = 2^{-4} = 1/16$ and the largest is $\mathbf{\Omega} = 7$. The subnormal numbers are $1/16$, $1/8$ and $3/16$ and the smallest normal number is $2^{e_{\min}} = 1/4$.

The following property immediately follows from Definition 2.1.

Property 2.2. If a real number x is an integer multiple of β^k, with $k \geq e_{\min} - p + 1$ and $|x| \leq \min\{\mathbf{\Omega}, \beta^{k+p}\}$, then $x \in \mathbb{F}$.

In all cases of interest for a numerical analyst,[2] β is equal to 2, but base 10 is provided in any case on most systems (mostly for accounting applications). Other bases (such as 4 or 8) were used in the early days of computer arithmetic, but studies performed in the 1970s showed that they are of little interest (Brent 1973, Kuki and Cody 1973). *As a consequence, from now on, unless stated otherwise, we assume a binary floating-point arithmetic, that is,*

$$\beta = 2.$$

[2] A possible exception is the decimal arithmetic of the computer algebra system Maple.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Table 2.1. Main parameters of the binary interchange formats specified by the IEEE-754-2019 Standard (IEEE 2019), and of the bfloat16 format (Intel 2018).

	binary16	binary32	binary64	binary128	bfloat16
Former name		single precision	double precision		
p	11	24	53	113	8
$e_{\max}$	+15	+127	+1023	+16383	+127
$e_{\min}$	−14	−126	−1022	−16382	−126

One of the advantages of base 2 is that the leftmost digit of the significand of a floating-point number is necessarily a 1 if it is normal and a 0 if it is subnormal. Since normality can easily be encoded in the exponent field of the representation, the leftmost bit of the significand does not need to be stored. This is the 'hidden bit' or 'implicit bit' convention of IEEE-754. This saving of one bit is especially precious in small formats.

The IEEE-754 Standard for Floating-Point Arithmetic (IEEE 2019) specifies several binary and decimal formats. Among these, of special importance are the *interchange formats*, whose encodings are fully specified as bit strings, which allows for lossless data interchange between platforms. The parameters of the binary interchange formats are given in Table 2.1, along with those of the bfloat16 format (Intel 2018).

The binary32, binary64 and binary128 formats are named *basic formats* in the Standard. IEEE-754 also recommends, for the widest available basic format, the support of an *extended format* with a wider precision and wider range (IEEE 2019, Table 3.7). The underlying idea is that the availability of the extended format greatly facilitates the implementation of functions for the corresponding basic format that are both very accurate and free of spurious underflows or overflows (see Section 2.2). In practice, the extended format that our readers might well encounter is the Intel *double-extended format* ($p = 64$, $e_{\max} = 16383$, $e_{\min} = -16382$), used in the x86 instruction set.

Let us say a few words about the very small floating-point formats. Among the binary formats specified by IEEE-754 since its 2008 version, only binary32, binary64 and binary128 are called *basic formats*, which means that for the designers of the Standard, only these ones were intended to be used for arithmetic. The binary16 format was meant to be used for *storage* only. However, it became evident in the years thereafter that 16-bit floating-point arithmetic was useful for the huge amount of computation required when training neural networks. Furthermore, for these applications the small exponent range of binary16 was a clear penalty. For example, experiments performed at Google (Wang and Kanwar 2019) suggest that such training computations are more sensitive to the exponent range than to the

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Table 2.2. Dynamic range of several FP formats: α is the smallest positive FP number, $2^{e_{\min}}$ is the smallest positive normal FP number and $\mathbf{\Omega}$ is the largest finite FP number. Beware: in bfloat16 arithmetic, subnormal numbers are not necessarily supported.

Format	$\alpha = 2^{e_{\min}-p+1}$	$2^{e_{\min}}$	$\mathbf{\Omega} = 2^{e_{\max}}(2 - 2^{-p+1})$
binary16	$5.960 \cdot 10^{-8}$	$6.104 \cdot 10^{-5}$	$6.550 \cdot 10^{+4}$
bfloat16	$9.184 \cdot 10^{-41}$	$1.175 \cdot 10^{-38}$	$3.390 \cdot 10^{+38}$
binary32	$1.401 \cdot 10^{-45}$	$1.175 \cdot 10^{-38}$	$3.403 \cdot 10^{+38}$
binary64	$4.941 \cdot 10^{-324}$	$2.225 \cdot 10^{-308}$	$1.798 \cdot 10^{+308}$
Intel 'double-extended'	$3.645 \cdot 10^{-4951}$	$3.362 \cdot 10^{-4932}$	$1.190 \cdot 10^{+4932}$
binary128	$6.475 \cdot 10^{-4966}$	$3.362 \cdot 10^{-4932}$	$1.190 \cdot 10^{+4932}$

precision. This motivated the introduction of the bfloat16 (or BF16) format (Intel 2018, Henry, Tang and Heinecke 2019, Osorio *et al.* 2022) and of the DLFloat (or DLFLT-16) format (Agrawal *et al.* 2019, Lichtenau *et al.* 2022). Smaller 8-bit or 9-bit floating-point formats (with 2-bit or 3-bit significands) (Chung *et al.* 2018), or even 4-bit formats (Sun *et al.* 2020), have been suggested for AI applications. In particular, 8-bit floating-point formats recently received specific attention from several major companies (Noune *et al.* 2022, Micikevicius *et al.* 2022). At the time of writing, a working group is being set up to prepare a standard on arithmetic formats for machine learning.[3] For experimenting with very low precisions even if they are not yet available in hardware, several simulation strategies and tools have been designed (Lefèvre 2013, Rump 2017, Higham and Pranesh 2019, Fasi and Mikaitis 2020).

To help the reader get an intuition about these numerous formats, Table 2.2 presents the dynamic range of the most supported floating-point formats.

2.1. Rounding and standard model

In general, the sum, product or quotient of two elements of $\mathbb{F}$ is not an element of $\mathbb{F}$: it must be *rounded*. With some of the early computers, this just meant that the returned value was 'not too far' from the exact result. One of the most fruitful ideas popularized by IEEE-754 is the notion of *correct rounding*, although the term was already mentioned by Wilkinson (1960). One chooses a *rounding function* $\circ$ from $\overline{\mathbb{R}}$ to $\overline{\mathbb{F}}$ (see below), and each time an arithmetic operation $a\Delta b$ is called, what is returned is $\circ(a\Delta b)$. If the computer implementation of Δ satisfies this requirement, Δ is said to be *correctly rounded*. IEEE-754 requires correct rounding of the four

[3] See https://sagroups.ieee.org/p3109wgpublic/.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

arithmetic operations, the square root and the *fused multiply-add (FMA)* instruction, defined for $a, b, c \in \overline{\mathbb{F}}$ as

$$\mathrm{FMA}(a, b, c) = \circ(ab + c). \tag{2.4}$$

That instruction first appeared in 1990 in the IBM POWER instruction set (Cocke and Markstein 1990), and was incorporated into the 2008 version of IEEE-754. Besides making the evaluation of polynomials and the computation of inner products faster and, in general, more accurate, it greatly facilitates the software implementation of correctly rounded division and square root (Markstein 1990, Cornea-Hasegan *et al.* 1999), in part thanks to Properties 2.13 and 2.14 below. In some cases it also makes possible correctly rounded multiplication by real constants (Brisebarre and Muller 2008). The IEEE-754 Standard also recommends (but does not require[4]) correct rounding of a small set of algebraic and transcendental functions. In some instruction sets, e.g. Intel AVX512F (Anderson, Zhang and Cornea 2018), the rounding function can be encoded in the opcode of the floating-point instruction.

Let us now give a more formal definition of a rounding function, inspired by Kulisch (1971).

Definition 2.3. A function $\circ$ from $\overline{\mathbb{R}}$ to $\overline{\mathbb{F}}$ is a *rounding function* if

- it is *monotone:* for all $a, b \in \overline{\mathbb{R}}$, $a \leq b \Rightarrow \circ(a) \leq \circ(b)$;
- for all $a \in \overline{\mathbb{F}}$, $\circ(a) = a$.

A function $\circ$ that satisfies Definition 2.3 is such that for any real number t, if $t \notin \mathbb{F}$ then $\circ(t)$ is one of the two elements of $\overline{\mathbb{F}}$ that surround t (i.e. it is a special case of what we call a *faithful rounding*; see below). In practice we seldom use 'general' rounding functions such as the ones specified by Definition 2.3, because they lack useful properties such as simple relations between $\circ(t)$ and $\circ(-t)$, or between $\circ(t)$ and $\circ(2^k t)$ for some integer k, etc.

The classical rounding functions that are of practical interest in floating-point arithmetic are as follows.

- *Directed* rounding functions.
  - Round towards $-\infty$: $\mathrm{RD}(t)$ is the largest element of $\overline{\mathbb{F}}$ less than or equal to t.
  - Round towards $+\infty$: $\mathrm{RU}(t)$ is the smallest element of $\overline{\mathbb{F}}$ greater than or equal to t.
  - Round towards zero: $\mathrm{RZ}(t) = \mathrm{RD}(t)$ if $t \geq 0$ and $\mathrm{RU}(t)$ otherwise.

[4] This requirement was thought complicated because of the Table Maker's Dilemma. See Section 3.3.1.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

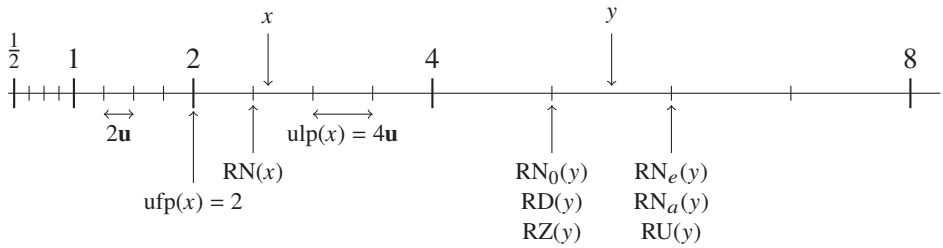

Figure 2.3. The FP numbers between $1/2$ and 8 in the toy system $\beta = 2$, $p = 3$ (i.e. $\mathbf{u} = 1/8$).

- *Round-to-nearest* functions. $\mathrm{RN}(t)$ is the element of $\overline{\mathbb{F}}$ nearest to t, with the following possible *tie-breaking* choices if t is halfway between two consecutive finite FP numbers; in the following, such halfway numbers are called *midpoints*.
  - Round-to-nearest, ties-to-even: $\mathrm{RN}_e(t)$ is the one of these two FP numbers whose integral significand is even.
  - Round-to-nearest, ties-to-away: $\mathrm{RN}_a(t)$ is the one with largest magnitude.
  - Round-to-nearest, ties-to-zero: $\mathrm{RN}_0(t)$ is the one with smallest magnitude.

They are called *rounding direction attributes* in the IEEE-754 Standard. See also Figure 2.3 for an illustration of the rounding functions.

To that list we should add a more complex rounding 'function' that does not follow the requirements of Definition 2.3 but is slowly gaining importance: *stochastic rounding* (SR). If $x \in \mathbb{F}$, then $\mathrm{SR}(x) = x$, otherwise SR maps the real number x to $\mathrm{RD}(x)$ with probability $\mathcal{P}(x)$ and to $\mathrm{RU}(x)$ with probability $1 - \mathcal{P}(x)$. Two choices for $\mathcal{P}(x)$ have been considered in the literature. The first is a constant function (in general equal to $1/2$), and the second is

$$\mathcal{P}(x) = \frac{\mathrm{RU}(x) - x}{\mathrm{RU}(x) - \mathrm{RD}(x)}.$$

Stochastic rounding can be traced back to the first years of electronic computing (Barnes, Cooke-Yarborough and Thomas 1951, Forsythe 1959). It is not deterministic (which makes calculations difficult to reproduce) and it is significantly more complex to implement than the other rounding functions, but (especially when we consider the second choice for $\mathcal{P}$) it has many interesting properties. More precisely, it prevents some correlation of errors, avoids the phenomenon of *stagnation*,[5] and gives much better error bounds on common algorithms such as sums or inner products. This active topic is covered by a recent survey by Croci *et al.*

[5] An example of stagnation is the computation of $s_n = \sum_{i=1}^{n}(1/i)$ in the obvious way in rounded-to-nearest arithmetic. We all know that the *exact* value of s_n goes to $+\infty$ as $n \to +\infty$; however,

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

(2022). Stochastic rounding has also been suggested to obtain probabilistic estimates of the round-off error of a calculation (La Porte and Vignes 1974, Parker, Pierce and Eggert 2000).

In the IEEE-754 Standard, for the round-to-nearest, ties-to-even or ties-to-away rounding functions, the rules presented above are slightly modified for huge arguments. If $|t|$ is larger than or equal to $2^{e_{\max}+1} - 2^{e_{\max}-p}$, then ∞ (with the same sign as t) is returned. Equivalently this can be viewed as if we first applied the above rules with an unbounded above exponent range before replacing all results of magnitude larger than $\mathbf{\Omega}$ by $\pm\infty$. The designers of the Standard correctly estimated that if the exact result of a calculation is huge (i.e. of magnitude much larger than $\mathbf{\Omega}$), returning an infinity makes more sense than returning $\pm\mathbf{\Omega}$.

The default rounding function in the IEEE-754 Standard is $\mathrm{RN}_e(t)$. The 'ties-to-even' tie-breaking rule may look strange, but it has the double advantage of being unbiased (which may matter in long summations) and straightforwardly implementable.[6] The directed roundings RD, RU and RZ are useful for getting certain upper or lower bounds on a result, and more generally for implementing *interval arithmetic* (see Section 4.1). RN_a is used in accounting; the IEEE Standard requires its availability in base-10 arithmetic only. RN_0 is used with the 'augmented operations' (see Section 4.2.3) and is needed particularly in some reproducible summation algorithms (Riedy and Demmel 2018). In the following, RN means any of RN_e, RN_a or RN_0 unless stated otherwise, and $\mathrm{R}_{\mathrm{D}}^{\mathrm{U}}$ is RN, RZ, RU or RD.

Although it is not a rounding in the sense expressed by Definition 2.3, an important notion is that of *faithful rounding*. We will say that $X \in \mathbb{F}$ is a *faithful rounding* of $x \in [-\mathbf{\Omega}, +\mathbf{\Omega}]$ if X is either $\mathrm{RD}(x)$ or $\mathrm{RU}(x)$. Note that any of the previously defined roundings is a faithful rounding by definition.

Now, let us define the following parameters.

Definition 2.4 (unit round-off, ulp and ufp).

- The *unit round-off* is the number

$$\mathbf{u} = 2^{-p}.$$

Let $t \in \mathbb{R}$, $|t| \leq \mathbf{\Omega}$.

- The *unit in the last place* of $t \neq 0$ is the number

$$\mathrm{ulp}(t) = 2^{\max\{e_{\min}, \lfloor \log_2 |t| \rfloor\} - p + 1}.$$

- The *unit in the first place* of t, for $t \neq 0$, is the number

$$\mathrm{ufp}(t) = 2^{\lfloor \log_2 |t| \rfloor}.$$

the *computed* value of s_n remains constant when n is larger than some threshold n_0 that depends on the precision p.

[6] Ties-to-away would be slightly simpler to implement, but it has the disadvantage of being biased in the relatively frequent case where one performs summations of numbers that all have the same sign; an example is the numerical integration of a function in a domain where its sign is constant.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

By a simple extension by continuity we can also define $\mathrm{ulp}(0) = 2^{e_{\min}-p+1}$ and $\mathrm{ufp}(0) = 0$. In the normal domain, the ulp and ufp functions can be interchanged (with the adequate scaling factor) in all formulas, since $\mathrm{ufp}(t) = 2^{p-1}\,\mathrm{ulp}(t)$. But in the subnormal domain, they behave very differently. Indeed, all non-zero reals in the subnormal domain have the same ulp, equal to α, while their ufp can take $p-1$ distinct values.

The unit round-off was already appearing in a paper by Wilkinson (1960), and that notion is now widespread in rounding error analysis of numerical algorithms. The acronym 'ulp' for *unit in the last place* was coined by Kahan. Several slightly different definitions of the ulp function appear in the literature (Goldberg 1991, Harrison 1999, Kahan 2004*a*, Overton 2001), they differ near powers of 2. The ufp function seems to have appeared for the first time in a paper by Rump, Ogita and Oishi (2008).

Let $t \in \mathbb{R}$, $|t| \le \mathbf{\Omega}$. Then $\mathrm{RN}(t)$ is a multiple of $\mathrm{ulp}(t)$. The number $\mathrm{ulp}(t)$ is the distance between the two consecutive FP numbers a and b that satisfy $a \le |t| < b$. It follows that $|t - \mathrm{RN}(t)| \le \frac{1}{2}\mathrm{ulp}(t)$. Therefore:

- if t is in the normal domain (i.e. $2^{e_{\min}} \le |t| \le \mathbf{\Omega}$) then $\mathrm{ulp}(t) = 2\mathbf{u} \cdot \mathrm{ufp}(t)$, so
$$|t - \mathrm{RN}(t)| \le \mathbf{u} \cdot \mathrm{ufp}(t) \le \mathbf{u} \cdot |t|$$
and
$$|t - \mathrm{RN}(t)| \le \mathbf{u} \cdot \mathrm{ufp}(t) \le \mathbf{u} \cdot |\mathrm{RN}(t)|;$$
- if t is zero or a subnormal number (i.e. $|t| < 2^{e_{\min}}$) then
$$|t - \mathrm{RN}(t)| \le 2^{e_{\min}-p}.$$

From this, we deduce that, provided the arithmetic operations are correctly rounded, if $a \in \mathbb{F}$, $b \in \mathbb{F}$, $\mathrm{op} \in \{+, -, \times, /\}$ and $2^{e_{\min}} \le |a \,\mathrm{op}\, b| \le \mathbf{\Omega}$, then there exist real numbers ε_1 and ε_2 such that

$$\mathrm{RN}(a \,\mathrm{op}\, b) = (a \,\mathrm{op}\, b)(1+\varepsilon_1), \qquad |\varepsilon_1| \le \mathbf{u}, \tag{2.5a}$$
$$= (a \,\mathrm{op}\, b)/(1+\varepsilon_2), \qquad |\varepsilon_2| \le \mathbf{u}. \tag{2.5b}$$

Identity (2.5a), already used by Wilkinson (1960) for analysing some problems of linear algebra, is sometimes called the *standard model* of floating-point arithmetic. Most rounding error analysis is based on it (Higham 2002). Several modifications of this model are possible.

- First, if the result of an operation is subnormal, one cannot in general guarantee a *relative error bound* such as (2.5a) or (2.5b), but the *absolute error* is bounded by $\alpha/2 = 2^{e_{\min}-p}$.
- Second, since all FP numbers are multiple of α, the exact sum (or difference) of two elements of $\mathbb{F}$ is a multiple of α too. It follows from Property 2.2 that if such a sum has absolute value less than or equal to $2^p\alpha = 2^{e_{\min}+1}$, it is an

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

element of $\mathbb{F}$. In particular, *when the result of a floating-point addition or subtraction is in the subnormal range, that addition or subtraction is exact*. That property is sometimes called Hauser's theorem (Hauser 1996).

- Finally, the bound $\mathbf{u}$ on $|\varepsilon_1|$ in (2.5a) can be replaced by the slightly smaller $\mathbf{u}/(1+\mathbf{u})$, which is attained for example at the midpoint $1+\mathbf{u}$; see Knuth (1998, p. 232). This may seem a very small change, but it suffices to get simpler values for the error bounds on the evaluation of arithmetic expressions such as sums, dot products and Euclidean norms (Rump 2019).

From all this, the following hold.

- When $|a \text{ op } b| < 2^{e_{\min}}$ and op $\in \{+,-\}$, (2.5a) and (2.5b) still hold (indeed, with $\varepsilon_1 = \varepsilon_2 = 0$).
- If op $\in \{\times, /\}$ and underflow may occur, then (2.5a) and (2.5b) are modified as follows: there exist ε_1 and ε_2, with $|\varepsilon_1| \le \mathbf{u}/(1+\mathbf{u}) < \mathbf{u}$ and $|\varepsilon_2| \le \mathbf{u}$ and η_1 and η_2, with $|\eta_1|, |\eta_2| \le \alpha/2$, such that

$$\mathrm{RN}(a \text{ op } b) = (a \text{ op } b)(1+\varepsilon_1) + \eta_1 \tag{2.6a}$$
$$= (a \text{ op } b)/(1+\varepsilon_2) + \eta_2, \tag{2.6b}$$

  and $\varepsilon_1\eta_1 = \varepsilon_2\eta_2 = 0$.

Finally, for some specific arithmetic operations, the relative error bounds $\mathbf{u}$ or $\mathbf{u}/(1+\mathbf{u})$ can be slightly improved. More precisely, Jeannerod and Rump (2018) prove the optimality of the bound $\mathbf{u}/(1+\mathbf{u})$ for addition, subtraction and multiplication. They also show that in binary FP arithmetic the optimal bound is $\mathbf{u} - \mathbf{u}^2$ for division and $1 - 1/\sqrt{1+2\mathbf{u}}$ for square root.

The standard model can also be used with directed (and faithful) roundings. Indeed, if RN is replaced by RU, RD or RZ, (2.5a), (2.5b), (2.6a) and (2.6b) still apply, with the bounds on $|\varepsilon_1|$ and $|\varepsilon_2|$ now equal to $2\mathbf{u}$ and the bounds on $|\eta_1|$ and $|\eta_2|$ replaced by α.

Table 2.3 summarizes the notation introduced in this section, and Figure 2.3 illustrates parts of it in the case of the toy system $\beta = 2$ and $p = 3$.

One should not underestimate the importance of the standard model. Indeed, the fact that the individual arithmetic operations have a bounded relative error has been at the heart of all of numerical error analysis since the pioneering work of Wilkinson (1960). It is very unwise to perform critical calculations using a computer number system that does not obey the standard model.[7]

[7] At least in 'large' formats. With 'small' formats, and when the calculation depends on a few input values only, the analysis is sometimes straightforward. For instance, the best way to check an implementation of a 32-bit function of one variable or a 16-bit function of two variables is to test all possible input values.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Table 2.3. Notation for the most important FP parameters and functions.

Notation	Numerical value	Explanation
$\mathbf{\Omega}$	$2^{e_{\max}} \cdot (2 - 2^{-p+1})$	largest finite FP number
α	$2^{e_{\min}-p+1}$	smallest positive FP number
$2^{e_{\min}}$	$2^{e_{\min}}$	smallest positive normal FP number
$\mathbf{u}$	2^{-p}	unit round-off
$\mathrm{ulp}(x)$ $(x \in \mathbb{R}, x \neq 0)$	$2^{\max\{\lfloor \log_2 \lvert x \rvert \rfloor, e_{\min}\}-p+1}$	unit in the last place
$\mathrm{ufp}(x)$ $(x \in \mathbb{R}, x \neq 0)$	$2^{\lfloor \log_2 \lvert x \rvert \rfloor}$	unit in the first place
$\mathrm{RD}(x)$	$\max\{X \in \overline{\mathbb{F}}, X \leq x\}$	round towards $-\infty$
$\mathrm{RU}(x)$	$\min\{X \in \overline{\mathbb{F}}, X \geq x\}$	round towards $+\infty$
$\mathrm{RZ}(x)$	$\mathrm{RU}(x)$ if $x < 0$, $\mathrm{RD}(x)$ otherwise	round towards 0
$\mathrm{RN}_e(x)$	$\mathrm{sign}(x) \cdot \infty$ if $\lvert x \rvert \geq 2^{e_{\max}+1} - 2^{e_{\max}-p}$, otherwise $X \in \mathbb{F}$ such that: (i) for all $Y \in \mathbb{F}$, $\lvert Y - x \rvert \geq \lvert X - x \rvert$, (ii) if there is $Y \in \mathbb{F}, Y \neq X$ such that $\lvert Y - x \rvert = \lvert X - x \rvert$, then X has an even integral significand	round-to-nearest, ties-to-even
$\mathrm{RN}(x)$	'generic' round-to-nearest (arbitrary tie-breaking rule)	'generic' round-to-nearest
$\mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(x)$	any of $\mathrm{RD}(x)$, $\mathrm{RU}(x)$, $\mathrm{RZ}(x)$, $\mathrm{RN}(x)$	'generic' rounding function
$\mathrm{FMA}(a, b, c)$	$\mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(a \cdot b + c)$	fused multiply-add

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

2.2. *Underflow and overflow*

We will say that an operation *underflows* if its result is of absolute value less than $2^{e_{\min}}$ *and* inexact. This is the condition for raising the *underflow flag* in the default exception handling of the IEEE-754 Standard.[8] At first glance that choice may look strange, but it makes sense. Indeed, since the underflow flag is a warning that the computed result may be less accurate than expected (because we are outside of the domain of validity of the standard model), there is no point in giving that warning when the result is exact.

An operation *overflows* if the rounded result we would obtain if the exponent range were unbounded has a magnitude strictly larger than $\mathbf{\Omega}$. For instance, with the round-to-nearest, ties-to-even or ties-to-away rounding function, an operation overflows when the exact result has absolute value larger than or equal to $2^{e_{\max}+1} - 2^{e_{\max}-p}$. Still with the RN function, when an operation overflows, the returned result is $\pm\infty$ with the correct sign. With the other rounding functions, it will be $\pm\mathbf{\Omega}$ or $\pm\infty$, according to the definition of these rounding functions given in Section 2.3.

As the IEEE-754 Standard specifies two infinities, it also specifies two zeros, +0 and −0. The idea of having signed zeros may sound strange, but it greatly facilitates the handling of branch cuts when implementing complex functions (Kahan 1987).

One of the major difficulties when designing function software that is supposed to work in all cases is to avoid *spurious underflows or overflows*. These are underflows or overflows that occur in an intermediate operation, resulting in inaccurate or infinite results, whereas the exact value of the function is in the normal domain. A simple example is the 'naive' calculation of $\texttt{hypot}(x, y) = \sqrt{x^2 + y^2}$. In binary64 arithmetic, with $x = 1.5 \times 2^{511}$ and $y = 2^{512}$ we obtain an infinite result because the computation of y^2 overflows, whereas the exact result is $5 \cdot 2^{510}$, which is much smaller than $\mathbf{\Omega}$. Similarly, with $x = y = (45/64) \times 2^{-537}$, the computed result is 0, whereas the exact result is around 1.9887×2^{-538}, i.e. much larger than $2^{e_{\min}}$. To overcome this difficulty, two strategies have been suggested over the years.

(1) Start with the simple, straightforward and fast algorithm. Check with the IEEE-754 flags if an exception has occurred (Hull, Fairgrieve and Tang 1994). If this is the case (which hopefully should not happen frequently), redo the calculation with an alternative (and in general significantly slower) algorithm.

(2) *Scale* the operands, i.e. multiply them by a well-chosen value (Beebe 2017, §8.2). In the case of function $\texttt{hypot}(x, y)$ a typical scaling factor is $1/\max(|x|, |y|)$, or even better, 2^{-K} where $K = \lfloor \log_2(\max(|x|, |y|)) \rfloor$.

[8] The Standard does not specify if the comparison to $2^{e_{\min}}$ must be done after or before rounding. For instance, the rounded-to-nearest computation of $\sin(2^{e_{\min}})$ would underflow in the second case but not in the first.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

2.3. *Not a Number (NaN)*

With some early floating-point systems the occurrence of a 'forbidden' or 'undefined' operation such as $\sqrt{-7}$ or ∞/∞ (with any signs) would halt the computation. In general this is a poor decision, as the result of a complex calculation is rarely just *one* number. Imagine a long computing process that would have generated thousands of meaningful outputs and that is stopped because an error occurred during the calculation of just one, possibly unimportant, parameter.

The IEEE-754 Standard introduced a special floating-point value, NaN, for representing the result of such operations. NaN stands for 'Not a Number' or 'Not any Number' (Kahan 1997). More precisely, the Standard distinguishes two different kinds of NaNs: *signalling* NaNs (sNaN) and *quiet* NaNs (qNaN). Signalling NaNs do not appear as the result of an arithmetic operation (the Standard suggests that they can be used to represent uninitialized inputs), and the reader will seldom encounter them in practice.

On the contrary, under the default exception handling of the Standard, a quiet NaN is delivered when performing one of the following arithmetic operations: $\sqrt{\text{negative number}}$, $0 \times \infty$, $0/0$, ∞/∞ (with any signs), $\infty - \infty$ (when both signs are equal), Remainder(anything, 0) and Remainder($\pm\infty$, anything). Moreover, when at least one of the inputs of an arithmetic operation is a quiet NaN, a quiet NaN is returned.

There is an exception, though. If a multivariate function always delivers the same value once some of its inputs are set, then that same value is returned even if the remaining inputs are a quiet NaN. For instance, the Standard suggests that an implementation of $\texttt{hypot}(x, y) = \sqrt{x^2 + y^2}$ should satisfy $\texttt{hypot}(\pm\infty, \text{qNaN}) = +\infty$.

2.4. *Beyond the standard model*

The standard model (2.5a) is simple and powerful, and, interestingly, floating-point arithmetic is not the only kind of arithmetic that satisfies it. Logarithmic number systems (Swartzlander and Alexpoulos 1975) also guarantee a bounded relative error for rounding functions, so an error analysis that only uses the standard model also applies to computations using these systems.

However, some useful algorithms and properties of floating-point arithmetic require more than just the standard model to be analysed and proved. The simplest (and maybe the most useful) example is a simple subtraction of two floating-point numbers that are close enough to each other.

Theorem 2.5 (Sterbenz 1974). Let $a, b \in \mathbb{F}$. If $a/2 \leq b \leq 2a$ then $a - b \in \mathbb{F}$. This implies that the subtraction will be computed exactly in FP arithmetic, whatever the chosen rounding function.

Clearly the standard model does not suffice to deduce Theorem 2.5, as logarithmic number systems do not satisfy that property. We will not give many proofs in this

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

survey, but, following Rump *et al.* (2008), let us quickly prove Theorem 2.5 to give an idea of the kind of reasoning that frequently appears in FP arithmetic. Since a and b play a symmetric role we can assume $a \geq b$. The two FP numbers a and b are multiples of $\mathrm{ulp}(b)$, so $a - b$ is a (positive) multiple of $\mathrm{ulp}(b)$. From the hypothesis $a/2 \leq b$, we deduce $a - b \leq b$. Since $b \in \mathbb{F}$, $b < 2^p\,\mathrm{ulp}(b)$. Let $k = \log_2(\mathrm{ulp}(b)) \geq e_{\min} - p + 1$. Since $a - b$ is a multiple of 2^k less than 2^{k+p}, it is an FP number by Property 2.2.

Two important things must be said concerning Theorem 2.5.

- It remains true in base-β FP arithmetic. Beware: the '2' that appears in the expressions '$a/2$' and '$2a$' in Theorem 2.5 must remain a '2', as the theorem no longer holds if it is replaced by β.
- There is no contradiction between Theorem 2.5 and the traditional (and rightful!) rule that says subtracting two numbers very close together is dangerous. Indeed, $a - b$ is computed exactly, but if a approximates a real number a^* and b approximates a real number b^*, even if these approximations have a very small relative error, $a - b$ may be a very loose approximation to $a^* - b^*$, as the relative errors between a and a^* and b and b^* may be magnified.

Another useful example that cannot be analysed just by using the standard model is the Fast2Sum algorithm (Møller 1965, Dekker 1971), i.e. Algorithm 1. If the floating-point exponents e_a and e_b of a and b are such that $e_a \geq e_b$, and if the first operation does not overflow, then t is the error of the floating-point addition $\mathrm{RN}(a + b)$, i.e. $s + t = a + b$, as shown by Knuth (1998) and later formally proved by Daumas, Rideau and Théry (2001); see also the work by Lange and Oishi (2020) for further extensions and applications. If we make the stronger assumption $|a| \geq |b|$, the proof becomes a straightforward consequence of the Sterbenz theorem (Theorem 2.5). We will see some other similar algorithms in Section 4.2.

Algorithm 1 Fast2Sum(a, b)

$s \leftarrow \mathrm{RN}(a + b)$
$z \leftarrow \mathrm{RN}(s - a)$
$t \leftarrow \mathrm{RN}(b - z)$
return (s, t)

2.5. *Errors in ulps vs relative errors*

The errors of numerical programs are in general expressed either in terms of relative error or in ulps. The two ways of expressing the errors do not convey the same amount of information. Let us now examine that difference. First, note that in general when we talk about the error in ulps of a program, we implicitly mean 'error in ulps *of the exact result*', as expressing an error in ulps of the *computed result* might result in dubious conclusions. Assume for instance that the exact result

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

is the real $x = 1 + \mathbf{u}$ and consider two (quite poor) computed floating-point results: $a = 2 - 2\mathbf{u}$ and $b = 2 + 4\mathbf{u}$. Since $x < a < b$, it would make no sense to consider that b is a better approximation to x than a. And yet x is within $(2^{p-1} - 3/2)\,\mathrm{ulp}(a)$ from a, and within $(2^{p-2} + 3/4)\,\mathrm{ulp}(b)$ from b.

Suppose a program is written to approximate a function $f : \mathbb{R} \to \mathbb{R}$. For the sake of simplicity, assume that the rounding function is RN. A 'perfect' program, when called with input $x \in \mathbb{F}$, will return $\mathrm{RN}(f(x))$. The error in ulps of this program[9] will be bounded by 0.5 ulp, and (assuming that the output is in the normal range), the relative error will be bounded by $\mathbf{u}$. The converse is almost true for the error in ulps. Indeed, if a program has an error bounded by 0.5 ulp then, when called with an input $x \in \mathbb{F}$, it always returns one of the FP numbers nearest $f(x)$. But the result is not necessarily equal to $\mathrm{RN}(f(x))$, in particular if $f(x)$ is a midpoint and the error is exactly 0.5 ulp.

On the other hand, a program may have a relative error bounded by $\mathbf{u}$ and deliver results that are quite far from being correctly rounded. For example, if the exact result is $2 - 2\mathbf{u} + 5\mathbf{u}^2$ and the computed result is 2, then the relative error is about $\mathbf{u} - \frac{3}{2}\mathbf{u}^2$ (and thus less than $\mathbf{u}$) and, at the same time, correct rounding is not achieved, since the exact result is much closer to the FP number $2 - 2\mathbf{u}$ than to the computed result 2.

The reason for this difference is that the bound (2.5a) is tight only when t is just above a power of 2. The local maximum relative error wobbles between $\mathbf{u}/2$ and $\mathbf{u}$, as illustrated in Figure 2.4. With a base-β floating-point arithmetic, the situation would be worse, as the 'wobbling factor' would be β instead of 2.[10]

From that point of view, an error expressed in ulps conveys more information than a relative error. This explains why the error of 'atomic' calculations (e.g. the elementary functions: sine, cosine, logarithm, etc.) are in general expressed in ulps; see for instance a very useful analysis of the quality of current elementary function software by Innocente and Zimmermann (2022). On the contrary, manipulating ulps in large algorithms is almost infeasible, whereas relative errors are rather easily manipulated. For instance, one easily deduces a relative error bound on $f \times g$ from the relative error bounds on f and g, and the relative error bound $\mathbf{u}$ of the multiplication. As a consequence, relative errors are favoured for larger computations.

Finally, in the normal domain, one can always deduce an error bound in ulps from a relative error bound, and conversely, using Property 2.6 below. Note that we may lose information during these conversions. It shows when using Property 2.6 twice, to convert an error in ulps to relative error, and then back to error in ulps, we end up with twice the initial bound.

[9] As mentioned above, unless stated otherwise, 'error in ulps' means 'error in ulps *of the exact result*'.

[10] This wobbling factor and the implicit bit convention are the main reasons why the best base for numerical computing is 2.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

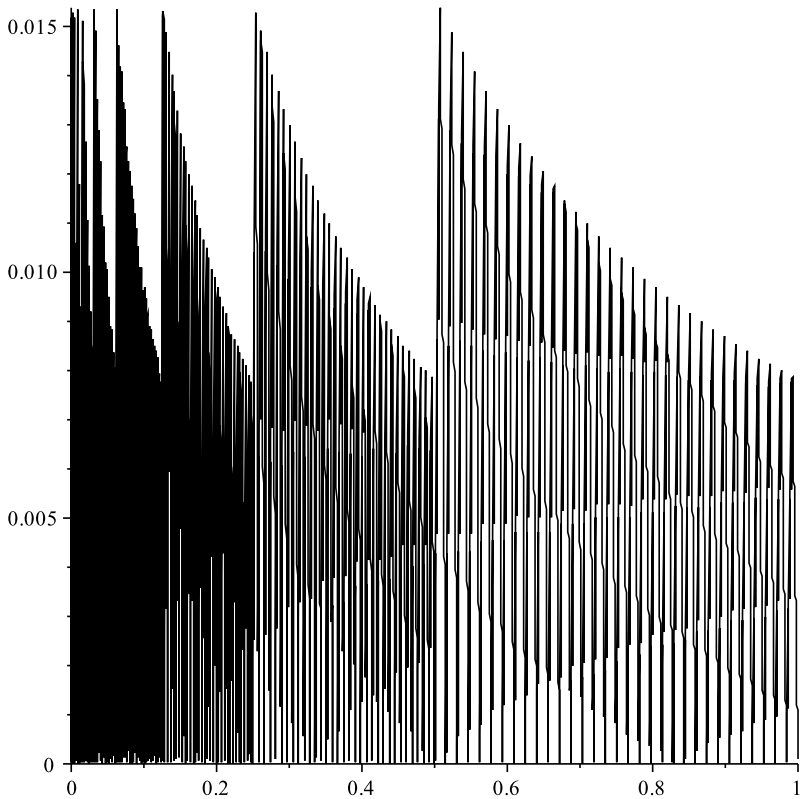

Figure 2.4. The relative error due to rounding x to nearest, i.e. $|x - \mathrm{RN}(x)|/x$, for x between 0 and 1, assuming an FP system of base 2 and precision 6. The local maximum value of the relative error wobbles between $\mathbf{u}/2$ and $\mathbf{u}$.

Property 2.6 (link between error in ulps and relative errors). If $x \in \mathbb{R}$ and $\hat{x} \in \mathbb{F}$ are in the normal domain, then

- $|x - \hat{x}| \leq \varepsilon \cdot \mathrm{ulp}(x) \Rightarrow \left|\dfrac{x - \hat{x}}{x}\right| \leq 2\mathbf{u}\varepsilon = 2^{1-p}\varepsilon;$
- $\left|\dfrac{x - \hat{x}}{x}\right| \leq \varepsilon \Rightarrow |x - \hat{x}| \leq \dfrac{\varepsilon}{\mathbf{u}}\,\mathrm{ulp}(x) = 2^{p}\varepsilon \cdot \mathrm{ulp}(x).$

If $x \in \mathbb{R}$ lies in the subnormal domain, i.e. $|x| < 2^{e_{\min}}$, the situation is quite different, as $\mathrm{ulp}(x)$ is now the constant $2^{e_{\min}-p+1}$. Even if an error bound in ulps is small, the corresponding relative error can be very large. For example, if $\hat{x} = 0$ and $x = \varepsilon \cdot 2^{e_{\min}-p+1}$ with $\varepsilon > 0$, then the error in ulps is ε while the relative error is 1. On the other hand, if $|(x - \hat{x})/x| \leq \varepsilon$, then $|x - \hat{x}| \leq \varepsilon \cdot 2^{e_{\min}} = 2^{p-1}\varepsilon \cdot \mathrm{ulp}(x)$.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

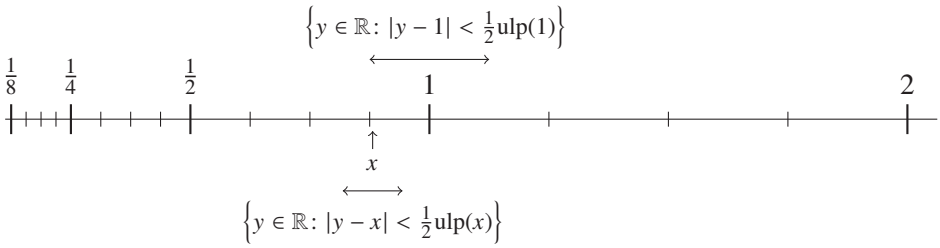

Figure 2.5. Correct rounding and ulp in the toy system $\beta = 2$, $p = 3$, given $x = 1 - \mathbf{u} + \mathbf{u}^2$.

2.6. *Errors in ulps and correct or faithful rounding*

It is common to read that 'correct rounding' is equivalent to 'error less than half an ulp', and that 'faithful rounding' is equivalent to 'error less than one ulp'. This is almost true but not entirely. First, ulp is not a constant but a function, and it must be clearly stated whether we consider the ulp of the exact result or the ulp of the computed result (as mentioned above, the latter choice is a dubious one). Second, the ulp function is discontinuous at the powers of 2. Properties 2.7 and 2.8 clarify the slight differences.

Property 2.7 (links between error in ulps and correct rounding). Assume that $x \in [-\mathbf{\Omega}, +\mathbf{\Omega}]$ and $\hat{x} \in \mathbb{F}$. We have:

- if $\hat{x} = \mathrm{RN}(x)$ then $|x - \hat{x}| \le \frac{1}{2}\mathrm{ulp}(x) \le \frac{1}{2}\mathrm{ulp}(\hat{x})$;
- if $|x - \hat{x}| < \frac{1}{2}\mathrm{ulp}(x)$ then $\hat{x} = \mathrm{RN}(x)$.

Beware: in rather infrequent cases, we may have $|x - \hat{x}| < \frac{1}{2}\mathrm{ulp}(\hat{x})$ and $\hat{x} \neq \mathrm{RN}(x)$. An example is $x = 1 - \mathbf{u} + \mathbf{u}^2$ as shown in Figure 2.5, for which we have $|x - 1| < \frac{1}{2}\mathrm{ulp}(1)$ and yet $\mathrm{RN}(x) = 1 - \mathbf{u} \neq 1$. The situation is worse in bases larger than 2. Indeed, if $\beta = 10$ and $p = 4$, consider the real number $x = 1.0002$. We have[11] $\mathrm{ulp}(x) = 0.001$ and therefore the FP number $\hat{x} = 0.9998$ is within $0.5\,\mathrm{ulp}(x)$ from x, and yet it is not equal to $\mathrm{RN}(x)$. In fact there are even two floating-point numbers between x and $\hat{x}$.

Property 2.8 (links between error in ulps and faithful or directed rounding). Assume that $x \in [-\mathbf{\Omega}, +\mathbf{\Omega}]$ and $\hat{x} \in \mathbb{F}$. If $\hat{x} \in \{\mathrm{RD}(x), \mathrm{RU}(x)\}$ (or equivalently if $\hat{x}$ is a faithful rounding of x), then $|\hat{x} - x| < \mathrm{ulp}(x) \le \mathrm{ulp}(\hat{x})$.

Near powers of 2, the converse is not true. Indeed, if x is any real number in the interval $(1, 1 + \mathbf{u})$ and $\hat{x} = 1 - \mathbf{u}$, then $|x - \hat{x}| < \mathrm{ulp}(x) = 2\mathbf{u}$, whereas $\mathrm{RD}(x) = 1$ and $\mathrm{RU}(x) = 1 + 2\mathbf{u}$.

[11] We have not defined the ulp function in the case of non-binary bases, but in the case considered here it is the distance between the two FP numbers that surround x.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

2.7. Various properties

When analysing numerical programs, we often have to compute bounds iteratively on the possible values of variables.[12] For example, assume we have previously shown that $a \leq B_a$ and $b \leq B_b$, and we want to bound the possible values of a variable s after execution of the program line

```
s = a + b
```

Since $a+b \leq B_a+B_b$, if B_a+B_b is in the normal domain, the standard model tells us that $s \leq (B_a + B_b) \cdot (1 + \mathbf{u})$. However, it is often the case that $B_a + B_b \in \mathbb{F}$. In such a case, we have $s = \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(a + b) \leq \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(B_a + B_b) = B_a + B_b$. The corresponding (straightforward!) property is

Property 2.9. If $\hat{x} \in \mathbb{F}$, then $t \leq \hat{x} \Rightarrow \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(t) \leq \hat{x}$ and $t \geq \hat{x} \Rightarrow \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(t) \geq \hat{x}$.

The following property is frequently used in proofs for bounding rounding errors.

Property 2.10. If $x \in \mathbb{R}$, $|x| \leq 2^k + 2^{k-1-p}$ with $e_{\min} \leq k \leq e_{\max}$, then $|x - \mathrm{RN}(x)| \leq 2^{k-1-p}$.

2.8. Exact correcting terms

Although, as explained before, the sum, product or quotient of two floating-point numbers is not, in general, a floating-point number and hence must be rounded, Dekker (1971) and Pichat (1976) remarked early on that a *correcting term* can very often be represented by a floating-point number. We will see below that such correcting terms can be computed quite easily in floating-point arithmetic (with the same precision), which makes it possible to use them later on in a calculation, to at least partly *compensate* for the rounding error. A systematic study of these correcting terms was done by Bohlender, Walter, Kornerup and Matula (1991), with the assumption that no underflow occurs. Subsequently, this assumption was removed by Boldo and Daumas (2003). We give the main results of that paper below, adapted to the notation of this article.

Property 2.11. Let $a, b \in \mathbb{F}$ and $s = \mathrm{RN}(a + b)$. If the FP addition of a and b does not overflow (i.e. $|s| \leq \mathbf{\Omega}$), then $s - (a + b) \in \mathbb{F}$.

Note that Property 2.11 does not necessarily hold with rounding functions different from RN. For example, if $a = 1$ and $b = \mathbf{u}^3$, then $s := \mathrm{RU}(a + b) = 1 + 2\mathbf{u}$, so that $s - (a + b) = 2\mathbf{u} - \mathbf{u}^3 \notin \mathbb{F}$. (For RD, the same can be observed by simply negating these values of a and b.)

[12] Let us give an example with function $\mathtt{hypot}(x, y) = \sqrt{x^2 + y^2}$ assuming that x and y are normal FP numbers and the rounding function is RN. Without loss of generality, we can assume $1 \leq x < 2$ and $0 \leq y \leq x$. To tightly bound the error in ulps of the addition of x^2 and y^2 we need to show that the *computed* values of x^2 and y^2 are less than 4.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Now, for multiplication such a restriction on the rounding function is not necessary, and we can thus state the next property using the 'generic' rounding function $\mathrm{R}_{\mathrm{D}}^{\mathrm{U}} \in \{\mathrm{RN}, \mathrm{RD}, \mathrm{RU}, \mathrm{RZ}\}$.

Property 2.12. Let $a, b \in \mathbb{F}$ and $\pi = \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(ab)$. If the FP multiplication of a and b does not overflow (i.e. $|\pi| \leq \mathbf{\Omega}$), then $\pi - ab \in \mathbb{F}$ if and only if there exist integers k_a, N_a, k_b and N_b, with $|N_a|, |N_b| \leq 2^p - 1$ and $k_a, k_b \geq e_{\min}$ such that $a = N_a \cdot 2^{k_a - p+1}$, $b = N_b \cdot 2^{k_b - p+1}$ and $k_a + k_b \geq e_{\min} + p - 1$.

In particular (and this will be the most useful application of Property 2.12), if the exponents e_a and e_b of a and b satisfy $e_a + e_b \geq e_{\min} + p - 1$, then $\pi - ab \in \mathbb{F}$, but Property 2.12 is more general.[13]

Let us give two examples. Assume binary32 arithmetic ($p = 24$, $e_{\min} = -126$) and round-to-nearest, ties-to-even. Note that $e_{\min} + p - 1 = -103$.

Let us first choose $a = 16777215 \cdot 2^{-10-23}$ (i.e. $N_a = 16777215$ and $k_a = -10$) and $b = 16777213 \cdot 2^{-95-23}$ (i.e. $N_b = 16777213$ and $k_b = -95$). Observe that $k_a + k_b < e_{\min} + p - 1$; the condition of Property 2.12 is not satisfied. We easily find that $\pi = \mathrm{RN}(ab) = 4194303 \cdot 2^{-125}$, so that $\pi - ab = -3 \cdot 2^{-151}$. This cannot be a floating-point number since it is not a multiple of $\alpha = 2^{-149}$.

If we choose the same a as previously and $b = 8388611 \cdot 2^{-80-23}$ (the condition of Property 2.12 is now satisfied), an elementary calculation shows that $\mathrm{RN}(ab) - ab = -8388605 \cdot 2^{-136} \in \mathbb{F}$.

For division, an exact relationship is obtained by considering not the absolute error itself (whose bit string can be infinitely long) but the associated residual. This fact was already noted by Pichat (1976, p. 43), and a general statement is as follows.

Property 2.13. Let $a, b \in \mathbb{F}$, $b \neq 0$ and $q = \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(a/b)$. If the FP division does not overflow, then $a - bq \in \mathbb{F}$ if and only if there exist integers k_b, N_b, k_q and N_q, with $|N_b|, |N_q| \leq 2^p - 1$ and $k_b, k_q \geq e_{\min}$ such that $b = N_b \cdot 2^{k_b - p+1}$ and $q = N_q \cdot 2^{k_q - p+1}$, and

- $k_b + k_q \geq e_{\min} + p - 1$, and
- $|q| \neq \alpha$ or $\alpha/2 \leq |a/b|$.

Again, $a - bq \in \mathbb{F}$ holds when the second condition is satisfied and the first is replaced by $e_b + e_q \geq e_{\min} + p - 1$, where e_b and e_q are the floating-point exponents of b and q, but Property 2.13 is more general. If the FP division does not underflow (i.e. $|q| \geq 2^{e_{\min}}$) then the second condition is satisfied.

Let us give an example in binary32, round-to-nearest, ties-to-even arithmetic. Consider $a = 8388609 \cdot 2^{-128}$ and $b = 8388611 \cdot 2^{-100-23}$. The number $q = \mathrm{RN}(a/b)$ is equal to $4194303 \cdot 2^{-27} = 4194303 \cdot 2^{-4-23}$. The largest k_b and k_q such that one

[13] This can be useful when the significand of a (or b) has some trailing zeros, which occurs in particular when performing splitting operations; see Section 4.3. In such a case, we know that a has representation $N_a \cdot 2^{k_a - p+1}$ with $|N_a| \leq 2^p - 1$ and $k_a > e_a$, so that the conditions of Property 2.12 may be satisfied even if $e_a + e_b < e_{\min} + p - 1$.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

can write $b = N_b \cdot 2^{k_b - p+1}$ and $q = N_q \cdot 2^{k_q - p+1}$ are -100 and -4 (because 8388611 and 4194303 are odd integers). Therefore $k_b + k_q \leq -104 < e_{\min} + p - 1$ and the condition of Property 2.13 is not satisfied. We easily observe that $a - bq = 3 \cdot 2^{-150}$ is not a floating-point number.

Now, with the same value of b, if $a = 8388609 \cdot 2^{-127}$, then q is twice as large as previously, so that $a - bq = 3 \cdot 2^{-149} = 3 \cdot 2^{e_{\min} - p+1} \in \mathbb{F}$.

Property 2.13 is one of the main motivations for implementing an FMA instruction, as defined in (2.4). Indeed, when $a - bq \in \mathbb{F}$, it is computed exactly with just one FMA. This makes it possible, with some care, to deduce $\mathrm{RN}(a/b)$ from a faithful rounding of a/b, that is, from a value q equal to either $\mathrm{RD}(a/b)$ or $\mathrm{RU}(a/b)$. In turn, this makes it possible to efficiently implement floating-point division in software (Cornea-Hasegan *et al.* 1999).

Finally, a similar property holds for square root as well, but under the restriction that rounding is to nearest.

Property 2.14. Let $a \in \mathbb{F}_{\geq 0}$ and $s = \mathrm{RN}\left(\sqrt{a}\right)$. The term $a - s^2$ belongs to $\mathbb{F}$ if and only if there exist integers k_s and N_s, with $N_s \leq 2^p - 1$ such that $s = N_s \cdot 2^{k_s - p+1}$ and $2k_s \geq e_{\min} + p - 1$.

Note that Boldo and Daumas (2003) give an additional condition, equivalent in our notation to $k_s \geq e_{\min}$. It is not needed here since, as we have assumed $e_{\min} < 0$, it is a consequence of the assumption $2k_s \geq e_{\min} + p - 1$. Again, if the floating-point exponent e_s of s satisfies $2e_s \geq e_{\min} + p - 1$, then $a - s^2 \in \mathbb{F}$, but Property 2.14 is slightly more general.

Note also that unlike multiplication and division (but similarly to addition), this property of square root does not always hold if rounding is not to nearest. For example, if $a = 1 - 4\mathbf{u}$ then $s := \mathrm{RD}(\sqrt{a}) = 1 - 3\mathbf{u}$, so that $a - s^2 = 2\mathbf{u} - 9\mathbf{u}^2 \notin \mathbb{F}$; if $a = 1 - 3\mathbf{u}$ then $s := \mathrm{RU}(\sqrt{a}) = 1 - \mathbf{u}$, from which it follows that $a - s^2 = -\mathbf{u} - \mathbf{u}^2 \notin \mathbb{F}$.

2.9. Playing with the exceptions

The default exception handling of IEEE-754 arithmetic was designed so that it should frequently allow us to obtain a usable result, even when an exception occurs. For instance, for a huge value of x, the calculation of $3 + 1/x^5$ with rounded to nearest arithmetic still delivers the very accurate result 3, even if x^5 is infinite due to overflow. However, one should be cautious. Consider the following example by Lynch and Swartzlander (1992): $f(x) = x^2/\sqrt{x^3 + 1}$. For large x, $f(x) \approx \sqrt{x}$. However, if x is large enough for x^3 to overflow but not so large as to make x^2 overflow, the computed value will be 0. A typical example in binary64 arithmetic is with $x = 2^{400} \approx 2.5822 \cdots \times 10^{120}$: the computed value of $f(x)$ is 0 whereas the exact value is around 1.6069×10^{60}.

For arithmetic operations, what should be returned in exceptional cases is in general rather intuitive.[14] There is no debate on the fact that $(+\infty) \cdot (-\infty)$ should

[14] An exception is $\sqrt{-0} = -0$.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

be $-\infty$. When it comes to more complex functions, things become less clear, and from time to time somebody reopens the debate on what should be returned when computing $(\pm 1)^{+\infty}$ or 0^0; see an interesting discussion by Kahan (1997) and the choices suggested by the Standard (IEEE 2019, §9.1). Take as an example the function $\mathrm{cospi}(x) = \cos(\pi \cdot x)$ for $x = \pm\infty$. The mathematical function $\mathrm{cospi}(t)$ has no limit as $t \to \pm\infty$, hence it is natural to suggest that $\mathrm{cospi}(\pm\infty)$ should be NaN (this is the choice suggested by IEEE-754). However, since any sufficiently large floating-point number is an even integer, one may arguably claim that the choice $\mathrm{cospi}(\pm\infty) = 1$ would maintain consistent behaviour of the numerical program when the input variable x becomes infinite because of an overflow. These debates are interesting, and it is of course important to maintain the consistency and correctness of our calculations as much as possible, even in extreme cases. Nevertheless, despite the cleverness of the arithmetic of infinities, NaNs and zeros in IEEE-754, it does not automatically solve all problems, so we need a case-by-case analysis of any critical program that is supposed to work even when variables become infinite.

2.10. When more than one floating-point format is available

There is seldom only one floating-point format available on a given platform. There is even more diversity now than 20 years ago. In general, we used to have only binary32/single precision, binary64/double precision, and the 80-bit Intel 'double-extended' format. Now we still have these formats, plus binary128/quad precision, at least one 16-bit 'half-precision' format (binary16 or bfloat16), and sometimes even an 8-bit format. This diversity gives us more control when finding a compromise between speed and accuracy, and allows *mixed-precision* computations (Higham and Mary 2022), but it also brings the following difficulties.

2.10.1. Underflows/overflows due to change of formats

As Table 2.2 shows, the various floating-point formats in current use have considerably different ranges. Hence, to avoid underflows and overflows, conversion to a narrower format requires much care, that is, *scaling* data before conversion is often necessary. This is especially true when the target format is binary16.[15] Scaling strategies in LU factorizations algorithms are presented by Higham and Mary (2022, §7.4).

2.10.2. Double roundings

Double rounding occurs in round-to-nearest arithmetic when a result is first computed with a larger precision than the target precision. This was very frequent with the x86 instructions, as all arithmetic operations would first be performed using 'double-extended', precision-64, arithmetic operators, and then converted to the

[15] The very narrow range of the binary16 format was the major motivation for the introduction of bfloat16 for machine learning applications.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

target binary64 format. In fact, to prevent double rounding, a control register allows us to set the floating-point precision so that the results are not first rounded to double-extended precision, but setting it is a very costly operation, since it triggers a flush of the processor pipeline (Boldo and Melquiond 2008).

If the intermediate result is a midpoint of the target format, one cannot guarantee that the final result is a floating-point number of the target format nearest the exact result. Let us give an example. Assume that RN is round-to-nearest, ties-to-even,[16] and suppose that the target precision is 9 and the intermediate precision is 14. Consider the real number

$$x = 1.00101101011111111110101\cdots.$$

The floating-point number of the target precision nearest x is $x_{\text{correct}} = 1.00101101$. When rounding x to intermediate precision, we obtain

$$x_{\text{ext}} = 1.0010110110000,$$

and then, when rounding x_{ext} to target precision, we obtain

$$x_{\text{final}} = 1.00101110,$$

which is different from x_{correct}. Indeed, we have rounded x *up*, whereas it should have been rounded *down*. Figueroa (1995) and Roux (2014) showed that, when the intermediate precision is large enough, double rounding is innocuous for the basic arithmetic operations.

A solution to avoid double rounding would be to have the operations of the 'larger' format performed with the 'round-to-odd' rounding function RO, defined as follows. If $t \in \mathbb{F}$ then $\mathrm{RO}(t) = t$, otherwise $\mathrm{RO}(t)$ is the one of the two consecutive FP numbers that surround t whose integral significand is an odd integer. Unfortunately, RO is not among the rounding functions specified by IEEE-754. A variant of round-to-odd (for which $\mathrm{RO}(t)$ is always odd, which implies that when $t \in \mathbb{F}$ is even, $\mathrm{RO}(t) \neq t$) was first considered by von Neumann when designing the arithmetic unit of the EDVAC. A recent use of round-to-odd to avoid double roundings is in the decimal-to-floating-point conversion of the GCC compiler.[17] This rounding and its properties were also formally studied by Boldo and Melquiond (2008).

2.10.3. Portability/reproducibility issues

When an arithmetic operation $z = x \text{ op } y$ is performed, even if x, y and z do not have the same floating-point format, the IEEE-754 Standard is clear about what must be returned: the exact result rounded (with the chosen rounding function) to the format of z. The problem is that there is often no such thing as 'the format of z'.

[16] Double rounding also occurs with the other tie-breaking rules.

[17] https://www.exploringbinary.com/gcc-avoids-double-rounding-errors-with-round-to-odd/

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

This occurs whenever z is not an *explicit variable* but an *implicit one*, such as the result of $(a + b)$ when evaluating the expression $(a + b) \cdot (c + d)$.

Several choices may make sense: the widest format available in hardware (this would preserve accuracy and minimize the risk of spurious underflow or overflow but might sometimes slow down the calculation considerably), the maximum of the width of the operands, the format chosen for the variable that stores the final result of the expression evaluation, a 'preferred format' chosen by the programmer through some mechanism, etc.

Different choices made by compiler designers may result in different behaviours of the same program, which would result in portability and reproducibility issues. Even with the *same choice* we may meet reproducibility problems, as 'the widest format available in hardware' may not mean the same thing on different platforms. In its appendix (Section 10), the IEEE-754-2019 Standard recommends (yet does not require) 'preferred width' mechanisms to at least partly alleviate this problem.

2.10.4. Design of function libraries: need to consider many programs

A large number of floating-point formats is a challenge for the designers of libraries for the elementary (cos, sin, exp, arctan, etc.) or special functions. If one wishes to implement a library of around 30 functions (which is not that many – physicists and statisticians would like to have more; see Beebe 2017 for examples of functions that are frequently needed in numerical computing), the library designer needs to provide real and possibly complex functions for each of the FP formats. This is already a large number but in practice many variants are needed for each of the pairs function/format: one optimized for accuracy, one for latency, one for throughput. Beyond that, it will be interesting, for each of these combinations, to have a 'generic' version that works reasonably well on every platform, and specialized versions that are optimized for each widely distributed architecture. One may easily end up with thousands of programs to maintain, improve, keep mutually consistent, etc. The medium-term solution is presumably to (at least partly) automate the generation of function libraries (Brunie, de Dinechin, Kupriianova and Lauter 2015), and the long-term solution might well be to generate the function approximations at compilation time.

2.11. Problems related to the decimal input or output of numerical values

Numerical constants in a program, input values entered on a keyboard or read from a file, and output values displayed or written into a file are often represented in decimal. Base conversions are therefore a very frequent operation. It is well known that some numbers that have a finite decimal representation do not have a finite binary representation (a typical example is $1/10$). Although the numbers with a finite binary representation do have a finite decimal representation, that decimal representation may be too long to be convenient for most purposes. For instance, the smallest positive normal number in the binary64 format, i.e. 2^{-1022},

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

once converted into decimal, is

$$2.2250738585072013830902327173324040642192159804623318305533274168872044348139181958542831590125110205640673397310358110051$$
$$5243416155346010885601238537771882113077799353200233047961014744258363607192156504694250373420837525080665061665815894872049$$
$$1179968591639648500635908770118304874799780887753749949451580451605050915399856582470818645113537935804992115981085766051 9$$
$$9243335211435239014879569960959128889160299264151106346631339366347758651302937176204732563178148566435087212282863764204 48$$
$$4681140761391147706280168985324411002416144742161856716615054015428508471675290190316132277889672970737312333408698898317 50$$
$$6783884692609277397797285865965494109136909540613646756870239867831529068098461721092462539672851 5625 \times 10^{-308}.$$

As a consequence, inexact conversions between decimal and binary are very frequent. While this is harmless in general, the reader should be aware of a few issues, listed below.

2.11.1. Constants expressed in decimal in programs

When a constant such as `0.1` appears in a program, it is converted to a floating-point number. The format of that floating-point number is not obvious and depends on the specification of the programming language. For instance in C it will be `double` – i.e. binary64 – unless an `f`, `F`, `l` or `L` suffix is appended to the constant, whereas in Fortran it will be binary32 by default – although compiler options[18] may allow this to be changed. Numerical programmers should always explicitly declare the type of their constants. An even better solution, though less readable, is to express the constants in the hexadecimal representation of FP numbers specified by IEEE-754 (IEEE 2019, §5.12.3), which completely avoids conversion errors.

2.11.2. Roundtrip conversions

An intermediate result, computed in binary floating-point arithmetic, may be stored in decimal in a file, or displayed in decimal on a screen, and later on reread to carry on a calculation. The consequence is a *roundtrip base conversion:* first from binary to decimal and then from decimal to binary. Ideally, one would like that roundtrip conversion to be errorless, i.e. the initial binary FP number is recovered. Assuming that the binary format has precision p_2, the decimal input/output format has precision p_{10}, and both conversions are to a nearest value, the condition that ensures that, at least in the absence of underflow and overflow, all roundtrip conversions are identity (Matula 1968, Goldberg 1967) is

$$10^{p_{10}-1} > 2^{p_2}. \tag{2.7}$$

Table 2.4 gives the smallest value of p_{10}, say p_{10}^*, that satisfies (2.7) for the most common binary precisions. The IEEE-754 Standard (IEEE 2019, §5.12.2) requires that correctly rounded conversions to and from decimal formats of precision up to at least $p_{10}^* + 3$ are provided.

For a given binary number x, a good binary-to-decimal conversion choice is to use the *smallest* value of p_{10} that allows for an errorless roundtrip conversion (Steele Jr and White 2004).

[18] Such as `fpconstant` with the Intel Fortran compiler.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Table 2.4. Minimal decimal precision p_{10}^* that allows for an error-free roundtrip conversion, depending on the precision p_2 of the binary floating-point format.

Format	bfloat16	binary16	binary32	binary64	double-extended	binary128
p_2	8	11	24	53	64	113
p_{10}^*	4	5	9	17	21	36

2.12. *Exotic yet standard operators*

Beyond the usual arithmetic operations, the IEEE-754 Standard specifies some 'operations' that can be extremely useful in practice. Let us now present the most important of them.

2.12.1. *scaleB and logB operations*

If they are implemented efficiently, the following functions are extremely useful when one needs to *scale* a calculation to avoid spurious underflow or overflow. Typical examples are the calculation of Euclidean norms, complex square roots, etc. They are specified by IEEE (2019, §5.3.3).

- **scaleB**(x, k) returns (in a binary format, which is the case considered in this paper) $x \cdot 2^k$ (where x is an FP number and k is an integer).
- **logB**(x) returns $\lfloor \log_2 |x| \rfloor$ (where x is an FP number).

In the C programming language, they are called `scalbn` and `logb` in binary64 arithmetic, and `scalbnf` and `logbf` in binary32 arithmetic.

2.12.2. *min, max and copysign operations*

In pipelined calculations, branchings can considerably hinder performance. The following instructions (if implemented efficiently) make it possible in some cases to avoid them. They are specified by IEEE (2019, §9.6) and IEEE (2019, §5.5.1).

- **minimum**(x, y) and **minimumNumber**(x, y) are equal to x if $x \leq y$ and y otherwise. If one of the operands is -0 and the other one is $+0$ then -0 is returned. The difference between the two instructions is that $\text{minimum}(x, \text{NaN}) = \text{minimum}(\text{NaN}, x) = \text{NaN}$, whereas $\text{minimumNumber}(x, \text{NaN}) = \text{minimumNumber}(\text{NaN}, x) = x$. The availability of these instructions is only recommended (i.e. not required) by IEEE-754.
- **maximum**(x, y) and **maximumNumber**(x, y) are equal to y if $x \leq y$ and x otherwise. If one of the operands is -0 and the other one is $+0$ then $+0$ is returned. The difference between the two instructions is that $\text{maximum}(x, \text{NaN}) = \text{maximum}(\text{NaN}, x) = \text{NaN}$, whereas $\text{maximumNumber}(x, \text{NaN}) = \text{maximumNumber}(\text{NaN}, x) = x$. The availability of these instructions is only recommended (i.e. not required) by IEEE-754.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

- **minimumMagnitude**(x, y) is y if $|x| > |y|$, x if $|y| > |x|$, otherwise minimum(x, y). The availability of this instruction is only recommended.
- **maximumMagnitude**(x, y) is x if $|x| > |y|$, y if $|y| > |x|$, otherwise maximum(x, y). The availability of this instruction is only recommended.
- **copysign**(x, y) returns $\text{sign}(y) \times x$.

3. Execution environment

While the IEEE-754 Standard precisely specifies what floating-point numbers are and how arithmetic operators handle them, this should be taken with a grain of salt, as the execution environment might not comply with it. Indeed, for performance reasons, some corners might have been cut. This might happen at the hardware level (Section 3.1) or in programming languages and compilers (Section 3.2). Moreover, when it comes to mathematical libraries, the Standard is much more permissive, and various trade-offs between accuracy and performance exist (Section 3.3). Finally, considerations related to parallelism can also lead to non-reproducible computations (Section 3.4). Monniaux (2008) explored some of these issues in more detail.

3.1. Hardware

The IEEE-754 Standard does not mandate that a compliant environment has to be implemented in hardware. A pure software library would be just as compliant (and indeed there are such libraries, such as Hauser's excellent Berkeley SoftFloat[19]) but this would have a major impact on the performance. Let us explore what a hardware implementation entails, in order to understand which features are likely to be provided in hardware.

3.1.1. Subnormal numbers

Consider the rounded product of two binary floating-point numbers $M_1 \cdot 2^{e_1-p+1}$ and $M_2 \cdot 2^{e_2-p+1}$, i.e. $\text{RN}((M_1 \times M_2) \cdot 2^{e_1+e_2-2p+2})$. At first glance this is a simple operation: we compute the integer product of M_1 and M_2 and then truncate it to fit the target format, possibly adjusting it by one unit when rounding toward infinity. Let us assume that both inputs are positive normal numbers, i.e. $2^{p-1} \leq M_1 < 2^p$ and $2^{p-1} \leq M_2 < 2^p$. Thus the integer product has either $2p - 1$ or $2p$ bits. This means that if the result is also a normal number, the truncation position is almost known statically. One can systematically drop the $p - 1$ least significant bits, and possibly one more, if the most significant bit of the product is non-zero. Thus, in the normal range, the hardware design for the floating-point product is straightforward (Muller *et al.* 2018, §8.4.1).

[19] http://www.jhauser.us/arithmetic/SoftFloat.html

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

But if the result is in the subnormal range, the truncation position is no longer known beforehand, so the circuit has to be able to perform a large shift to the right. This increases the area of the floating-point unit, and more importantly, it might increase the pipeline depth by a few cycles. Subnormal numbers are also an issue when they appear as inputs, as some circuits are designed with normal inputs in mind. In that case the inputs need to be renormalized beforehand, which again incurs the use of shifters and hence an additional delay.

Thus processor designers might consider that supporting subnormal numbers in hardware is not worth the cost. Several options then appear, the first of which is simply to forsake both gradual underflow and compliance with the IEEE-754 Standard. In that case, subnormal inputs will be silently interpreted as floating-point zeros, while subnormal outputs will be flushed to zero. Note that the behaviour might vary depending on the operator. For example, it is possible to design a floating-point adder in such a way that subnormal inputs and outputs are handled at no extra cost, in contrast to the multiplier (Muller *et al.* 2018, §7.3.3).

Since the IEEE-754 Standard does not mandate any hardware support, another approach is to emulate subnormal numbers using either some microcode or a software library. As a consequence, performance will be optimal in general (i.e. in the normal range), but might plummet as soon as a computation encounters a subnormal number (Lawlor *et al.* 2005). To avoid this pitfall, the hardware might give the programmers the choice, either globally or on a per-instruction basis, to disable subnormal handling. In that case, the floating-point unit does not have to request any kind of software emulation to complete the operation. In other words, the floating-point environment is no longer compliant, but the decision to drop compliance is put in the user's hands.

3.1.2. Rounding directions

Subnormal numbers are not the only part of the Standard that are an impediment for hardware. Rounding directions are also an issue. This time there is no difficulty at the level of the floating-point unit. If it supports round-to-nearest, supporting the other directions does not incur any extra cost. The issue lies instead in the way the information about the rounding direction reaches the floating-point unit.

A first approach would be to encode this direction in the opcode of the floating-point instruction, but space is usually at a premium there. So hardware designers might decide that it is not worth wasting several bits on it to support the few programs that might need to round in another direction than to nearest (mainly users of interval arithmetic). These programs might thus have to rely on software emulation.

Another approach is to dedicate a control register to dynamically inform the floating-point unit about the current rounding direction. This time all the computations are performed in hardware. Unfortunately, on some architectures, changing the value of control registers might have a drastic impact. For example, it might stall the processor pipeline, so that all the floating-point operations in flight can

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

complete before the rounding direction is effectively changed. Thus routinely switching from round-to-nearest to another direction and back might considerably degrade performance.

3.1.3. FMA, division and square root

Another impediment to hardware design is the FMA operation (defined in (2.4)). Since there is no rounding after the multiplication, the subsequent addition needs to be much wider than usual (e.g. 106 bits for binary64) and so is the shifter to normalize the result. But that might not be the main issue. Indeed, the FMA operation might actually be the only opcode in the whole instruction set that needs to read three registers at once. As a consequence, supporting it in hardware might require a complete redesign of the entire processor architecture just for that purpose.

Division and square root are two other operations that come with their share of difficulties. Indeed, rather than having some costly dedicated hardware for them, it might be cheaper to use an iterative algorithm that computes a few bits of the result every cycle. Another approach is to implement a variant of the Newton–Raphson iteration using an FMA, which roughly doubles the number of bits of the result at each iteration, and performs a final correction essentially based on Property 2.13 (Cornea-Hasegan *et al.* 1999). In both cases, division and square root no longer fit inside the traditional pipeline; they might instead be handled by a microcode loop, thus impacting other operations around them.

Useful and detailed tables of latencies and throughput of instructions for the most common processors are regularly updated by Agner Fog.[20] In terms of latency as well as in terms of throughput, the cost of floating-point division is between 3 and 10 times the cost of floating-point addition or multiplication, and the cost of square root is similar or worse. This must be taken into account, for instance, when hesitating between approximating some function by a polynomial or by a rational function. There is a chicken-or-egg issue here: since division is slower, programmers of numerical applications tend to avoid using it, and since it is less used, computer manufacturers do not make the necessary efforts to significantly accelerate it.

3.1.4. Decimal arithmetic

While binary arithmetic is most often used when it comes to numerical computations, the revised IEEE-754 Standard also describes what a decimal floating-point arithmetic 'shall' comply with. Most of the previous considerations apply equally to binary and decimal arithmetic. Subnormal numbers, however, are much less of an issue as inputs. Indeed, there is not a unique representation of a given decimal number but a *cohort* of them: any number of most significant digits can be zero, as long as no information is lost (IEEE 2019, §3.5.1). In other words, any decimal number has a subnormal representation, in essence.

[20] See https://www.agner.org/optimize/instruction_tables.pdf.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Another peculiarity of decimal arithmetic is that the IEEE-754 Standard did not succeed in mandating a single encoding of the interchange formats for decimal floating-point numbers (IEEE 2019, §3.5.2). Indeed, two encodings of the significand are proposed.[21] When using the *binary* encoding of decimal floating-point numbers, the significand is mostly stored as a binary integer, which makes it simpler for software implementations to read and write floating-point numbers. When using the *decimal* encoding, the significand is mostly stored by groups of three decimal digits (hence ten bits), which makes it simpler for hardware implementations to perform shifts on the representation. Both encodings fortunately represent the same set of floating-point values, so they have no impact on the computed results. But they might introduce some portability issues, when accessing or transmitting decimal numbers.

3.2. Systems, languages, and compilers

Hardware is not the only factor one has to take into consideration when devising floating-point programs. Programming languages and compilers might also impede portability and reproducibility, even when the code seems to have a deterministic semantics according to the IEEE-754 Standard.

3.2.1. Java

Let us illustrate the issue with the Java programming language. The original language aimed at reproducibility of floating-point computations, and to do so, it was mostly sticking to the semantics of the IEEE-754 Standard. Unfortunately, few processors at the time were complying to the Standard, thus crippling the implementation of the Java Virtual Machine. In particular, the legacy floating-point unit of the x86 architecture was using 80-bit registers but made it possible to emulate binary32 and binary64 operations by setting a control register. This support, however, was only partial; the significand was rounded at the specified precision, but the exponent range was left unchanged. As a consequence, the floating-point result would always comply with the IEEE-754 Standard, except in the subnormal range where it might rarely be off by one, due to double rounding.

The inability to have both hardware support and reproducible results led the developers to weaken the language in the late 1990s. Starting from Java SE 1.2, reproducibility of floating-point computations would no longer be guaranteed by the language, unless the Java methods were explicitly annotated with the `strictfp` keyword. In that case the Java Virtual Machine would fall back to software emulation on hardware that was not compliant with the IEEE-754 Standard. Today non-compliant hardware has been sufficiently phased out to restore the original Java semantics and make `strictfp` unnecessary (Darcy 2017).

[21] Note that, unlike for binary floating-point numbers, the exponent and significand fields are not cleanly separated. Indeed, a variable number of bits of the exponent field of decimal floating-point numbers act as the most significant bits of the significand.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

3.2.2. C and Fortran

In contrast to Java, languages such as C and Fortran predate the IEEE-754 Standard and reproducibility was hardly a concern at the time. Instead, in the case of Fortran, the semantics of floating-point operations was motivated by the usage of the language, e.g. numerical simulation. So, floating-point numbers were just an approximation of real numbers. This led the Fortran standard to mandate that 'the processor may evaluate any mathematically equivalent expression' (ISO Fortran 2008, §7.1.5.2.4). Here, mathematically equivalent expressions should be understood as having the same values when rounding, underflow and overflow are ignored. For example, if the original program evaluates the expression $X - Y + Z$, the compiler can rewrite it into $X - (Y - Z)$, regardless of any consideration for catastrophic cancellation.

The authors of the Fortran standard did, however, provide an escape hatch, i.e. parentheses. Indeed, the transformation is only allowed if 'the integrity of the parentheses is not violated'. In other words, while a Fortran compiler can rewrite $X - Y + Z$ into $X - (Y - Z)$, it cannot perform the converse transformation, as the input expression now contains parentheses. This also means that parentheses are not just used to disambiguate the parsing of arithmetic expressions: they are semantically relevant. For example, a compiler can factorize $X \times Y + X \times Z$ into $X \times (Y + Z)$ but it cannot factorize $(X \times Y) + X \times Z$. It can, however, use an FMA to evaluate both expressions.

Regarding floating-point computations, the C language is stricter than Fortran. Thus compliant compilers cannot perform this kind of algebraic rewriting, even in the absence of parentheses.[22] However, compilers still have some leeway. Indeed, except for assignment and cast, the C11 standard states that 'the values yielded by operators with floating operands ... are evaluated to a format whose range and precision may be greater than required by the type' (ISO C 11, §5.2.4.2.2). The main consequence is that even if the type of an expression is `float`, it might be evaluated as a binary64 number rather than a binary32 number. This excess precision can in turn cause double rounding issues and thus lower the accuracy in some rare cases.

A less obvious consequence is that it allows C compilers to make use of FMA operations. Consider the expression $X \times Y + Z$. Irrespective of the type of $X \times Y$, this product could be virtually performed with an infinite precision and thus fused with the subsequent addition into a single FMA operation. To prevent this optimization, the user has to explicitly cast the product $X \times Y$ to its target type or to assign its result to a variable, before proceeding with the sum.

Note that the ability of C and Fortran compilers to fuse multiplication and addition can lead to some unintuitive behaviours. Indeed, given the expression

[22] This assumption only holds if compiler options that break compliance with the C Standard, e.g. `-ffast-math`, are not in use.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

$X \times X - X \times X$, the language standards allow the compilers to produce a code that actually computes $\mathrm{FMA}(X, X, -\mathrm{RN}(X^2))$, whose result is often non-zero.

3.3. Mathematical libraries

Previous sections dealt with basic arithmetic operators. Let us now move to mathematical functions. While processors might provide some primitives and compilers might know about them, floating-point approximations of mathematical functions are, for the most part, handled in software libraries. These libraries provide various trade-offs. The reader interested in mathematical function algorithms can consult the books by Beebe (2017) and Muller (2016). Tests of recent mathematical libraries can be found in Innocente and Zimmermann (2022).

We do not address here the problem of generating random FP numbers, but it is an important topic, and the naive solutions can be deceptive (Goualard 2022).

3.3.1. The table-maker dilemma

First of all, in contrast to the situation with basic arithmetic operators, correct rounding is no longer a given. Indeed, correct rounding in round-to-nearest (resp. directed rounding) requires the ability to decide on which side of the midpoint between two consecutive floating-point numbers (resp. which side of a floating-point number) lies an infinitely precise mathematical value. This might require a library to perform its internal computations with very high precision. For example, consider $\sin(x) = \sin(x - 2\pi k)$ for some integer k. By choosing a large enough dyadic number x, one can make $x - 2\pi k$ arbitrarily close to any real number in $[-\pi, \pi]$, which in turn makes $\sin(x)$ arbitrarily close to a floating-point midpoint. Fortunately, as a finite floating-point number, x has both a bounded precision and a bounded range, so there is an actual limit to how close $\sin(x)$ can be to a midpoint. This limit gives an indication regarding the accuracy needed when approximating $\sin(x)$ in order to round it correctly. This issue is not specific to the periodic nature of the trigonometric functions. It also impacts other mathematical functions. For example, there are a few inputs x in the binary64 range such that $\ln(x)$ is at a relative distance $\sim 2^{-117}$ of a floating-point number or a midpoint, which means that $\ln(x)$ has to be approximated with an accuracy of about 117 bits in order to compute its correctly rounded binary64 value. An example by Lefèvre and Muller (2001) is the binary64 number

$$x = 6239214722492854 \times 2^{626},$$

whose logarithm (with the significand expressed in binary) is equal to

$$\overbrace{1.1101011001000111100111101011101001111100100101110001}^{53 \text{ bits}}\underbrace{000000000000000000\cdots000000000000000000}_{65 \text{ zeros}}111001\cdots \times 2^8.$$

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

In the case of the natural logarithm in binary64, the hard-to-round results are known, which means that one can bound the time and space complexities of an algorithm that computes its correctly rounded approximation. But for some other functions, e.g. sin in binary64, this is not yet the case. For such a function, the algorithm has to compute increasingly accurate approximations of the infinitely precise result until it can decide what the correctly rounded result is. Thus, while the time and space complexities are necessarily bounded for a given floating-point format, our inability to put upper bounds on them means that correct rounding of these functions is not yet advisable for critical real-time systems. Hopefully this inability will be temporary: progress is being made on the topic (Brisebarre, Hanrot and Robert 2017).

Despite performance issues, correct rounding has some important properties. Indeed, its uniqueness guarantees portability and reproducibility. Moreover, its accuracy guarantees that monotony properties of the mathematical functions are also satisfied by their approximation. Thus it would be unfortunate to drop correct rounding entirely. For example, while sin is not yet fully understood on the whole binary64 range, we know its hard-to-round cases for some useful input intervals, e.g. $[-2\pi, 2\pi]$. So, a trade-off can be reached: even if a mathematical library does not provide correct rounding everywhere, it can still document the functions and the input intervals for which it is guaranteed. This is the approach followed by the IEEE-754 Standard. Let us mention the important current effort around the CORE-MATH library[23] of very efficient correctly rounded functions (Sibidanov, Zimmermann and Glondu 2022).

3.3.2. Rounding directions

Considering correct rounding is only meaningful when a rounding direction is decided. Most correctly rounded libraries at least support the default rounding mode, i.e. round-to-nearest, ties-to-even. The situation regarding directed rounding is more diverse. First, a library might only support round-to-nearest. Second, a library might provide alternative implementations of all the mathematical functions, one for each rounding mode. Third, a library might provide only one implementation of each function, but written in such a way that the behaviour of the function respects the dynamic rounding mode of the processor, as the hardware arithmetic operators do. Conversely, it should be noted that libraries that fall into the first two categories are usually not resilient to the dynamic rounding mode and will work correctly only when set to round-to-nearest.

3.3.3. Trade-off between accuracy and performances

Let us now assume that a given mathematical library has forsaken the optimality of correct rounding on some input ranges. This opens the way to a wide range of potential implementations, with a trade-off between speed and guaranteed accuracy.

[23] See https://core-math.gitlabpages.inria.fr/.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

For example, a mathematical library could promise that the computed results are at a distance of $0.5 + \varepsilon$ ulp of the infinitely precise results for round-to-nearest. For ε small enough, e.g. 10^{-4}, this would allow for correct rounding in most of the cases. By decreasing ε even further, at the expense of a larger computation time, one could even make the probability of an incorrectly rounded result negligible. At the other end of the spectrum, some applications might not even need an accuracy close to the precision of the format: they could live with only faithfully rounded results, or even worse.

Up to now, we have considered a single evaluation of $\sin(x)$ in isolation. But some applications might need to perform numerous evaluations in parallel. Some mathematical libraries hence provide implementations that, through careful use of pipelined or vector floating-point units, can perform such parallel computations. This severely constrains the kind of algorithms that can be implemented, and might thus limit the accuracy of such implementations (Shibata and Petrogalli 2020). Consider the use of a width-n vector floating-point unit to evaluate $\sin(x_1), \ldots, \sin(x_n)$ in parallel. The n evaluations need to have the same control flow as often as possible, which means that conditional branching has to be avoided. Instead, the algorithm will use predicated instructions to control whether a given floating-point operation is performed on a given $\sin(x_i)$ lane.[24] As a consequence, the latency for producing $\sin(x_1), \ldots, \sin(x_n)$ is at least the latency of the slowest lane. When the predication varies widely between the n lanes, it could even end up being as slow as if the n evaluations had been performed sequentially. Thus algorithms have to be simple and short to benefit from the use of a vector unit, which limits the accuracy. Mathematical libraries for graphical processing units suffer from similar shortcomings.

3.4. Reproducible computations

Except for Not-a-Number floating-point data, the principles of the IEEE-754 Standard, especially correct rounding, guarantee the reproducibility of floating-point computations across several program executions. These principles could even ensure the portability of floating-point results across various environments. Unfortunately, for larger systems, some other factors have to be taken into account, most prominently parallelism.

Let us consider the example of a large sum of floating-point numbers, e.g. an inner product as commonly encountered in matrix computations. If the order of floating-point additions is set once and for all, then reproducibility and portability are guaranteed. But setting this order statically might hinder performances. Indeed, assuming that several computation cores are available, one might want to split this large sum into smaller blocks of numbers and dispatch them to the various cores, so that all the blocks can be summed in parallel. The final sum can then be performed

[24] Conditional moves can also be used, in which case the floating-point operation is performed on all the lanes, but its result might be discarded on some of them.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

by adding the partial sums. While this approach is sensible in an infinitely precise setting, the lack of associativity of floating-point addition makes the results non-reproducible when the size of the blocks is set dynamically, depending on the availability, speed and bandwidth of the cores.

Even if only one core is available, computations might still be performed per block, for performance reasons. For instance, matrix multiplication might be subdivided into smaller blocks (hence partial inner products) in order to best benefit from cache locality. Such an algorithm would thus have non-portable results, as they would depend on the size of the cache. Another cause for divergence is the availability of a floating-point vector unit. This is quite similar to the previous instances, except that the number of blocks then depends on the width of the vector unit, and the blocks are interleaved instead of contiguous. Even if there is no vector unit, performances might still benefit from interleaving floating-point computations depending on the length of the instruction pipeline.

While some of these choices can be decided statically at compilation time rather than dynamically at execution time, they still impact the portability of something as simple as matrix multiplication, when performances matter. An illustration of these considerations lies in the ATLAS implementation of the BLAS routines.[25] As explained by Whaley, Petitet and Dongarra (2001), its routines take into account the size of the L1 cache, whether the FMA instruction is available, the pipeline depth, the number of registers (for loop unrolling), and so on.

To improve reproducibility, various solutions have been proposed. Most of them come down to performing only exact operations, so as to completely ban the issue of floating-point addition not being associative once rounded. For the binary64 format, Kulisch (2013) proposed using a long accumulator that would cover the whole range of floating-point numbers or, in the case of the scalar product, the whole range of products of floating-point numbers. This 4288-bit register requires some expensive hardware (Uguen and de Dinechin 2017) but offers a correctly rounded result by not losing any bit of information during computation. Software implementation of the long accumulator usually does not fare well, due to the indirect memory access it entails. However, if the dynamic range of the inputs is not too large, a careful implementation of the summation can get close to the optimal performance of the non-associative floating-point addition, as shown by Collange, Defour, Graillat and Iakymchuk (2015).

If correct rounding is not needed, one does not need the whole accumulator: only the most significant bits of the sum need be kept. For the performed additions to be exact, all the inputs can be split according to a given boundary (Section 4.3), ensuring that their significands are aligned. In the case of a distributed computation, it might be difficult to find a suitable boundary in a reproducible way. Demmel and Nguyen (2015) have proposed that every core should compute its partial sum as a triple of floating-point numbers aligned on boundaries of the form $2^{i \cdot W}$, where

[25] https://math-atlas.sourceforge.net/

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

W is 40 for the binary64 format (see Section 5.3). The final reduction can then keep only the relevant most significant parts of the triples and sum them, in order to obtain the same triple that would have been obtained if there had been only one core.

4. Taming rounding errors

Most prominently, floating-point arithmetic is used as a cheap way of performing computations on real numbers. As a consequence, the main issue is the rounding error that taints the result of every operation. This raises the question: How does one make sure that the computed result is close to the real one? Interval arithmetic answers by enclosing the real result between two floating-point bounds instead of approximating it with only one floating-point number, as shown in Section 4.1.

In some cases, one can recover the rounding error as a floating-point number, thus making it possible to propagate it along computations. That is the purpose of the error-free transformations presented in Section 4.2. Splitting floating-point numbers is another way of controlling rounding errors, by making sure only exact operations will be performed, as shown in Section 4.3.

When we are able to bound the error of a calculation (for instance using the standard model), we can use *floating-point filters* to avoid long-precision calculations in many situations. This is described in Section 4.4.

In the same way floating-point numbers are used to approximate real numbers, pairs of floating-point numbers are used to approximate complex numbers. The rounding error can then be considered from two different perspectives: normwise or componentwise. Section 4.5 provides accurate algorithms for both cases. Section 4.6 then considers rounding errors in higher dimensions.

Finally, Section 4.7 presents a few tools that make it possible to accurately approximate functions using floating-point polynomials.

4.1. Interval arithmetic

Given an expression e over the real numbers, it can be mechanically translated into an expression $\tilde{e}$ over the floating-point numbers. Replacing exact operations with their rounded counterparts introduces rounding errors in general, but evaluating $\tilde{e}$ will hopefully give a result sufficiently close to e. Guaranteeing this property requires an error analysis of the algorithm.

Rather than approximating e by $\tilde{e}$, a slightly different approach consists in enclosing e by an interval $\mathbf{e}$. This can be done mechanically by replacing every exact operation with its counterpart given by interval arithmetic (Moore 1979, Neumaier 1990, Moore, Kearfott and Cloud 2009, Rump 2010, IEEE 2015). Indeed, interval operators are defined in a way that guarantees the *containment* property. For instance, the difference of two intervals $\mathbf{v}$ and $\mathbf{w}$ satisfies the following property:

$$\text{for all } v, w \in \mathbb{R}, \quad v \in \mathbf{v} \wedge w \in \mathbf{w} \Rightarrow v - w \in \mathbf{v} - \mathbf{w}.$$

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Thus, by induction on the expression e, we get the enclosure $\mathbf{e}$ of e.

Intervals are usually implemented as pairs of floating-point numbers. These can represent either the lower and upper bounds of the interval, or its centre and radius. Directed rounding functions are especially useful when it comes to implementing an interval operation. For instance, the difference $\mathbf{v} - \mathbf{w}$ of two intervals given by their floating-point bounds $\mathbf{v} = [\underline{v}, \overline{v}]$ and $\mathbf{w} = [\underline{w}, \overline{w}]$ can be implemented as follows:

$$\mathbf{v} - \mathbf{w} = [\mathrm{RD}(\underline{v} - \overline{w}), \mathrm{RU}(\overline{v} - \underline{w})].$$

The monotonicity properties of subtraction and rounding, as well as the correct handling of infinities in the case of overflow, guarantee that the resulting interval indeed satisfies the containment property above. Note that even seemingly straightforward operations such as computing the midpoint of an interval require some care (Goualard 2014).

The containment property guarantees that the mathematical result e lies in the computed interval $\mathbf{e}$. Thus, as long as the resulting interval is narrow, this completely alleviates the need for an error analysis, at the expense of a computation that is roughly twice as slow. Another way to see it is that the error analysis that would be performed on $\tilde{e}$ is performed at run-time by the code of $\mathbf{e}$.

Even disregarding the performance issue, interval arithmetic is not a panacea for rounding errors, though. Indeed, there is no guarantee that the interval $\mathbf{e}$ is narrow enough for a given application. For example, if one needs to decide whether $e \geq 0$, one could compute its enclosing interval $\mathbf{e}$ and check whether it contains only non-negative values (or only non-positive values). But what if it straddles zero? In the worst case, one might even end up with $\mathbf{e} = [-\infty, \infty]$, which trivially satisfies the containment property but is utterly pointless.

The main reason for this issue is the *dependency problem*, also known as the *wrapping effect* in the context of dynamical systems (Lohner 2001). Indeed, naive interval arithmetic does not remember how sub-expressions are correlated; it has to assume that they are completely independent from each other. As soon as two sub-expressions depend on the same input variables, this assumption no longer holds, which implies that the resulting interval may be overly pessimistic. The archetypal example is the subtraction $x - x$:

$$x - x \in \mathbf{x} - \mathbf{x} = [\mathrm{RD}(\underline{x} - \overline{x}), \mathrm{RU}(\overline{x} - \underline{x})].$$

Unless the interval $\mathbf{x}$ that encloses x is a singleton, the resulting interval cannot be the optimal interval $[0, 0]$.

The example $x - x$ might seem artificial but it is representative of various algorithms. For example, consider the Newton–Raphson iteration used to find the root of a function f:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

If we let $\varepsilon_n = x_n - x$ denote the distance between x_n and the target root x, we hope that the quotient $f(x_n)/f'(x_n)$ is close to ε_n, so that x_{n+1} is close to the root x. In other words, the code is expected to compute a difference akin to $(x+\varepsilon_n)-\varepsilon_n$. This is bound to fail with interval arithmetic. While the floating-point iteration usually converges toward x, the following interval iteration would compute an ever larger interval:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{\mathbf{f}(\mathbf{x}_n)}{\mathbf{f}'(\mathbf{x}_n)}.$$

Thus we should be careful not to blindly replace floating-point operations with interval operations. An analysis of the original algorithm might be needed to reduce correlations and hence to ensure useful results. For instance, there is an interval version of the Newton–Raphson iteration given by Moore (1979, equation (5.16)), but it is slightly different from a naive translation. In particular, it involves the centre $m(\mathbf{x}_n)$ of the input interval $\mathbf{x}_n$ in some select places:

$$\mathbf{x}_{n+1} = \mathbf{x_n} \cap \left(m(\mathbf{x_n}) - \frac{\mathbf{f}(m(\mathbf{x}_n))}{\mathbf{f}'(\mathbf{x}_n)} \right).$$

If the initial interval $\mathbf{x}_0$ is large enough to contain the root x, any subsequent interval $\mathbf{x}_n$ is guaranteed to contain x too, thanks to a combination of both the containment property and the mean-value theorem applied to f around $m(\mathbf{x}_n)$:

$$0 = f(x) = f(m(\mathbf{x}_n)) + (x - m(\mathbf{x}_n)) \cdot f'(\xi) \quad \text{with } \xi \in \mathbf{x}_n.$$

4.2. *Error-free transformations*

An immediate consequence of the properties of Section 2.8 about correcting terms is that, more often than not, the error is also an FP number that can be computed with FP operations, with no need for multiple-precision software. The sequence of operations that returns both the result of a floating-point operation and the error of that operation is called an *error-free transformation* (EFT). The first ideas seem to go back to Gill (1951), in the context of fixed-point arithmetic.

Algorithms 1 (page 219) and 2 (below) are explicitly mentioned by Møller (1965). Algorithm 1 also appears in the summation algorithm of Kahan (1965), which is Algorithm 20 (page 274). Several EFTs are given and used in the seminal article by Dekker (1971). The term *error-free transformation* was introduced by Ogita, Rump and Oishi (2005). We will now review the various error-free transformations and their specifics.

Note that some of these EFTs are given below with the rounding function RN. With other rounding functions, these transformations will not always be error-free. In particular, when discussing Property 2.11 we have given the example of an addition with a *directed* rounding function, whose error is not a floating-point number. However, under conditions discussed by Boldo, Graillat and Muller (2017), the various EFTs of addition and subtraction in any case return a value close

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

to the exact error,[26] so their results can still be useful for designing compensated algorithms (see Section 5.3). In all cases, the availability of subnormal numbers is needed for the EFTs to return a correct result.

4.2.1. EFT for addition

When rounding is *to nearest*, the error of an FP addition is an FP number (this is Property 2.11), and algorithms for computing that error have been available since at least the 1960s (Møller 1965).

A first such algorithm is Fast2Sum, i.e. Algorithm 1 described in Section 2.4. The two input FP numbers a and b are transformed into two FP numbers s and t such that, exactly, $a + b = s + t$ with $s = \mathrm{RN}(a + b)$. But it requires their floating-point exponents to satisfy $e_a \geq e_b$ (which is implied by $|a| \geq |b|$, a condition easier to check in practice). This condition is important, since t can be *very far* from the error of the FP addition of a and b when it is not satisfied. A simple example in binary64 arithmetic is $a = 1$ and $b = 2^{55}$, for which the algorithm returns $t = 0$ instead of 1.

In the general case, when we do not know beforehand whether $|a| \geq |b|$, we can rely on the 2Sum algorithm (Møller 1965, Knuth 1998), i.e. Algorithm 2. As before, the inputs are FP numbers a and b and the outputs are FP numbers s and t such that $a + b = s + t$ exactly and $s = \mathrm{RN}(a + b)$, but unlike in Algorithm 1, no assumption has to be made on a and b.

Algorithm 2 2Sum(a, b)

$s \leftarrow \mathrm{RN}(a + b)$
$a' \leftarrow \mathrm{RN}(s - b)$
$b' \leftarrow \mathrm{RN}(s - a')$
$\delta_a \leftarrow \mathrm{RN}(a - a')$
$\delta_b \leftarrow \mathrm{RN}(b - b')$
$t \leftarrow \mathrm{RN}(\delta_a + \delta_b)$
return (s, t)

Of course, the floating-point addition of a and b can overflow (Section 2.2). However, if it does not, then 2Sum *almost* never overflows,[27] while Fast2Sum is fully immune to spurious overflow. Indeed, if the first addition does not overflow, then none of the other operations overflow either (Boldo *et al.* 2017). Furthermore, 2Sum is 'optimal' in the following sense. If an algorithm made up of floating-point additions and subtractions always returns the same results as 2Sum (i.e. the result and the error of the FP addition of a and b), then it takes at least the same number of operations as 2Sum (Kornerup, Lefèvre, Louvet and Muller 2012).

[26] The same holds for square root, as we shall see that its FMA-based EFT is a sequence of only two floating-point operations.

[27] If $|a| \neq \mathbf{\Omega}$, it cannot overflow.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Priest (1991) has given an algorithm for addition that does *almost* the same thing as 2Sum, even when the rounding function $\circ$ is not RN (in fact, it only needs to be a faithful rounding). It returns a pair (s, t) such that $s + t = a + b$, with the guarantee that the floating-point exponent of t is at most the floating-point exponent of s minus p; however, s is not guaranteed to be $\circ(a + b)$.

4.2.2. EFTs for multiplication, division and square root

If two FP numbers a and b satisfy the condition of Property 2.12, then the error of their floating-point product can be obtained very simply using the 2Prod algorithm shown below. It is also called Fast2Mult (Kahan 1997, Nievergelt 2003, Muller *et al.* 2018). Note that it requires an FMA instruction to compute $\mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(ab - \pi)$.

Algorithm 3 2Prod(a, b)

$\pi \leftarrow \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(ab)$
$\rho \leftarrow \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(ab - \pi)$
return (π, ρ)

Algorithm 3 provides the exact rounding error, or the rounding of it in the case of underflow when computing ρ. Note the use of the 'generic' rounding function $\mathrm{R}_{\mathrm{D}}^{\mathrm{U}}$ instead of RN. Indeed, by Property 2.12, $ab - \pi$ is an FP number, so it is computed exactly, whatever the rounding function.

When no FMA is available, one can use an older algorithm by Veltkamp (1968, 1969) and Dekker (1971), which has been formally proved by Boldo (2006). It splits the FP numbers a and b into two (see Section 4.3), multiplies the halves and then subtracts all the parts in a way that ensures exact results thanks to properties similar to those of Section 2.4. It takes 17 floating-point operations, which is quite costly compared to Algorithm 3.

For division and square root, FMA-based algorithms similar to Algorithm 3 follow Properties 2.13 and 2.14, with the restriction that round-to-nearest is required to guarantee an exact residual in the case of square root.

4.2.3. Augmented operations from the 2019 revision of the IEEE Standard

The 2019 release of the IEEE-754 Standard for Floating-Point Arithmetic recommends (but does not require) that new augmented operations should be provided for the binary formats.[28] These operations are called **augmentedAddition**, **augmentedSubtraction** and **augmentedMultiplication**, and they are defined as follows (IEEE 2019, Riedy and Demmel 2018).

- **augmentedAddition**(x, y) delivers (a_0, b_0) such that $a_0 = \mathrm{RN}_0(x + y)$ and, when $a_0 \notin \{\pm\infty, \mathrm{NaN}\}$, $b_0 = x + y - a_0$. When $b_0 = 0$, it is required to have the same sign as a_0.

[28] History and motivation are given by Riedy and Demmel (2018).

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

- **augmentedSubtraction**(x, y) is the same as augmentedAddition$(x, -y)$.
- **augmentedMultiplication**(x, y) delivers (a_0, b_0) such that $a_0 = \mathrm{RN}_0(xy)$ and, when $a_0 \notin \{\pm\infty, \mathrm{NaN}\}$, $b_0 = \mathrm{RN}_0(xy - a_0)$. When $xy - a_0 = 0$, the number b_0 (equal to 0) is required to have the same sign as a_0.

Special cases (a_0 equal to $\pm\infty$ or NaNs) are described by the Standard (IEEE 2019). Note the use of the rounding function RN_0 (i.e. round-to-nearest, *ties-to-zero*), which differs from the default RN_e (round-to-nearest, *ties-to-even*). With this rounding function, these operations would significantly help to implement bitwise reproducible summation and dot product, using an algorithm given by Demmel, Ahrens and Nguyen (2016).

At the time of writing, these augmented operations are not implemented in hardware on any commercial platform. If, one day, an efficient implementation is provided on some processor, they will be good candidates for replacing the 2Sum, Fast2Sum and 2Prod algorithms presented above. However, due to the special rounding function, we will need to check if an algorithm that has been proved to work with the usual error-free transformations still works with these operations.

Boldo, Lauter and Muller (2021) have shown how to emulate these operations using conventional, ties-to-even, arithmetic. This implies a significant performance penalty, but makes it possible to design and test programs that use the augmented operations. These programs will be ready for use with full efficiency when the augmented operations become available in hardware.

4.2.4. EFT for the FMA

Assuming a round-to-nearest rounding function, the error of an FMA operation cannot, in general, be expressed as a single FP number. It can, however, be expressed as the unevaluated sum of two FP numbers (i.e. a *double-word*: see Section 5.1), which can be obtained using Algorithm 4. This EFT was introduced by Boldo and Muller (2005) and has since then been formally verified with the Coq proof assistant (Boldo and Muller 2011). Given three FP numbers a, b, c and assuming that no underflow or overflow occurs, this algorithm makes it possible to convert the exact expression $ab + c$ into the unevaluated sum of three floating-point numbers:

$$ab + c = d + e_1 + e_2,$$

where d, e_1, e_2 are such that

$$d = \mathrm{RN}(ab + c), \quad |e_1 + e_2| \le \tfrac{1}{2}\mathrm{ulp}(d), \quad |e_2| \le \tfrac{1}{2}\mathrm{ulp}(e_1).$$

Since the 2Prod algorithm performs one multiplication and one FMA, and since the 2Sum (resp. Fast2Sum) algorithm performs six (resp. three) additions, we deduce that the ErrFma algorithm performs one multiplication, two FMAs and 17 additions, i.e. 20 operations in total. In general, it is not known whether this cost can be reduced (which is in contrast to the optimality of 2Sum mentioned

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 4 ErrFma(a, b, c)

1: $d \leftarrow \mathrm{RN}(ab + c)$
2: $(v_1, v_2) \leftarrow \mathrm{2Prod}(a, b)$
3: $(s_1, s_2) \leftarrow \mathrm{2Sum}(c, v_2)$
4: $(s_3, s_4) \leftarrow \mathrm{2Sum}(v_1, s_1)$
5: $t \leftarrow \mathrm{RN}(\mathrm{RN}(s_3 - d) + s_4)$
6: $(e_1, e_2) \leftarrow \mathrm{Fast2Sum}(t, s_2)$
7: **return** (d, e_1, e_2)

in Section 4.2.1). However, if we know in advance that a, b, c are such that the exact error $ab + c - \mathrm{RN}(ab + c)$ fits into a single FP number instead of two (i.e. $e_2 = 0$), then the call to Fast2Sum in line 6 can be replaced by a single addition, $e_1 \leftarrow \mathrm{RN}(t + s_2)$, which saves two operations. This simplified version of ErrFma can be used for example when designing FP algorithms for correctly rounded square root reciprocals (Borges, Jeannerod and Muller 2022). Also, three floating-point operations (corresponding to the Fast2Sum of line 6) can be saved by directly returning (d, t, s_2), at the price of an 'un-normalized' output (Boldo and Muller 2005). This is much in the spirit of Lange and Rump's pair arithmetic (see Section 5.2).

In some other applications, the exact error of the FMA is in fact not needed and can be replaced by any good enough approximation. In such cases, we can thus relax the specification of ErrFma in order to produce a single FP number $\widehat{e} \approx e_1 + e_2$ instead of the pair (e_1, e_2) and, hopefully, obtain a cheaper algorithm. For example, if we choose to compute the correctly rounded value

$$\widehat{e} = \mathrm{RN}(e_1 + e_2)$$

then, using $e_1 + e_2 = t + s_2$ and $e_1 = \mathrm{RN}(t + s_2)$, it again suffices to replace line 6 in Algorithm 4 with $\widehat{e} \leftarrow \mathrm{RN}(t + s_2)$. This gives the best possible approximation to the error of the FMA in 18 FP operations instead of 20, for every input (a, b, c) such that no underflow or overflow occurs. This variant of ErrFMA is of course no longer an EFT, since the exact sum $d + \widehat{e}$ can now differ from $ab + c$, but the pair $(d, \widehat{e})$ can be viewed as a double-word approximation to $ab + c$:

$$\begin{aligned} |d + \widehat{e} - (ab + c)| &= |\mathrm{RN}(e_1 + e_2) - (e_1 + e_2)| \\ &\le \mathbf{u}|e_1 + e_2| \\ &\le \mathbf{u}^2|ab + c|. \end{aligned}$$

Here the relative error of the exact sum $d + \widehat{e}$ is at most $\mathbf{u}^2 = 2^{-2p}$, as if we had rounded $ab + c$ to nearest in precision $2p$. If we accept relaxing the accuracy specification a little further and can tolerate a relative error bound of the form $O(\mathbf{u}^2)$ for some modest hidden constant, then even faster solutions are possible. In particular, Algorithm 5 was proposed by Boldo and Muller (2011), who used 12 FP

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 5 ErrFmaApprox(a, b, c)

$d \leftarrow \mathrm{RN}(ab + c)$
$(v_1, v_2) \leftarrow 2\mathrm{Prod}(a, b)$
$(w_1, w_2) \leftarrow 2\mathrm{Sum}(c, v_1)$
$t \leftarrow \mathrm{RN}(w_1 - d)$
$e \leftarrow \mathrm{RN}(t + \mathrm{RN}(v_2 + w_2))$
return (d, e)

operations to produce a pair (d, e) such that

$$d = \mathrm{RN}(ab + c), \quad d + e = (ab + c)(1 + \delta), \quad |\delta| \leq 14\mathbf{u}^2.$$

Algorithms 4 and 5 were used by Kouya (2019) to implement some BLAS1 functions in double-word arithmetic. The experiments reported there for $\mathbf{u} = 2^{-53}$ are clearly in favour of the second algorithm, which in this specific context is significantly faster than the first one while preserving the same level of accuracy. Comparisons with Lange and Rump pair arithmetic (see Section 5.2) still need to be done. The EFTs can be used to emulate stochastic roundings when no hardware implementation is available (Févotte and Lathuilière 2016, Croci *et al.* 2022).

4.3. Splitting algorithms

We sometimes need to *split* a precision-p floating-point number, i.e. to decompose it into two FP numbers x_h and x_ℓ of smaller significand size, such that $x = x_h + x_\ell$ and $|x_\ell| \leq \frac{1}{2}\mathrm{ulp}(x_h)$ (or, in some cases, $|x_\ell| < \mathrm{ulp}(x_h)$). Among the many applications of this, let us mention the following ones.

- By requiring x_h to have a one-bit significand, we can compute $\mathrm{ufp}(x)$.
- Very similarly, but with looser constraints, we may require $|x_h|$ to be a power of 2 close to $|x|$. This can be useful when *scaling* some calculations, for example to avoid spurious underflows or overflows when computing Euclidean norms and performing some operations in complex arithmetic.
- By requiring $|x_\ell|$ to be less than one and x_h to be an integer, we compute $x \bmod 1$, which for instance is useful when implementing function $\mathrm{cospi}(x) = \cos(\pi x)$.
- By requiring x_h and x_ℓ to have $\lfloor p/2 \rfloor$-bit significands (which is always possible in base-2 FP arithmetic, even when p is odd), and doing the same for another FP number y decomposed into y_h and y_ℓ, we ensure that $x_h y_h$, $x_h y_\ell$, $x_\ell y_h$ and $x_\ell y_\ell$ are FP numbers (and hence are computed exactly with FP multiplications). This is the key to the algorithm of Dekker (1971), used in place of 2Prod when no FMA instruction is available (see Section 4.2.2).

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 6 NearestInteger(x)

Require: $2^{p-1} \le C \le 2^p$
$s \leftarrow \mathrm{RN}(C + x)$
$x_h \leftarrow \mathrm{RN}(s - C)$
$x_\ell \leftarrow \mathrm{RN}(x - x_h)$
return (x_h, x_l)

- It can be used to accurately compute the sum of n FP numbers, by splitting them into parts that can be accumulated without errors. An example of such a summation algorithm is given and analysed in detail by Rump *et al.* (2008). See also Sections 3.4 and 5.3.

Like Jeannerod, Muller and Zimmermann (2018), we distinguish between *absolute splittings*, where we require x_h to be a multiple of some constant and $|x_\ell|$ to be less than that constant, and *relative splittings*, where the required bound for x_ℓ is in some sense proportional to $|x|$. Algorithm 6 does an absolute splitting of x, similarly to algorithm `Extractscalar` of Rump *et al.* (2008).

Note that Algorithm 6 is similar to Fast2Sum (Algorithm 1). Only the input assumptions differ somewhat. This algorithm is, for instance, used for reducing the initial argument in the Julia implementation[29] of function `cospi`.

Theorem 4.1 (Jeannerod *et al.* 2018). Assume that C is an integer satisfying $2^{p-1} \le C \le 2^p$. If

$$2^{p-1} - C \le x \le 2^p - C,$$

then the floating-point number x_h returned by Algorithm 6 is an integer such that $|x - x_h| \le 1/2$ (i.e. x_h is equal to x rounded to a nearest integer). Furthermore, $x = x_h + x_\ell$.

Some absolute splitting techniques were also introduced by Rump *et al.* (2008), for scalars and vectors, in the context of accurate summation of many FP numbers. Extensions to the matrix case have then been proposed by Ozaki, Ogita, Oishi and Rump (2012), which allow us to compute EFTs of matrix products.

The next algorithm performs a relative splitting of x and can be seen as an FMA-based variant of Veltkamp's splitting used by Dekker (1971).

Theorem 4.2 (Jeannerod *et al.* 2018). Let $x \in \mathbb{F}$ and $s \in \mathbb{Z}$ such that $1 \le s < p$. Then, barring underflow and overflow, Algorithm 7 computes $x_h, x_\ell \in \mathbb{F}$ such that $x = x_h + x_\ell$ and the significands of x_h and x_ℓ have at most $p - s$ and s bits, respectively.

[29] See https://docs.julialang.org/en/v1/base/math/.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 7 FMAsplit(x)

Require: $C = 2^s + 1$
$\gamma \leftarrow \mathrm{RN}(Cx)$
$x_h \leftarrow \mathrm{RN}(\gamma - 2^s x)$
$x_\ell \leftarrow \mathrm{RN}(x - x_h)$ {or $x_\ell \leftarrow \mathrm{RN}(Cx - \gamma)$}
return (x_h, x_ℓ)

Note that the upper bounds $p - s$ and s on the bit-lengths of x_h and x_ℓ can be attained simultaneously. For example, if $x = 2-2^{1-p}$, then $\gamma = (1+2^{-s}-2^{1-p})\cdot 2^{s+1}$, and therefore

$$x_h = (2^{p-s} - 1) \cdot 2^{s-p+1}, \quad x_\ell = (2^s - 1) \cdot 2^{1-p}.$$

When $s = p - 1$, Theorem 4.2 implies that x_h will be an integral power of 2, but note that $|x_h|$ need not be exactly $\mathrm{ufp}(x)$. Using again the input $x = 2 - 2^{1-p}$, we can check that $x_h = 1 = \mathrm{ufp}(x)$ when $s = p - 1$, while taking $x = 2 - 2^{2-p}$ gives $x_h = 2 \neq \mathrm{ufp}(x)$.

Exact computation of $\mathrm{ufp}(x)$ can be done thanks to an algorithm due to Rump (2009), recalled in Algorithm 8. Rump's algorithm uses four FP operations, without any FMA. Furthermore, if in this algorithm we replace φ with $\varphi \cdot 2^{1-p}$ and ignore the possibility of underflow and overflow, then we can obtain $\mathrm{ulp}(x)$ in four FP operations as well.

Algorithm 8 ufp(x)

Require: $\varphi = 2^{p-1} + 1$, $\psi = 1 - 2^{-p}$
$q \leftarrow \mathrm{RN}(\varphi x)$
$r \leftarrow \mathrm{RN}(\psi q)$
$\delta \leftarrow \mathrm{RN}(q - r)$
return $|\delta|$

If an FMA is available and used, then one can obtain $\mathrm{ulp}(x)$ in just three FP operations, by adapting a scheme introduced by Rump, Zimmermann, Boldo and Melquiond (2009) for computing the predecessor and successor of a given FP number (Jeannerod *et al.* 2018, Algorithm 8). Note also that this paper by Rump *et al.* (2009) provides a good illustration of the care needed to handle subnormal inputs properly.

Finally, let us mention that alternative splitting algorithms have been proposed by Graillat, Lefèvre and Muller (2020) when rounding is assumed to be either towards $-\infty$ or towards $+\infty$ (instead of to nearest). Such algorithms can then be used to implement Dekker's EFT for the product of two FP numbers when directed roundings are used.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

4.4. Floating-point filters

In some cases, despite the use of numerical computations, one might be looking not for a numerical result but for a Boolean result. A typical example is in computational geometry, where decisions such as *are these two points on the same side of this plane?* or *is this point inside this sphere?* must be made. These decisions usually reduce to knowing the sign of a determinant.

If the rounding error of the floating-point determinant can be (hopefully tightly) bounded, then we will in most cases be able to make the decision with a simple floating-point calculation. The use of exact or highly accurate arithmetic will be reserved to the (hopefully few) cases where the absolute value of the computed determinant is less than the error bound. This is the idea behind the use of *floating-point filters*, suggested by Fortune and Van Wyk (1993). See also the works by Shewchuk (1997), Pion (1999), Demmel and Hida (2004) and Bartels, Fisikopoulos and Weiser (2022).

4.5. Error bounds in complex arithmetic

Once floating-point arithmetic is available for approximating and manipulating real numbers, it can naturally be used to create ways to approximate and manipulate complex numbers, and this has been common practice at least since the 1960s (Smith 1962, Wilkinson 1965, Friedland 1967).

Typically, a complex number $z \in \mathbb{C}$ will be approximated by a *complex floating-point number* of the form $\widehat{z} = a + ib$ with $i^2 = -1$ and a, b two FP numbers from the same set $\mathbb{F}$ of binary, precision-p FP numbers. If a and b are obtained by rounding the real and imaginary parts of z to nearest in $\mathbb{F}$, then in the absence of underflow and overflow, each of the two associated relative errors is at most $\mathbf{u} = 2^{-p}$, and therefore

$$\widehat{z} = z(1+\delta), \quad \delta \in \mathbb{C}, \quad |\delta| \le \mathbf{u}.$$

This is just the complex analogue of what we have seen in Section 2.1, the relative error δ now being a complex number whose complex absolute value is bounded by the unit round-off independently of z. Similarly to the real case, $\mathbf{u}$ can be replaced by $\mathbf{u}/(1+\mathbf{u})$ when rounding is to nearest, and by $2\mathbf{u}$ in the case of directed or faithful roundings.[30] Also, the possibility of underflow for a or b is taken into account by adding a suitable complex term to the product $z(1+\delta)$ (Demmel 1984).

Given two complex FP numbers $x = a + ib$ and $y = c + id$, we can now consider how to operate on them. In principle, this should be straightforward at least for basic arithmetic, which is defined by simple real functions of a, b, c, d:[31]

[30] In the case where $|z| = 1$, which occurs in particular when z is a root of unity, then the bounds $\mathbf{u}$ and $2\mathbf{u}$ can be divided by $\sqrt{2}$, as shown by Brisebarre *et al.* (2020).

[31] For square root, choosing a negative real part could be possible as well, provided the sign of the imaginary part is adjusted accordingly; also, we assume that sign(0) = +1. See the work of Kahan (1987, p. 201) for a sign function supporting signed zeros.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

- $|x| = \sqrt{a^2 + b^2}$,
- $x \pm y = a \pm c + i(b \pm d)$,
- $xy = ac - bd + i(ad + bc)$,
- $x/y = (ac + bd)/s + i(bc - ad)/s$ with $s = c^2 + d^2$, and
- $\sqrt{x} = \sqrt{\dfrac{|x| + a}{2}} + i\,\mathrm{sign}(b)\sqrt{\dfrac{|x| - a}{2}}$.

In practice, providing suitable FP support for this set of basic operations is non-trivial, as various issues arise, pertaining notably to specification, robustness and accuracy. These issues are well known, and detailed discussions can be found for example in the writings of Kahan (1987), Kahan and Thomas (1991) and Beebe (2017, Chap. 15, 17), so here we only make a few comments.

Specification. Given some complex FP inputs whose real and imaginary parts can be FP finite numbers as well as infinities or NaNs, what complex FP output should we return for $|x|$, $x + y$, etc.? For the complex absolute value as well as complex addition/subtraction we can simply rely on the IEEE-754 specifications of the hypotenuse function and of addition/subtraction. Note, however, that since the support of the hypotenuse function is not required but only recommended by the Standard (IEEE 2019, §9), most existing implementations of $|x|$ might follow a weaker specification, where a correctly rounded result is not guaranteed.

For complex multiplication, division and square root, there is no such complete specification to rely on, at least because strict requirements for the behaviour of an operation like $ab + cd$ have so far been beyond the scope of IEEE-754 arithmetic. Several implementation choices are thus possible, sometimes leading to inconsistencies for numerical values as well as exceptional behaviours (Demmel *et al.* 2022, §2.2).

Robustness. Another difficulty comes from the possibility of undue intermediate underflows and overflows when evaluating expressions such as $\sqrt{a^2 + b^2}$, $ac - bd$, and $(ac + bd)/(c^2 + d^2)$ in the straightforward way by means of basic FP arithmetic. For example, if $ac \approx bd \gg \mathbf{\Omega}$, then the exact value $ac - bd$ can be in the normal domain of $\mathbb{F}$. In such a case, it could in principle be represented accurately by an element of $\mathbb{F}$, while $\mathrm{R}^{\mathrm{U}}_{\mathrm{D}}(\mathrm{R}^{\mathrm{U}}_{\mathrm{D}}(ac) - \mathrm{R}^{\mathrm{U}}_{\mathrm{D}}(bd)) = \mathrm{R}^{\mathrm{U}}_{\mathrm{D}}(+\infty - (+\infty)) = \mathrm{NaN}$.

To avoid this kind of phenomenon as much as possible, various scaling techniques have been proposed, especially for division and square root (Smith 1962, Friedland 1967, Stewart 1985, Kahan 1987, Li *et al.* 2000, Priest 2004, Baudin and Smith 2012), and some of them appear for example in the working draft of the C Standard (ISO/IEC 2022, Annex G). Also, a scaling technique for multiplication is mentioned by Sterbenz (1974, §13).

Note finally that the availability of a correctly rounded operator $(a, b, c, d) \mapsto ab + cd$ would immediately lead to a robust multiplication; for division, however, inputs such that $ac + bd \approx c^2 + d^2 \gg \mathbf{\Omega}$ would remain problematic.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Accuracy. A third difficulty is due to the fact that damaging cancellation can be caused by straightforward FP evaluations of expressions such as $ac - bd$ or $|x| - |a|$, that appear in the above definitions of xy, x/y and $\sqrt{x}$. For the other operations, namely $|x|$ and $x \pm y$, this is clearly not an issue, and high accuracy is ensured even for naive implementations of the hypotenuse (Hull *et al.* 1994, Ziv 1999), although even better accuracy can be obtained with some care, as shown by Borges (2021).

In the next three subsections we will examine this accuracy issue more carefully, thus focusing on multiplication, division and square root, and see when and how the possibility of heavy cancellation should be dealt with in order to ensure some high level of accuracy. A number of error bounds will be reviewed, which have been given in the literature under the assumption that underflows and overflows do not occur. These bounds, however, remain compatible with the scaling techniques mentioned in the previous paragraph provided the scaling factors are integral powers of two.

The last subsection will be devoted to extensions to the complex case of some of the error-free transformations seen in Section 4.2.

4.5.1. Ensuring normwise accuracy: multiplication and division

For complex multiplication and division, it was noticed early on that the definitions recalled above can be used directly in FP arithmetic to obtain values $\widehat{z}$ such that

$$\widehat{z} = z(1+\delta), \quad z \in \{xy, x/y\}, \quad \delta \in \mathbb{C}, \quad |\delta| \le \lambda \mathbf{u} + O(\mathbf{u}^2) \tag{4.1}$$

for some *modest* constant $\lambda \in \mathbb{R}_{>0}$ (Champagne 1964, Wilkinson 1965). All we need here is that each FP operation is done with relative error at most $\mathbf{u}$ (or even a small multiple of it) and that underflows and overflows do not occur.

In other words, despite the possibility of damaging cancellation when computing the real or imaginary part of the result (resulting in poor *componentwise* accuracy), the traditional expressions for complex multiplication and division are enough to ensure high relative accuracy in the *normwise* sense, for any of the usual rounding functions. In the specific case of round-to-nearest, several detailed error analyses have been done in order to determine the possible values of λ in (4.1) depending on the context (e.g. availability of an FMA). We briefly review some of these results in what follows.

Multiplication and squaring. For multiplication, Brent, Percival and Zimmermann (2007) showed that one can take $\lambda = \sqrt{5}$ when $ac - bd$ and $ad + bc$ are obtained after four FP multiplications and two FP additions. This constant can be decreased to $\lambda = 2$ when an FMA is used to obtain the real and imaginary parts as $\mathrm{RN}(ac - \mathrm{RN}(bd))$ and $\mathrm{RN}(ad + \mathrm{RN}(bc))$, or as any of the three other ways (Jeannerod, Kornerup, Louvet and Muller 2017*a*). In both cases the term $O(\mathbf{u}^2)$ in (4.1) can be removed and the constant is best possible in the sense that there exist FP inputs a, b, c, d yielding a relative error δ such that $|\delta| \sim \lambda \mathbf{u}$ as $\mathbf{u} \to 0$.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

It was also noted by Jeannerod *et al.* (2017*a*) that in the special case of complex squaring, where

$$x^2 = a^2 - b^2 + i \cdot 2ab,$$

one can take $\lambda = 2$ even in the absence of an FMA, that is, when the real part is obtained as $\mathrm{RN}(\mathrm{RN}(a^2) - \mathrm{RN}(b^2))$.

Division and inversion. For division, the definition amounts to first computing the complex product $x\overline{y}$ with $\overline{y} = c - id$, and then dividing the real and imaginary parts of this product by

$$s = y\overline{y} = c^2 + d^2.$$

Evaluating this sum of squares as $\mathrm{RN}(\mathrm{RN}(c^2) + \mathrm{RN}(d^2))$ or $\mathrm{RN}(c^2 + \mathrm{RN}(d^2))$ or $\mathrm{RN}(\mathrm{RN}(c^2) + d^2)$ yields an FP number $\widehat{s}$ of the form

$$\widehat{s} = s(1 + \delta_s), \quad \delta_s \in \mathbb{R}, \quad |\delta_s| \le 2\mathbf{u},$$

and each FP division then yields a relative error at most $\mathbf{u}$. Thus, as noted by Olver (1983), the computed quotient $\widehat{z}$ satisfies (4.1) with

$$\lambda = \lambda_{\mathtt{mul}} + 3, \tag{4.2}$$

where $\lambda_{\mathtt{mul}}$ denotes the constant associated with the algorithm used to evaluate $x\overline{y}$. Consequently, we can always take $\lambda = \sqrt{5} + 3$ and, if an FMA is available, $\lambda = 5$.

An interesting special case is complex inversion, defined by

$$1/y = c/s - id/s.$$

Since no complex multiplication is involved, we can take $\lambda_{\mathtt{mul}} = 0$, and we deduce immediately that $\lambda_{\mathtt{inv}} = 3$. The detailed error analysis by Jeannerod, Louvet, Muller and Plet (2016) reveals that for $p \ge 10$ and without an FMA, we can in fact take $\lambda_{\mathtt{inv}} = 2.70712\ldots$, and that this leading constant is reasonably sharp. For example, for $p = 53$ (binary64 IEEE format), evaluating $z = (c + id)^{-1}$ in this way with $c = 4503599709991314$ and $d = 6369051770002436 \cdot 2^{26}$ produces $\widehat{z}$ such that $|\delta| = |\widehat{z} - z|/|z| = 2.70679\ldots\mathbf{u}$.

Complex inversion can of course also be used to implement complex division as

$$x \cdot y^{-1} = a(c/s) + b(d/s) + i(b(c/s) - a(d/s)).$$

This approach has the same operation count as the initial one, based on $x/y = (ac + bd)/s + i(bc - ad)/s$, but less instruction-level parallelism, since the sum of squares and the complex product can no longer be evaluated simultaneously. However, it has the following nice accuracy property. The computed $\widehat{z}$ satisfies (4.1) with

$$\lambda = \lambda_{\mathtt{inv}} + \lambda_{\mathtt{mul}},$$

which for $\lambda_{\mathtt{inv}} < 2.708$ and $\lambda_{\mathtt{mul}} \le \sqrt{5} < 2.237$ implies $\lambda < 4.945$. Consequently, for complex division we can always take $\lambda = 5$ in (4.1), provided we use x/y if an FMA is available, and $x \cdot y^{-1}$ otherwise.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

If besides the FMA we can also use tests or call functions such as minimumMagnitude and maximumMagnitude (see Section 2.12.2), then the sum of squares can now be evaluated as

$$\widehat{s} = \mathrm{RN}(m^2 + \mathrm{RN}(n^2)), \quad m = \max(|c|, |d|),\ n = \min(|c|, |d|). \tag{4.3}$$

In this case it is known that the associated relative error $|\delta_s|$ can be bounded by $1.5\mathbf{u}$ instead of $2\mathbf{u}$ without sorting $|c|$ and $|d|$. This directly allows us to replace equation (4.2) with $\lambda = \lambda_{\mathtt{mul}} + 2.5$ and therefore to obtain a complex division algorithm for which $\lambda = 4.5$ (Jeannerod, Louvet and Muller 2013*b*).

4.5.2. *Ensuring normwise accuracy: square root*

In contrast to multiplication and division, the formula recalled above for square root does not ensure a normwise relative error in $O(\mathbf{u})$ when evaluated in FP arithmetic. This fact, which is due to the possibility of heavy cancellation in one of the two expressions $|x| \pm a$, was recognized early on, and a now classical workaround is the following rewriting, which for example was already proposed by Strachey (1959). Given $x = a + ib$ with $a, b \in \mathbb{F}$, the real and imaginary parts of $\sqrt{x} = R + iI$ can be expressed as follows:

- if $a \geq 0$, then $R = \sqrt{\dfrac{|x| + a}{2}}$ and $I = b/(2R)$;
- if $a < 0$, then $I = \mathrm{sign}(b)\sqrt{\dfrac{|x| - a}{2}}$ and $R = b/(2I)$.

By using the standard model, Hull *et al.* (1994) showed that in the absence of underflow and overflow, Strachey's formula produces a complex FP number $\widehat{z} = \widehat{R} + i\widehat{I}$ such that

$$\widehat{z} = \sqrt{x}\,(1 + \delta), \quad \delta \in \mathbb{C}, \quad |\delta| \leq \frac{\sqrt{37}}{2}\mathbf{u} + O(\mathbf{u}^2).$$

This bound holds independently of whether an FMA is used for evaluating the sum of squares in $|x| = \sqrt{a^2 + b^2}$, and for directed and faithful roundings it holds with $\mathbf{u}$ replaced by $2\mathbf{u}$.

For round-to-nearest, the term $O(\mathbf{u}^2)$ can be removed and the leading constant $\sqrt{37}/2 = 3.041\ldots$ is reasonably sharp. For example, for $p = 53$ (binary64 IEEE format), if we take $a = 650824205667 \cdot 2^{-52}$ and $b = 4507997673885435 \cdot 2^{-51}$ and evaluate $a^2 + b^2$ as $\mathrm{RN}(\mathrm{RN}(a^2) + \mathrm{RN}(b^2))$ or $\mathrm{RN}(a^2 + \mathrm{RN}(b^2))$, we obtain a relative error δ such that $|\delta| > 3.023\mathbf{u}$ (Jeannerod and Muller 2017). Of course, if one can sort $|a|$ and $|b|$ and evaluate $a^2 + b^2$ with an FMA as in (4.3), then a constant slightly smaller than $\sqrt{37}/2$ is easily established, such as for example $\sqrt{545}/8 = 2.918\ldots < 3$.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

4.5.3. Ensuring componentwise accuracy

We have just seen that for the five basic operations on complex FP numbers the usual definitions (and, in the case of square root, Strachey's rearrangement) suffice to ensure that the associated normwise relative errors $|\widehat{z} - z|/|z|$ can be bounded by small multiples of $\mathbf{u}$. A nice consequence of this is that the error analyses based on the standard model of FP arithmetic can often be extended to the complex case directly, by a suitable increase of the leading constants in the error bounds.

What if we now want high relative accuracy in the *componentwise* sense? In this case, both the real and imaginary parts of $\widehat{z}$ must have relative errors in $O(\mathbf{u})$. For complex addition and subtraction, obtained as $\mathrm{RN}(a \pm c) + i\mathrm{RN}(b \pm d)$, this is obviously true. For complex square root, Strachey's rewriting, introduced in Section 4.5.2 in order to ensure normwise accuracy, turns out to ensure componentwise accuracy as well. Indeed, if $a \geq 0$ (and with or without an FMA), it is easily seen that repeated applications of the standard model lead to

$$\frac{|\widehat{R} - R|}{|R|} \leq \frac{5}{2}\mathbf{u} + O(\mathbf{u}^2), \qquad \frac{|\widehat{I} - I|}{|I|} \leq \frac{7}{2}\mathbf{u} + O(\mathbf{u}^2).$$

For $a < 0$, these two bounds can be swapped and, again, when rounding is to nearest they are reasonably tight and the terms $O(\mathbf{u}^2)$ are not even needed (Hull *et al.* 1994, Jeannerod and Muller 2017).

We are thus left with complex multiplication and division, for which the only issue is the accurate FP evaluation of expressions of the form $ac - bd$. Specifically, if $abcd \geq 0$ then $\mathrm{sign}(ac) = \mathrm{sign}(bd)$ and $\mathrm{sign}(ad) = \mathrm{sign}(bc)$, which implies that only the real part in the complex product $ac - bd + i(ad + bc)$ can suffer from heavy cancellation and that its imaginary part will be obtained with high relative accuracy. Similarly, if $abcd < 0$ then only $ad + bc$ can be inaccurate. These remarks apply to division as well, where inaccuracy can only come from the FP evaluation of one of the numerators in $(ac + bd)/(c^2 + d^2) + i(bc - ad)/(c^2 + d^2)$.

In practice, a naive FP evaluation of $ac - bd$ can indeed yield totally wrong results, with or without an FMA. This fact is well known and we simply recall here a binary64 example by Jeannerod, Monat and Thévenoux (2017*b*). For $p = 53$, if $a = 1 + 2^{-51}$, $b = 1 + 3 \cdot 2^{-52}$, $c = 1 - 2^{-53}$ and $d = 1 - 3 \cdot 2^{-53}$, then the exact result is

$$ac - bd = 7 \cdot 2^{-105} \approx 1.72 \times 10^{-31}$$

and can be represented exactly as a binary64 FP number, while

$$\begin{aligned} \mathrm{RN}(\mathrm{RN}(ac) - \mathrm{RN}(bd)) &= 0, \\ \mathrm{RN}(ac - \mathrm{RN}(bd)) &= 1.11\ldots \times 10^{-16}, \\ \mathrm{RN}(\mathrm{RN}(ac) - bd) &= -1.11\ldots \times 10^{-16}. \end{aligned}$$

An efficient way to avoid such damaging cancellations is to call the 2Prod algorithm from Section 4.2.2. For example, it implements the EFT $ac = \pi + e$,

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 9 cmulCHT(a, b, c, d)

$(\pi, e) \leftarrow 2\mathrm{Prod}(a, c)$
$(\pi', e') \leftarrow 2\mathrm{Prod}(b, d)$
$\widehat{R} \leftarrow \mathrm{RN}(\mathrm{RN}(\pi - \pi') + \mathrm{RN}(e - e'))$
$(\rho, f) \leftarrow 2\mathrm{Prod}(a, d)$
$(\rho', f') \leftarrow 2\mathrm{Prod}(b, c)$
$\widehat{I} \leftarrow \mathrm{RN}(\mathrm{RN}(\rho + \rho') + \mathrm{RN}(f + f'))$
return $(\widehat{R}, \widehat{I})$

Algorithm 10 cmulKahan(a, b, c, d)

$(\pi, e) \leftarrow 2\mathrm{Prod}(a, c)$
$\widehat{R} \leftarrow \mathrm{RN}(\mathrm{RN}(\pi - bd) + e))$
$(\rho, f) \leftarrow 2\mathrm{Prod}(a, d)$
$\widehat{I} \leftarrow \mathrm{RN}(\mathrm{RN}(\rho + bc) + f))$
return $(\widehat{R}, \widehat{I})$

where π is the rounded product $\mathrm{RN}(ac)$ and e is the corresponding error. Adding this error to the error of the product $\mathrm{RN}(bd)$ yields a correction that can then be used to improve the accuracy of the naive evaluation $\mathrm{RN}(\mathrm{RN}(ac) - \mathrm{RN}(bd))$. This scheme is an example of what is sometimes called a *compensated algorithm* (see Section 5.3). It was presented and analysed by Cornea, Harrison and Tang (2002, p. 273), who used it to evaluate both the real and imaginary parts R and I of $(a + ib)(c + id)$ very accurately. For round-to-nearest, this leads to Algorithm 9.

Detailed rounding error analyses of this approach show that both $|\widehat{R}/R - 1|$ and $|\widehat{I}/I - 1|$ are bounded by $2\mathbf{u} + O(\mathbf{u}^2)$, that the constant 2 is best possible, and that the possibility of removing the term in $O(\mathbf{u}^2)$ depends on the tie-breaking rule. If $\mathrm{RN} = \mathrm{RN}_e$ (ties-to-even), then $2\mathbf{u} + O(\mathbf{u}^2)$ can be replaced by $2\mathbf{u}$. But if $\mathrm{RN} = \mathrm{RN}_a$ (ties-to-away), the relative error in $\widehat{R}$ or $\hat{I}$ can be larger than $2\mathbf{u} + \mathbf{u}^2 - 4\mathbf{u}^3$ (Muller 2015, Jeannerod 2016).

An alternative to Algorithm 9 is Algorithm 10, which, following a technique by Kahan (1998), recovers the error of only one of the two products in $ac - bd$, and similarly for $ad + bc$. In the version displayed here the computed errors are those of $\mathrm{RN}(ac)$ and $\mathrm{RN}(ad)$, but three other ways are possible. In each case the resulting relative errors are bounded as

$$|\widehat{R}/R - 1| \le 2\mathbf{u}, \quad |\widehat{I}/I - 1| \le 2\mathbf{u}$$

and, again, the constant 2 cannot be reduced further (Jeannerod, Louvet and Muller 2013*a*).

In practice, these accurate algorithms have reasonable run-time overheads compared with the naive ones, for which the computed real or imaginary part may have no correct digit at all (Jeannerod *et al.* 2017*b*). Although Algorithm 10 is simpler and cheaper than Algorithm 9, the latter has the advantage of symmetry

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

and ensures that the complex multiplication it implements is commutative. Note also that both algorithms preserve the fact that $x\overline{x} = (a + ib)(a - ib) = a^2 + b^2$ is a real number; this is well known to be false for the naive FMA-based scheme, where the imaginary part is computed as $\mathrm{RN}(\mathrm{RN}(-ab) + ab)$ or $\mathrm{RN}(-ab + \mathrm{RN}(ab))$.

Application to division. Algorithms 9 and 10 can be used directly to ensure high componentwise accuracy for complex division. It suffices to evaluate the complex product $x\overline{y} = (a + ib)(c - id)$ with either of these algorithms, and then to divide the obtained real and imaginary parts by $y\overline{y} = c^2 + d^2$. Each component of the computed quotient then has relative error at most $5\mathbf{u} + O(\mathbf{u}^2)$ and, similarly to what has been said for the normwise case, evaluating $c^2 + d^2$ as in (4.3) leads further to the slightly smaller bound $4.5\mathbf{u} + O(\mathbf{u}^2)$ (Jeannerod *et al.* 2013*b*). As for complex inversion, the traditional formula $1/y = c/s - i \cdot d/s$ already ensures high componentwise accuracy.

The special case of squaring. To ensure componentwise accuracy when evaluating $x^2 = a^2 - b^2 + i \cdot 2ab$, the preceding algorithms need not be used and it suffices to evaluate $a^2 - b^2$ as $(a + b)(a - b)$. This rewriting was already suggested by Kahan (1987) and, for round-to-nearest, it is easily checked that the computed value

$$\widehat{R} = \mathrm{RN}(\mathrm{RN}(a + b) \cdot \mathrm{RN}(a - b))$$

satisfies $|\widehat{R}/R - 1| \leq 3\mathbf{u}$. The analysis done by Jeannerod (2020) shows further that the best possible constant is indeed 3 when RN rounds ties-to-away, but that it is $9/4$ when RN rounds ties-to-even. It was also noted there that $\widehat{R}$ can be strictly larger than $\mathrm{RN}(a^2)$, while this cannot occur when using any of the naive schemes $\mathrm{RN}(\mathrm{RN}(a^2) - \mathrm{RN}(b^2))$, $\mathrm{RN}(a^2 - \mathrm{RN}(b^2))$ or $\mathrm{RN}(\mathrm{RN}(a^2) - b^2)$, thanks to the monotonicity of the rounding function.

4.5.4. Complex EFTs

Extensions of error-free transformations from the real case to the complex case have been proposed by Graillat and Ménissier-Morain (2007, 2008, 2012) for addition, subtraction and multiplication, mostly in order to improve the normwise accuracy of polynomial evaluation in complex FP arithmetic.

For addition, where $x + y = (a + ib) + (c + id)$ is evaluated as the complex FP number $\widehat{z} = \mathrm{RN}(a + c) + i\mathrm{RN}(b + d)$, a complex EFT is obtained directly by applying the EFT from Section 4.2.1 twice, to $a + c$ and to $b + d$. Two calls to the 2Sum algorithm suffice to produce the FP numbers $f = a + c - \mathrm{RN}(a + c)$ and $g = b + d - \mathrm{RN}(b + d)$ and therefore the complex FP number e such that

$$x + y = \widehat{z} + e, \quad e = f + ig, \quad f, g \in \mathbb{F}. \tag{4.4}$$

We can proceed similarly for subtraction, and the remarks made in the real case still apply: rounding must be to nearest, 2Sum can be replaced by Fast2Sum up to sorting the inputs, and the (im)possibility of spurious overflow is well understood.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

For multiplication, the most common EFT is obtained by considering the evaluation of $xy = (a + ib)(c + id)$ without FMA, that is, as $\widehat{z} = \widehat{R} + i\widehat{I}$ where $\widehat{R} = \mathrm{RN}(\mathrm{RN}(ac) - \mathrm{RN}(bd))$ and $\widehat{I} = \mathrm{RN}(\mathrm{RN}(ad) + \mathrm{RN}(bc))$, and by using not only 2Sum but also the FMA-based 2Prod algorithm from Section 4.2.2 to compute the associated rounding errors. Specifically, for the real part we have

$$\begin{aligned} ac - bd &= \mathrm{RN}(ac) + f_1 - \mathrm{RN}(bd) + f_2 \\ &= \widehat{R} + f_1 + f_2 + f_3, \end{aligned}$$

where the errors $f_1, -f_2 \in \mathbb{F}$ are computed by 2Prod and the error $f_3 \in \mathbb{F}$ is computed by 2Sum. Proceeding similarly for the imaginary part yields three FP errors g_k such that $ad + bc = \widehat{I} + g_1 + g_2 + g_3$, and therefore three complex FP numbers e_k such that

$$xy = \widehat{z} + e_1 + e_2 + e_3, \quad e_k = f_k + ig_k, \quad f_k, g_k \in \mathbb{F}. \tag{4.5}$$

Depending on whether Fast2Sum or 2Sum is used, the costs of decomposing $x + y$ as in (4.4) and xy as in (4.5) range from, respectively, 6 to 12 and 14 to 20 FP operations. In practice, these EFTs have been used to improve the evaluation of elementary symmetric functions on complex FP inputs (Jiang, Graillat, Barrio and Yang 2016), as well as the computation of polynomial roots via the Ehrlich–Aberth method (Cameron and Graillat 2022).

Note finally that in the absence of an FMA it is still possible to implement the EFT in (4.5) by using the Veltkamp–Dekker algorithm (mentioned in Section 4.2.2) in order to recover the rounding errors of the four FP products, but at the price of at least twice as many FP operations as with 2Prod. If, on the contrary, an FMA is available and used to evaluate the product xy as $\widehat{z}' = \widehat{R}' + i\widehat{I}'$ with, say, $\widehat{R}' = \mathrm{RN}(\mathrm{RN}(ac) - bd)$ and $\widehat{I}' = \mathrm{RN}(\mathrm{RN}(ad) + bc)$, then we can obtain an EFT of the same form as in (4.5),

$$xy = \widehat{z}' + e_1 + e_2' + e_3',$$

with e_1 produced as before by two calls to 2Prod, and with e_2' and e_3' two other complex FP numbers now produced by two calls to the EFT of the FMA (algorithm ErrFMA from Section 4.2.4). Again, a significant cost overhead is to be expected here, due to the complexity of ErrFMA. Thus, although such variants might be optimized further, it seems that a reasonable choice for performing an EFT for complex multiplication remains the one based on (4.5), restricting the use of the FMA to 2Prod.

4.6. *Error bounds for higher-dimensional problems*

In the previous section we focused on complex FP arithmetic, whose key building block is the two-dimensional inner product $ab + cd$. We now consider similar problems in higher dimensions, such as $x^\top y = \sum_{i=1}^n x_i y_i$ with x and y two vectors of n FP numbers. As before, our focus will be on accuracy issues and the derivation

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

of error bounds, assuming (unless otherwise stated) that only one precision is used and that underflows and overflows do not occur.

4.6.1. Traditional worst-case bounds

Following Wilkinson (1960, 1963), *a priori* worst-case error bounds can often be produced by repeated applications of the standard models of FP arithmetic (2.5): (ratios of) expressions of the form

$$1 + \theta_h := \prod_{i=1}^{h} (1 + \delta_i) \tag{4.6}$$

are obtained, from which bounds on $|\theta_h|$ imply bounds on the backward error of the computed result. Here, each δ_i is the relative error of a single FP arithmetic operation, and h depends on the algorithm used to produce that result. For example, when evaluating the sum $x_1 + x_2 + x_3$ from left to right, the computed result $\widehat{s}$ can be written as

$$\begin{aligned}\widehat{s} &= ((x_1 + x_2)(1 + \delta_2) + x_3)(1 + \delta_1)\\ &=: x_1' + x_2' + x_3',\end{aligned}$$

where $x_1' := x_1(1+\theta_2)$, $x_2' := x_2(1+\theta_2)$ and $x_3' := x_3(1+\theta_1)$. In other words, $\widehat{s}$ is the exact sum of three numbers obtained by slight (relative) perturbations $(\theta_2, \theta_2, \theta_1)$ of (x_1, x_2, x_3), and its (relative) backward error is at most $\max(|\theta_1|, |\theta_2|)$.

For round-to-nearest, where $|\delta_i| \le \mathbf{u}/(1 + \mathbf{u}) \le \mathbf{u}$ for all i, we have

$$|\theta_h| \le (1 + \mathbf{u})^h - 1;$$

for directed roundings and faithful rounding the same holds with $\mathbf{u}$ replaced by $2\mathbf{u}$. Assuming h is fixed and $u \to 0$, we see that these bounds have the form $\lambda\mathbf{u} + O(\mathbf{u}^2)$ for some 'constant' λ that is either h or $2h$, depending on the rounding function.

The commonest way to bound $|\theta_h|$, however, is by replacing $(1 + \mathbf{u})^h - 1$ with the following slightly larger quantity:

$$|\theta_h| \le \frac{h\mathbf{u}}{1 - h\mathbf{u}} =: \gamma_h \quad \text{if } h < \mathbf{u}^{-1}. \tag{4.7}$$

As amply demonstrated by Higham (2002), this γ_h notation turns out to be the right tool for performing a great deal of *a priori* worst-case rounding error analysis. Indeed, it makes it possible to derive error bounds that are concise yet rigorous and free of the risk of having possibly large constants hiding in the $O(\mathbf{u}^2)$ terms. Furthermore, such bounds can be composed very conveniently by means of simple manipulation rules such as $\gamma_h + \gamma_k + \gamma_h\gamma_k \le \gamma_{h+k}$ if $h + k < \mathbf{u}^{-1}$ (Higham 2002, §3.4).

Note that when deriving a backward error bound of the form γ_h, the precise value of h can depend (sometimes significantly) on the FP algorithm used. For example, when evaluating a degree-n polynomial $\sum_{i=0}^{n} a_i x^i$ by Horner's rule, then, for $a_i, x \in \mathbb{F}$ and using n multiplications and n additions with round-to-nearest, the

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

resulting backward error is at most γ_{2n}. But if an FMA is used at each of the n iterations, this bound drops to γ_n and can thus be roughly halved.

As another classical example, consider the FP evaluation of the sum

$$s_n = x_1 + \cdots + x_n, \quad x_i \in \mathbb{F},$$

using exactly $n-1$ FP additions and any parenthesizing. For $n = 4$, this could be $((x_1 + x_2) + x_3) + x_4$ or $(x_1 + (x_2 + x_3)) + x_4$ or $(x_1 + x_2) + (x_3 + x_4)$, etc. In the absence of overflow[32] the computed sum $\widehat{s}_n$ has backward error bounded by γ_h, where $h \le n-1$ is the height of the binary tree that underlies the chosen summation algorithm (Higham 2002, §4.2). Thus the forward error of $\widehat{s}_n$ can be bounded as follows, independently of the summation ordering:

$$|\widehat{s}_n - s_n| \le \gamma_{n-1} \sum_{i=1}^{n} |x_i| \quad \text{if } n-1 < \mathbf{u}^{-1}. \tag{4.8}$$

This constant γ_{n-1} corresponds to *recursive summation*, where $\widehat{s}_n$ is produced by the recurrence defined by $\widehat{s}_1 := x_1$ and $\widehat{s}_k := \mathrm{RN}(\widehat{s}_{k-1} + x_k)$ for $2 \le k \le n$. At the other extreme, *pairwise summation* corresponds to the smallest possible height

$$h = \lceil \log_2 n \rceil.$$

An interesting intermediate situation is *blocked summation*, which underlies the design and implementation of various efficient and accurate algorithms for summation and inner products (Castaldo, Whaley and Chronopoulos 2009, Blanchard, Higham and Mary 2020). Given a block size b (which for simplicity is assumed here to divide n), we rewrite the exact sum s_n as $\sum_{j=1}^{n/b} \sum_{k=1}^{b} x_{(j-1)b+k}$ and evaluate it as follows. First, compute each of the n/b partial sums of length b, using a total of $n/b \cdot (b-1)$ FP additions; then compute the sum of the obtained partial sums using $n/b - 1$ FP additions. In this case, the term γ_{n-1} in (4.8) can always be replaced by γ_h with

$$h = b + n/b - 2, \tag{4.9}$$

whose minimal value $\approx 2\sqrt{n}$ is attained when b is nearest to $\sqrt{n}$.

Such variations in the values of γ_h are in general reflected directly by the error bounds of higher-level algorithms relying on FP summation, such as those for inner products, matrix–vector multiplication, matrix multiplication and factorization, etc. In particular, when evaluating inner products $x^\top y = \sum_{i=1}^{n} x_i y_i$ in the usual way by adding the individual products together, the resulting constant is $\gamma_{h+1} \le \gamma_n$, with $h \le n-1$ the height of the underlying summation tree. Finally, any of these traditional error bounds can be extended to the case of *complex FP arithmetic* by simply taking into account the small constants associated with complex FP $\times$, $/$, $\sqrt{}$ (and which were presented in Section 4.5).

[32] Recall that FP addition is exact in the case of underflow, so all the error bounds we shall give for FP summation actually need only assume that overflows do not occur.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

4.6.2. Refining some worst-case bounds

During the last decade or so, several of the traditional bounds mentioned in the previous section have been sharpened and simplified. The initial improvement is due to Rump (2012), who showed for recursive summation and round-to-nearest that the constant γ_{n-1} in (4.8) can be replaced by $(n-1)\mathbf{u}$, for all n. In other words, both the $O(\mathbf{u}^2)$ term in $\gamma_{n-1} = (n-1)\mathbf{u} + O(\mathbf{u}^2)$ and the assumption $(n-1)\mathbf{u} < 1$ can be removed.

The proof of this result does not use just the standard model (2.5a) but combines it with the following property of round-to-nearest:

$$|\mathrm{RN}(a+b) - (a+b)| \le \min(|a|, |b|) \quad \text{for } a, b, \in \mathbb{F}. \tag{4.10}$$

Furthermore, the proof can be adapted to any summation ordering, so that (4.8) can eventually be replaced by

$$|\widehat{s}_n - s_n| \le (n-1)\mathbf{u} \sum_{i=1}^{n} |x_i|. \tag{4.11}$$

Then, still assuming round-to-nearest, several similar refinements have been proposed by various authors, leading to $O(\mathbf{u}^2)$-free error bounds for several basic FP algorithms, including the evaluation of inner products, 2-norms, powers and continued products, the evaluation of univariate polynomials by means of Horner's rule, LU and Cholesky factorizations, and triangular system solving by substitution (Rump 2019, §III and the references therein). In all cases, the classical constants γ_h have been replaced by their linear term $h\mathbf{u}$, but sometimes at the price of some restriction on the largest possible values of h.

More recently, Lange and Rump (2017, 2019) introduced a general arithmetic framework for FP summation, which in particular makes it possible to go beyond (4.11) in at least three ways, by considering roundings other than to nearest, by incorporating the height of the summation tree into the error bounds, and by addressing the issue of the attainability of these bounds. Specifically, they gave the following analogue of (4.11) for more general roundings, such as faithful rounding (and thus also directed roundings), for which property (4.10) no longer holds.

Theorem 4.3 (Lange and Rump 2017). Let $s_n = \sum_{i=1}^{n} x_i$ with $x_i \in \mathbb{F}$. Then, for faithful rounding and any summation ordering, the computed sum $\widehat{s}_n$ satisfies

$$|\widehat{s}_n - s_n| \le (n-1) \cdot 2\mathbf{u} \sum_{i=1}^{n} |x_i| \quad \text{if } n-1 \le \tfrac{1}{2}\mathbf{u}^{-1}.$$

This bound has the same form as the one in (4.11), with $\mathbf{u}$ replaced by $2\mathbf{u}$, up to some restriction on n. Note that in general this restriction on n cannot be relaxed (and is thus unavoidable). For example, for recursive summation and rounding towards $+\infty$, taking $(x_1, x_2, \ldots, x_n) = (1, \mathbf{u}^2, \ldots, \mathbf{u}^2)$ with $n = \frac{1}{2}\mathbf{u}^{-1} + 2$ leads to $\widehat{s}_n - s_n > (n-1) \cdot 2\mathbf{u}s_n$.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Lange and Rump then introduced some $O(\mathbf{u}^2)$-free error bounds that not only cover general roundings but also incorporate the height h of the summation tree, as follows.

Theorem 4.4 (Lange and Rump 2019). Let $s_n = \sum_{i=1}^n x_i$ with $x_i \in \mathbb{F}$. Then, for any summation ordering whose underlying binary tree has height at most h, the computed sum $\widehat{s}_n$ satisfies

$$|\widehat{s}_n - s_n| \le h\mathbf{v} \sum_{i=1}^n |x_i| \quad \text{if } h \le \mathbf{v}^{-1/2} - 1,$$

where $\mathbf{v} = \mathbf{u}$ for round-to-nearest and $\mathbf{v} = 2\mathbf{u}$ for faithful rounding.

Note that here the height $h \le n-1$ is required to be at most of order $1/\sqrt{\mathbf{u}}$, while in the previous theorem n was required to be at most about $1/\mathbf{u}$. Importantly, this error bound holds for inner products as well, with $h \le n$ now being the height of the associated binary evaluation tree (Lange and Rump 2019, Corollary 4).

Finally, for round-to-nearest, the bound (4.11) can be replaced by the following attainable one, which holds whatever the ordering, provided n is not too large.

Theorem 4.5 (Lange and Rump 2019). Let $s_n = \sum_{i=1}^n x_i$ with $x_i \in \mathbb{F}$. Then, for round-to-nearest and any summation ordering, the computed sum $\widehat{s}_n$ satisfies

$$|\widehat{s}_n - s_n| \le \frac{(n-1)\mathbf{u}}{1+(n-1)\mathbf{u}} \sum_{i=1}^n |x_i| \quad \text{if } n-1 \le \tfrac{1}{2}\mathbf{u}^{-1}.$$

This bound is attained in particular for round-to-nearest, ties-to-even, when applying recursive summation to $(x_1, x_2, \ldots, x_n) = (1, \mathbf{u}, \ldots, \mathbf{u})$. Note that the same bound was proposed by Mascarenhas (2016) with a different proof technique and assuming recursive summation and $n \le \frac{1}{20}\mathbf{u}^{-1}$. Interestingly, no such attainable bound seems to be available for FP inner products. Despite this, and as shown by Lange and Rump (2019, §6), this optimal bound for FP summation in round-to-nearest is already enough to deduce rather directly several $O(\mathbf{u}^2)$-free bounds for higher-level problems such as, for example, sums of products $\sum_i \prod_j x_{ij}$, matrix–vector products for Vandermonde matrices, and blocked summation. In the latter case, this makes it possible to replace the traditional constant $\gamma_{b+n/b-2}$, seen in (4.9), with the linear term $(b + n/b - 2)\mathbf{u}$ provided $\max(b, n/b) - 1 \le \frac{1}{2}\mathbf{u}^{-1}$.

4.6.3. Computable error bounds

All the error bounds seen so far are real numbers involving exact expressions, such as $\sum_{i=1}^n |x_i|$, that cannot be represented exactly as FP numbers. If an FP error bound must be produced along with the approximate result, then a first way is via repeated applications of the second standard model (2.5b) instead of the first one. For example, for FP addition and round-to-nearest, this second model implies

$$|a+b| \le (1+\mathbf{u})|\mathrm{RN}(a+b)| \quad \text{for } a, b, \in \mathbb{F}. \tag{4.12}$$

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Applying this inequality $n-1$ times to the sum of the $|x_i|$ gives

$$\sum_{i=1}^{n} |x_i| \le (1+\mathbf{u})^{n-1}\widehat{t}_n,$$

where $\widehat{t}_n$ denotes the computed value of $\sum_{i=1}^{n} |x_i|$ obtained using recursive summation (or another ordering). Hence, recalling (4.11), the error of the computed sum of the x_i is bounded as

$$|\widehat{s}_n - s_n| \le (n-1)\mathbf{u}(1+\mathbf{u})^{n-1}\widehat{t}_n.$$

Here the right-hand side is in general not in $\mathbb{F}$, but it can be bounded by a computable FP number as follows. Let $f := (n-1)\mathbf{u}$ and $g := \mathrm{RN}(\widehat{t}_n/(1-n\mathbf{u}))$, and assume $n \le \mathbf{u}^{-1}$. It is easily seen that f and $1-n\mathbf{u}$ are FP numbers, and that $(1+\mathbf{u})^{n-1}\widehat{t}_n \le g$. Consequently,

$$\begin{aligned} |\widehat{s}_n - s_n| &\le fg \\ &\le (1+\mathbf{u})\mathrm{RN}(fg) \\ &\le \mathrm{RN}\left(\frac{\mathrm{RN}(fg)}{1-2\mathbf{u}}\right) =: \text{computable bound.} \end{aligned}$$

To summarize, a *rigorous* error bound for FP summation can be obtained easily in round-to-nearest FP arithmetic, by first evaluating the two sums $\sum_i x_i$ and $\sum_i |x_i|$ independently and then performing three extra FP operations. Similar examples of computable FP bounds can be found in various contexts and we refer to the works by Ogita *et al.* (2005), Langlois and Louvet (2007), Jiang *et al.* (2016) and Cameron and Graillat (2022).

Another way is to exploit ufp-based bounds instead of the second standard model (2.5b). For example, for FP addition,

$$|\mathrm{RN}(a+b) - (a+b)| \le \mathbf{u}\,\mathrm{ufp}(\mathrm{RN}(a+b)) \quad \text{for } a, b, \in \mathbb{F}.$$

This bound can be up to about half the size of the bound $\mathbf{u}|\mathrm{RN}(a+b)|$ implied by (2.5b). Furthermore, it is a computable bound, since the ufp of an FP number can be obtained using no more than four FP operations in round-to-nearest (Rump 2009, Algorithm 3.5); see also Section 4.3.

For the summation of n FP numbers with round-to-nearest, this bound can be exploited to obtain the following computable counterpart of (4.11).

Theorem 4.6 (Rump 2015). Let $s_n = \sum_{i=1}^{n} x_i$ with $x_i \in \mathbb{F}$, and let $\widehat{s}_n$ and $\widehat{t}_n$ be the computed sums of s_n and $\sum_{i=1}^{n} |x_i|$, in round-to-nearest and with the same ordering. Then $\widehat{s}_n$ satisfies

$$|\widehat{s}_n - s_n| \le (n-1)\,\mathbf{u}\,\mathrm{ufp}(\widehat{t}_n). \tag{4.13}$$

Barring underflow and overflow and for $n \le \mathbf{u}^{-1}$, this bound is an FP number, being the exact product of an FP number and an integral power of 2; as with the

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

first approach, it can be computed by evaluating both $\widehat{s}_n$ and $\widehat{t}_n$ and performing a few extra FP operations, using only round-to-nearest.

Rump (2015) notes that the bound (4.13) is attained for recursive summation, round-to-nearest, ties-to-even, and $(x_1, x_2, \ldots, x_n) = (1, \mathbf{u}, \ldots, \mathbf{u})$. He shows also that in order to achieve (4.13) it is necessary to use the same ordering for both $\widehat{s}_n$ and $\widehat{t}_n$, and that different orderings lead to the more pessimistic bound

$$|\widehat{s}_n - s_n| \le (n-1)\mathbf{u}\big(\widehat{t}_n + (n-1)\mathbf{u}\,\mathrm{ufp}(\widehat{t}_n)\big).$$

As before, a few FP operations will suffice to turn this bound into a computable one. From this, one can for example deduce computable error bounds for matrix multiplication, which can then be used to implement efficient routines for interval matrix multiplication (Ozaki, Ogita and Mukunoki 2021).

4.6.4. *Probabilistic error bounds*

For very high-dimensional problems and very low precision – a situation becoming more and more common with the increase of computing speed and the availability of small FP formats – products like $n\mathbf{u}$ can easily become larger than 1, thus making backward error bounds of the form $\gamma_n = n\mathbf{u}/(1 - n\mathbf{u})$ or $n\mathbf{u}$, such as those seen in the three previous subsections, totally uninformative.

These bounds, however, are worst-case bounds and can be attained or nearly attained only in extremely rare situations. In practice, for fixed $\mathbf{u}$ and increasing n, it has long been observed that the error growth tends to be much slower, typically as $\sqrt{h}\,\mathbf{u}$ rather than as $h\,\mathbf{u}$ (Wilkinson 1961, p. 318).

Recently, Higham and Mary (2019, 2020) initiated a path aimed at formalizing this rule of thumb, and obtained rigorous probabilistic analogues of (4.7), that is, bounds of the form

$$|\theta_h| \le \widetilde{\gamma}_h := \exp\left(\frac{\lambda\sqrt{h}\,\mathbf{u} + h\,\mathbf{u}^2}{1 - \mathbf{u}}\right) - 1,$$

which are proved to hold with probability at least, say, $1 - 2\exp(-\lambda^2/2)$. Here λ is a small constant that can be chosen so that the bound holds with very high probability and the new term $\widetilde{\gamma}_h = \lambda\sqrt{h}\,\mathbf{u} + O(\mathbf{u}^2)$ indeed grows like $\sqrt{h}\,\mathbf{u}$. For stochastic rounding, such probabilistic bounds hold unconditionally, while for deterministic roundings (such as those of IEEE-754 arithmetic), they follow from assumptions made about rounding errors and, sometimes, input data as well. For example, it can be assumed that all the individual rounding errors occurring during a computation form a sequence of mean-independent random variables of mean zero. In practice this framework yields *a priori* error bounds whose growth rates nicely match the observed ones.

Since 2019 the literature on this topic has been growing fast, with the publication of probabilistic error analyses for most of the fundamental building blocks such as sums, inner products and matrix–vector products, matrix multiplication and LU-based system solving. We refer in particular to the works of Ipsen and Zhou (2020)

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

and Connolly, Higham and Mary (2021), and the surveys of Croci *et al.* (2022) and Higham and Mary (2022). Such analyses have also just been proposed for Horner polynomial evaluation (El Arar, Sohier, de Oliveira Castro and Petit 2022), for Householder QR factorization (Connolly and Higham 2022), and in the context of mixed-precision algorithms (Hallman and Ipsen 2022, Connolly, Higham and Pranesh 2022).

Finally, as noted by Higham (2021*b*), these new $O(\sqrt{h}\,\mathbf{u})$ bounds can be combined with blocking techniques (which yield smaller constants as well, as in (4.9)) and with the features of modern architectures (whose block FMA units allow for accumulation in extended precision), in order to explain why huge computations using mostly small FP formats can succeed in practice.

4.7. Tools for polynomial approximation of functions

In general, at the hardware level, the only arithmetic operations that are available are addition, multiplication, division and square root. Since division and square root are significantly slower than the other operations (see Figure 1.1), if we avoid using them, the only functions of one variable we can compute are piecewise polynomials. Hence the mathematical functions are very often approximated by polynomials (Cody and Waite 1980, Muller 2016, Beebe 2017). It is therefore important to be able to tightly control the accuracy of polynomial approximations, as well as the accuracy with which we evaluate polynomials in floating-point arithmetic.

4.7.1. Sollya

The Sollya package[33] by Chevillard, Joldeş and Lauter (2010) provides useful tools for designing mathematical function libraries in floating-point arithmetic. Among the many features of Sollya, two are of particular interest.

- A certified infinite norm (function `supnorm`), which allows us to compute safe approximation error bounds, using an algorithm presented by Chevillard, Harrison, Joldeş and Lauter (2011).
- A special floating-point minimax procedure (function `fpminimax`), that computes best or nearly best polynomial approximations to functions given some constraints (such as the coefficients of the polynomial being floating-point numbers of a given format, or double-word numbers: see Section 5.1), using techniques presented by Brisebarre and Chevillard (2007).

Let us give a short example of a Sollya session. We wish to approximate $\sin(x)$, for $x \in [0, 1/128]$, by an odd polynomial of degree 9 whose degree-1 coefficient fits in one bit (which is just a way of imposing the fact that the coefficient should be equal to 1), whose degree-3 coefficient is a double-word number in binary64

[33] https://www.sollya.org/

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

arithmetic, and whose coefficients of degrees 5, 7 and 9 are binary64 FP numbers. Under these constraints, we wish to minimize the relative approximation error. The command line is:

```
> P := fpminimax(sin(x),[|1,3,5,7,9|],[|1,DD,D,D,D|],[0;1/128],relative);
```

Let us print the coefficients of P (in the form of an integer times a power of 2, so that we have an exact representation of them):

```
> display=powers;
> P;
x * (1 + x^2 * (-1081728512194755755943845273242 29 * 2^(-109)
+ x^2 * (4803839602528529 * 2^(-59) + x^2 * (-7320136537069203 * 2^(-65)
+ x^2 * (3253360761268093 * 2^(-70))))))
```

We then obtain a tight enclosure of the relative approximation error (now printed in decimal) as follows:

```
> prec=100!;
> display=decimal;
> supnorm(P,sin(x),[0;1/128],relative,2^(-20));
[4.138945864885702172513028063427e-30;4.138949688741871499318123375 54e-30]
```

In recent years Sollya has frequently been used by the developers of mathematical function libraries; see for example the work of Sibidanov *et al.* (2022).

4.7.2. Gappa

The Gappa tool[34] is dedicated to the formal verification of properties related to floating-point arithmetic, e.g. bounds on round-off errors (de Dinechin, Lauter and Melquiond 2011, Melquiond 2019). Let us illustrate the tool on the previous polynomial approximation. For conciseness, we will focus on bounding only the round-off error for the tail of the polynomial, i.e. the computation of

```
4803839602528529 * 2^(-59) + x^2 * (-7320136537069203 * 2^(-65)
+ x^2 * (3253360761268093 * 2^(-70))).
```

First we need to provide the tool with the values of the polynomial coefficients:

```
c5 = 4803839602528529b-59;
c7 = -7320136537069203b-65;
c9 = 3253360761268093b-70;
```

Then we can describe the various floating-point computations. For example, `float<ieee_64,ne>(x * x)` is the result of first computing the exact product of `x` by itself and then rounding to binary64 in round-to-nearest (ties-to-even). Since this rounding operator will be used more than once, let us give it the shorter name `rnd`:

```
@rnd = float<ieee_64,ne>;
x2 = rnd(x * x);
```

[34] https://gappa.gitlabpages.inria.fr/

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

We can describe the whole polynomial computation in the same way. But since `rnd` has to be applied around every arithmetic operation, we can instead put it on the left of the equals sign to achieve the same effect in a more readable way:

```
y rnd= c5 + x2 * (c7 + x2 * c9);
```

Since we are interested in bounding the round-off error, we need a few more definitions. In particular, we need to express the following infinitely precise computations:

```
Mx2 = x * x;
My = c5 + Mx2 * (c7 + Mx2 * c9);
```

Now, all that is left is to ask the tool to verify a bound on a round-off error. For example, the following formula means that, for any `x` less than 2^{-7} in absolute value, the relative error (denoted by operator `-/`) between `y` and `My` is bounded by 2^{-53}:

```
{ |x| <= 1b-7 -> |y -/ My| <= 1b-53 }
```

The tool validates that this formula actually holds. It can also produce a formal proof that can be mechanically checked by the Coq proof assistant. For analysis or debugging purposes, one can also leave holes in the formulas. For example, if one wonders what the absolute error is when `x` is much smaller, the following formula can be sent to Gappa:

```
{ |x| <= 1b-50 -> y - My in ? }
#@-Eprecision=200
```

The tool answers as follows:

```
Results:
  y - My in [0, 7320136537069203b-165 {1.5652e-34, 2^(-112.299)}]
```

In other words, the round-off error is always non-negative but no larger than $2^{-112} \simeq 1.6 \cdot 10^{-34}$. Note that the option `-Eprecision=200` has been passed to the tool in order to increase the internal precision of Gappa's interval arithmetic. Indeed, with its default precision, Gappa would only be able to prove an upper bound of 2^{-66}, which is correct but very pessimistic.

5. Extending the precision

As seen in Section 4.2, error-free transformations (EFTs) make it possible to compute the rounding error of some operations. Upon them, one can then build double-word operators to retain more accuracy using only processor floating-point operations as in Section 5.1. Pair arithmetic is a variation on this approach that avoids some *renormalization* steps, as shown in Section 5.2.

A double-word or pair arithmetic provides a way of extending the precision by systematically replacing floating-point operations with costlier basic blocks.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

A more parsimonious use of EFTs leads to the so-called *compensated algorithms*, as shown in Section 5.3.

These approaches are meant to emulate a floating-point precision that is double the working precision, but this might not be sufficient. Section 5.4 presents some tools and libraries that provide higher precision.

5.1. Double-word arithmetic

Since Algorithms 1 (Fast2Sum), 2 (2Sum) and 3 (2Prod) produce exact results r in the form of unevaluated sums of two floating-point numbers, whose first one is equal to $\mathrm{RN}(r)$, it is natural to try to perform calculations on such unevaluated sums. If the underlying floating-point format being used is binary64, this makes it easy to manipulate numbers with a precision larger than 100 bits. This can be very useful in critical parts of programs, since the binary128 format (often called *quad precision*), which has been specified in IEEE-754 since the 2008 version, is seldom implemented in hardware. To our knowledge, in recent years the only widely distributed computer with a hardwired binary128 arithmetic is the IBM z system.

In the 1960s, Gregory and Raney (1964) and Ikebe (1965) suggested representing large precision numbers as the sum of an FP number and scaled integers. In his seminal paper, Dekker (1971) introduced the first double-word algorithms (called *doublelength* in his paper) for addition, multiplication, division and square root. The first interest of double-word arithmetic is to obtain more accurate results. However, as mentioned by Riedy and Demmel (2018), it can also help to improve reproducibility of parallel codes (He and Ding 2000) or to accelerate some linear algebra algorithms (Yamazaki, Tomov and Dongarra 2015). Libraries that provide double-word arithmetic have been written by Hida, Li and Bailey (2001)[35] and Briggs.[36] A more recent library is CAMPARY[37] (Joldeş, Muller, Popescu and Tucker 2016).

The definition of a double-word number may vary slightly. We use the same definition as Joldeş, Muller and Popescu (2017).

Definition 5.1. A *double-word* (DW) number x is the unevaluated sum $x_h + x_\ell$ of two floating-point numbers x_h and x_ℓ such that $x_h = \mathrm{RN}(x)$.

The following algorithms are presented by Joldeş *et al.* (2017) for the sum or product of a double-word number and an FP number, the sum, product or quotients of two DW numbers, and by Lefèvre *et al.* (2022) for the square root. These algorithms are variants of Dekker's *doublelength* algorithms. The error bounds have been proved by Joldeş *et al.* (2017), Muller and Rideau (2022) and Lefèvre *et al.* (2022). There is a formal proof in Coq for all these error bounds.

[35] Their QD Library is available at https://www.davidhbailey.com/dhbsoftware/.
[36] No longer supported: see http://keithbriggs.info/doubledouble.html.
[37] https://homepages.laas.fr/mmjoldes/campary/

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 11 DWPlusFP(x_h, x_ℓ, y)

Require: (x_h, x_ℓ) is a DW number

$(s_h, s_\ell) \leftarrow$ 2Sum(x_h, y)
$v \leftarrow$ RN($x_\ell + s_\ell$)
$(z_h, z_\ell) \leftarrow$ Fast2Sum(s_h, v)
return (z_h, z_ℓ)

Algorithm 12 SloppyDWPlusDW(x_h, x_ℓ, y_h, y_ℓ)

$(s_h, s_\ell) \leftarrow$ 2Sum(x_h, y_h)
$v \leftarrow$ RN($x_\ell + y_\ell$)
$w \leftarrow$ RN($s_\ell + v$)
$(z_h, z_\ell) \leftarrow$ Fast2Sum(s_h, w)
return (z_h, z_ℓ)

Algorithm 13 AccurateDWPlusDW(x_h, x_ℓ, y_h, y_ℓ)

$(s_h, s_\ell) \leftarrow$ 2Sum(x_h, y_h)
$(t_h, t_\ell) \leftarrow$ 2Sum(x_ℓ, y_ℓ)
$c \leftarrow$ RN($s_\ell + t_h$)
$(v_h, v_\ell) \leftarrow$ Fast2Sum(s_h, c)
$w \leftarrow$ RN($t_\ell + v_\ell$)
$(z_h, z_\ell) \leftarrow$ Fast2Sum(v_h, w)
return (z_h, z_ℓ)

Algorithm 14 DWTimesFP1(x_h, x_ℓ, y)

$(c_h, c_{\ell 1}) \leftarrow$ 2Prod(x_h, y)
$c_{\ell 2} \leftarrow$ RN($x_\ell y$)
$(t_h, t_{\ell 1}) \leftarrow$ Fast2Sum($c_h, c_{\ell 2}$)
$t_{\ell 2} \leftarrow$ RN($t_{\ell 1} + c_{\ell 1}$)
$(z_h, z_\ell) \leftarrow$ Fast2Sum($t_h, t_{\ell 2}$)
return (z_h, z_ℓ)

Algorithm 11 (Joldeş *et al.* 2017, Algorithm 4) is implemented in the QD library. The returned double-word number $z = (z_h, z_\ell)$ is such that $z = (x_h + x_\ell + y) \cdot (1 + \varepsilon)$, with $|\varepsilon| \leq 2\mathbf{u}^2$.

Algorithm 12 computes the sum $(x_h, x_\ell) + (y_h, y_\ell)$. Its relative error is not bounded in the general case (hence the name 'sloppy'). But if x_h and y_h have the same sign, the relative error is less than $3\mathbf{u}^2$. A more accurate version is provided by Algorithm 13, which has a relative error less than $3\mathbf{u}^2$ in all cases.

Algorithm 14, suggested by Li *et al.* (2000), computes the product $(x_h, x_\ell) \times y$. Its relative error is less than $\frac{3}{2}\mathbf{u}^2 + 4\mathbf{u}^3$. Note that since the algorithm calls 2Prod(x_h, y),

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 15 DWTimesDW2$(x_h, x_\ell, y_h, y_\ell)$

$(c_h, c_{\ell 1}) \leftarrow$ 2Prod(x_h, y_h)
$t_\ell \leftarrow \mathrm{RN}(x_h y_\ell)$
$c_{\ell 2} \leftarrow \mathrm{RN}(t_\ell + x_\ell y_h)$
$c_{\ell 3} \leftarrow \mathrm{RN}(c_{\ell 1} + c_{\ell 2})$
$(z_h, z_\ell) \leftarrow$ Fast2Sum$(c_h, c_{\ell 3})$
return (z_h, z_ℓ)

Algorithm 16 DWDivDW2$(x_h, x_\ell, y_h, y_\ell)$

$t_h \leftarrow \mathrm{RN}(x_h / y_h)$
$(r_h, r_l) \leftarrow$ DWTimesFP1(y_h, y_ℓ, t_h) {approximation to $(y_h + y_\ell) \cdot t_h$}
$\pi_h \leftarrow \mathrm{RN}(x_h - r_h) = x_h - r_h$ {exact operation}
$\delta_\ell \leftarrow \mathrm{RN}(x_\ell - r_\ell)$
$\delta \leftarrow \mathrm{RN}(\pi_h + \delta_\ell)$
$t_\ell \leftarrow \mathrm{RN}(\delta / y_h)$
$(z_h, z_\ell) \leftarrow$ Fast2Sum(t_h, t_ℓ)
return (z_h, z_ℓ)

Algorithm 17 SQRTDWtoDW(x_h, x_ℓ)

if $x_h = 0$ **then**
  return $(0, 0)$
else
  $s_h \leftarrow \mathrm{RN}(\sqrt{x_h})$
  $\rho_1 \leftarrow \mathrm{RN}(x_h - s_h^2)$ {with an FMA instruction}
  $\rho_2 \leftarrow \mathrm{RN}(x_\ell + \rho_1)$
  $s_\ell \leftarrow \mathrm{RN}(\rho_2 / (2 \cdot s_h))$
  $(z_h, z_\ell) \leftarrow$ Fast2Sum(s_h, s_ℓ)
  return (z_h, z_ℓ)
end if

variables x_h and y must be in the domain of validity of 2Prod, that is, they must satisfy the condition of Property 2.12.

Algorithm 15 (Joldeş *et al.* 2017, Algorithm 11) computes the product $(x_h, x_\ell) \times (y_h, y_\ell)$, assuming an FMA operator is available. Its relative error is less than $5\mathbf{u}^2$. Again, since the algorithm calls 2Prod(x_h, y_h), variables x_h and y_h must be in the domain of validity of 2Prod, that is, they must satisfy the condition of Property 2.12.

Algorithm 16 performs the division $(x_h, x_\ell) \div (y_h, y_\ell)$. Its relative error is less than $15\mathbf{u}^2 + 56\mathbf{u}^3$.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Table 5.1. Summary, partly extracted from the paper by Muller and Rideau (2022), of the results presented by Joldeş *et al.* (2017), Muller and Rideau (2022) and Lefèvre *et al.* (2022), all formally proved. Unless stated otherwise, the largest errors observed in experiments are for RN being round-to-nearest, ties-to-even.

Operation	Name of algorithm in Joldeş *et al.* (2017) or Lefèvre *et al.* (2022)	Bound formally proved	Largest relative error built or observed in experiments
DW + FP	DWPlusFP	$2\mathbf{u}^2$	$2\mathbf{u}^2 - 6\mathbf{u}^3$
DW + DW	SloppyDWPlusDW	N/A	1
	AccurateDWPlusDW	$3\mathbf{u}^2 + 13\mathbf{u}^3$	$3\mathbf{u}^2 - 11\mathbf{u}^3 + O(\mathbf{u}^4)$
DW × FP	DWTimesFP1	$\frac{3}{2}\mathbf{u}^2 + 4\mathbf{u}^3$	$1.5\mathbf{u}^2$
	DWTimesFP2	$3\mathbf{u}^2$	$2.517\mathbf{u}^2$
	DWTimesFP3	$2\mathbf{u}^2$	$1.984\mathbf{u}^2$
DW × DW	DWTimesDW1	$5\mathbf{u}^2$ (ties-to-even)	$4.9916\mathbf{u}^2$ (ties-to-even)
		$5.5\mathbf{u}^2$ (general)	$5.4907\mathbf{u}^2$ (ties-to-0)
	DWTimesDW2	$5\mathbf{u}^2$	$4.9433\mathbf{u}^2$
	DWTimesDW3	$4\mathbf{u}^2$	$3.936\mathbf{u}^2$
DW ÷ FP	DWDivFP1	$3.5\mathbf{u}^2$	$2.95\mathbf{u}^2$
	DWDivFP2	$3.5\mathbf{u}^2$	$2.95\mathbf{u}^2$
	DWDivFP3	$3\mathbf{u}^2$	$2.95\mathbf{u}^2$
DW ÷ DW	DWDivDW1	$15\mathbf{u}^2 + 56\mathbf{u}^3$	$8.465\mathbf{u}^2$
	DWDivDW2	$15\mathbf{u}^2 + 56\mathbf{u}^3$	$8.465\mathbf{u}^2$
	DWDivDW3	$9.8\mathbf{u}^2$	$5.922\mathbf{u}^2$
$\sqrt{\text{DW}}$	SQRTDWtoDW	$\frac{25}{8}\mathbf{u}^2$	$\frac{25}{8}\mathbf{u}^2 - \frac{343}{8}\mathbf{u}^3$

Algorithm 17 computes the square root of the DW number (x_h, x_ℓ). Its relative error is less than $\frac{25}{8}\mathbf{u}^2$.

Table 5.1 refers to some algorithms (e.g. DWTimesFP2) that are not given in this article. These algorithms are presented (with the same names) in Joldeş *et al.* (2017). They are slight variants of Algorithms 14, 15 and 16. These variants allow us to choose different compromises between speed and accuracy. For instance, Algorithm DWTimesDW3 is obtained by adding the term $x_\ell y_\ell$ (which is neglected in Algorithm 15): this results in a more accurate yet slower calculation.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

5.2. *Lange and Rump's pair arithmetic*

The double-word algorithms presented in Section 5.1 are very accurate but they suffer from a drawback: they all end up with an instruction of the form '$(z_h, z_\ell) \leftarrow$ Fast2Sum(a, b)'. That instruction costs three consecutive floating-point operations (which is a significant penalty, especially for the simplest algorithms), and one may wonder why we perform that instruction, knowing that all the information on the result is already in variables a and b, since $z_h + z_\ell = a + b$.

As a matter of fact, that last Fast2Sum is a *renormalization*. The pair (a, b) is not necessarily (in fact, it seldom is) a double-word number, since the FP number a may not equal $\mathrm{RN}(a + b)$. If we perform one of the algorithms of Section 5.1 with double-word inputs but *without* the last Fast2Sum, the output (a, b) will satisfy the error bounds given in Table 5.1, but that output will *not* be a valid entry for a subsequent double-word algorithm. In other words, if the algorithms are fed with such outputs, the resulting error may be larger than the bounds of Table 5.1.

However, a natural question is: If we decide in any case to perform such algorithms *without* a final renormalization, what do we lose? We will almost certainly end up with larger final errors, but maybe we will be able to say something on the final result of such a sequence of *pair operations*? Lange and Rump (2020) undertook this task and gave conditions for the final result of a calculation to be a faithful rounding of the exact result. More precisely, they define the following *pair operations*.

- CPairSum$(x_h, x_\ell, y_h, y_\ell)$ is SloppyDWPlusDW$(x_h, x_\ell, y_h, y_\ell)$ (i.e. Algorithm 12), where the pair (s_h, w) is returned (i.e. we drop the last Fast2Sum).
- CPairProd$(x_h, x_\ell, y_h, y_\ell)$, adapted with our notation, is given by Algorithm 18 (quite similar to Algorithm 15 without the last Fast2Sum).
- CPairDiv$(x_h, x_\ell, y_h, y_\ell)$, adapted with our notation, is given by Algorithm 19.
- CPairSQRT(x_h, x_ℓ) is SQRTDWtoDW$(x_h, x_\ell, y_h, y_\ell)$ (i.e. Algorithm 17) where the pair (s_h, s_ℓ) is returned (i.e. we drop the last Fast2Sum).

Lange and Rump (2020, Theorems 4.2 and 5.4) give general sufficient conditions, for a computation represented by an evaluation tree whose nodes are pair operations, to return a faithfully rounded result. We will not present these conditions here, as this would require too much introductory material, but we give some of the corollaries below. These corollaries are with the implicit assumption that there are no underflows or overflows.

Corollary 5.2 (product of n FP numbers (Lange and Rump 2020)). We assume $(x_1, \ldots, x_n) \in \mathbb{F}^n$. If $n \leq 1/\sqrt{2\mathbf{u}} - 1$, then the product (c, g) of all x_i computed with the pair arithmetic in any order is such that $\mathrm{RN}(c + g)$ is a faithful rounding of the exact product.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 18 CPairProd$(x_h, x_\ell, y_h, y_\ell)$

$(c_h, c_{\ell 1}) \leftarrow 2\text{Prod}(x_h, y_h)$
$t_{\ell 1} \leftarrow \text{RN}(x_h \cdot y_\ell)$
$t_{\ell 2} \leftarrow \text{RN}(x_\ell \cdot y_h)$
$c_{\ell 2} \leftarrow \text{RN}(t_{\ell 1} + t_{\ell 2})$
$c_{\ell 3} \leftarrow \text{RN}(c_{\ell 1} + c_{\ell 2})$
return $(c_h, c_{\ell 3})$

Algorithm 19 CPairDiv$(x_h, x_\ell, y_h, y_\ell)$

$t_h \leftarrow \text{RN}(x_h / y_h)$
$\rho \leftarrow \text{RN}(x_h - y_h t_h) = x_h - y_h t_h$ {exact operation}
$\pi \leftarrow \text{RN}(\rho + x_\ell)$
$q \leftarrow \text{RN}(t_h \cdot y_\ell)$
$r \leftarrow \text{RN}(\pi - q)$
$s \leftarrow \text{RN}(y_h + y_\ell)$
$g \leftarrow \text{RN}(r/s)$
return (t_h, q)

Corollary 5.3 (sum of n FP numbers (Lange and Rump 2020)). We assume $(x_1, \ldots, x_n) \in \mathbb{F}^n$. Define

$$k = \left(\sum_{i=1}^{n} |x_i|\right) \Big/ \left|\sum_{i=1}^{n} x_i\right|.$$

If $n \leq 1/\sqrt{2k\mathbf{u}} - 1$, then the sum (c, g) of all x_i computed with the pair arithmetic in any order is such that $\text{RN}(c + g)$ is a faithful rounding of the exact sum.

Similar corollaries are given by Lange and Rump (2020) for the computation of dot products and the evaluation of polynomials using Horner's scheme.

5.3. *Compensated algorithms*

Error-free transformations give access to the rounding errors of individual operations. We can use them later on in the calculation to (at least partly) compensate for these errors. This is the principle behind *compensated algorithms*. Let us illustrate these algorithms in the case of the computation of the sum of n floating-point numbers. We do not claim exhaustiveness here, as there is a large body of literature on summation algorithms; see for instance the works by Rump *et al.* (2008), Demmel and Nguyen (2015), Higham (1993, 2002), Blanchard *et al.* (2020) and Lange (2022). We give a few examples below.

The *compensated summation algorithm* given in Algorithm 20 was found independently by Kahan (1965) and Babuška (1969).

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 20 Kahan–Babuška($x_1, \ldots, x_n$)

1: $s \leftarrow x_1$
2: $c \leftarrow 0$
3: **for** $i = 2$ to n **do**
4: $\quad y \leftarrow \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(x_i + c)$
5: $\quad t \leftarrow \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(s + y)$
6: $\quad c \leftarrow \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(y - \mathrm{R}_{\mathrm{D}}^{\mathrm{U}}(t - s))$
7: $\quad s \leftarrow t$
8: **end for**
9: **return** s

Lines 5 and 6 of Algorithm 20 are simply Fast2Sum(s, y) (Algorithm 1). Note, however, that the conditions that guarantee $t + c = s + y$ may not be satisfied. Indeed, the rounding function is not necessarily RN, and $|y|$ is not necessarily less than or equal to $|s|$, so the correcting term c used to update the next x_i may be only an approximation to the error of t. And yet the algorithm already enjoys an error bound better than (4.8): if each floating-point operation has relative error at most $\mathbf{v}$ then, as shown by Hallman and Ipsen (2022), the final value of s satisfies

$$\left| s - \sum_{i=1}^{n} x_i \right| \leq \left(3\mathbf{v} + O(n\mathbf{v}^2)\right) \cdot \sum_{i=1}^{n} |x_i|;$$

here, $\mathbf{v} = \mathbf{u}$ when all computations are done using round-to-nearest, while $\mathbf{v} = 2\mathbf{u}$ when resorting to some generic rounding function $\mathrm{R}_{\mathrm{D}}^{\mathrm{U}}$.

Later on, Pichat (1972) and Neumaier (1974) independently suggested a more accurate variant of Algorithm 20 where, instead of immediately adding at step i the previously computed value of the error term c (let us call it c_{i-1}), it accumulates these values c_i and adds their computed sum to s at the end of the calculation. They also performed tests to use Algorithm Fast2Sum in its domain of validity. Priest (1992) also gave a *doubly compensated summation algorithm*, which guarantees high relative accuracy provided the summands x_i have first been sorted into decreasing order of magnitude.

To avoid tests, Ogita *et al.* (2005) replaced the use of Fast2Sum and a test in the Pichat–Neumaier algorithm by a 2Sum (Algorithm 2), and considered a generalization (Algorithm 21), where the sum of the error terms is computed again with the same algorithm (which was already suggested by Pichat). Then, calling Algorithm 21 repeatedly, this gives the K-fold algorithm (Algorithm 22).

Rump *et al.* (2008) showed that the final value of variable c satisfies

$$\left| c - \sum_{i=1}^{n} x_i \right| \leq \left(\mathbf{u} + \gamma_{n-1}^2\right) \cdot \left| \sum_{i=1}^{n} x_i \right| + \gamma_{2n-2}^{K} \cdot \sum_{i=1}^{n} |x_i|,$$

where $\gamma_n := n\mathbf{u}/(1 - n\mathbf{u})$.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Algorithm 21 VecSum$(x_1, \ldots, x_n)$

```
p ← x
for i = 2 to n do
    (p_i, p_{i−1}) ← 2Sum(p_i, p_{i−1})
end for
return p
```

Algorithm 22 K-fold$(x_1, \ldots, x_n)$

```
for k = 1 to K − 1 do
    x ← VecSum(x)
end for
c ← x_1
for i = 2 to n do
    c ← c + x_i
end for
return c
```

Another, very different, way of using EFTs to perform accurate summation is the following. Malcolm (1971), starting from ideas expressed by Wolfe (1964), splits the range of floating-point exponents into *bins* of a fixed width $W < p$ (which depends on the number of terms to be added). Then, using a splitting algorithm similar to Algorithm 6, each input number x_i is decomposed as the sum of a small number of lower-precision FP numbers, called *slices* by Demmel *et al.* (2016), such that the weights of the significand bits of a slice all belong to the same bin. If W is well chosen, then all the elements of a bin can be accumulated without error. There remains to add the non-zero terms accumulated in the bins. Malcolm suggests doing that most significant bin first. The *reproducible summation algorithm* of Demmel *et al.* (2016) builds on this method. Another algorithm based on judicious splittings of the operands is the Accsum algorithm by Rump *et al.* (2008), which guarantees faithful rounding of a sum.

5.4. *Extended precision software*

As seen in Section 5.1, we may rely on the available processor FP operations to build extended precision arithmetic, for instance using double-word algorithms, that roughly double the available precision. Triple-word (Fabiano, Muller and Picot 2019) and quad-word algorithms (Hida *et al.* 2001) are available as well.

When additional precision is needed, even more than four FP numbers may be used, and a number may be represented by the unevaluated sum of an arbitrary number of floating-point numbers, with some conditions to avoid having too much *overlapping* between them. Such sums are called *floating-point expansions* (Priest 1991, Shewchuk 1997, Daumas 1999, Popescu 2017).

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Software libraries exist based on these ideas. Double-word and quad-word arithmetics are available in QD.[38] CAMPARY[39] provides the double-word algorithms formally proved by Muller and Rideau (2022) as well as arbitrary-precision computations based on floating-point expansions.

When the desired precision becomes very large, methods based on floating-point expansions are no longer efficient, and one needs to switch to another solution that consists in representing a number by an arbitrary-precision significand with a large enough exponent. The pioneering work was done by Brent (1978) with his MP multiple-precision package. GMP[40] is known for its very efficient arbitrary-precision integers, but it also provides rational numbers and floating-point numbers. GNU MPFR[41] is a C library based on GMP that provides multiple-precision floating-point arithmetic with correct rounding (Fousse *et al.* 2007). Pari/GP[42] provides fast multiple-precision arithmetic, either based on GMP or on its own native kernel (usually slower); it is mainly targeted at number theorists. MPFUN2020[38] is a Fortran package for multiple-precision arithmetic with special care taken to be thread-safe; it includes two versions, one based on MPFR.

Another way to ensure the correctness is to provide some enclosing of the exact result. This may be done by interval arithmetic as explained in Section 4.1. MPFI[43] is a C library for arbitrary-precision interval arithmetic, where intervals are represented by their endpoints; it is based on MPFR for the implementation of the endpoints (Revol and Rouillier 2005). Arb[44] is a C library for arbitrary-precision ball arithmetic (a form of interval arithmetic that uses a midpoint-radius representation of real and complex numbers); it includes an impressive library of special functions (Johansson 2013).

Extended precision may also be hidden to the user. For instance, computer algebra software systems as Maple,[45] Sage,[46] or Mathemagix[47] provide extended-precision floating-point numbers (based on GMP for Sage, and on MPFR for Mathemagix).

[38] https://www.davidhbailey.com/dhbsoftware/
[39] https://homepages.laas.fr/mmjoldes/campary/
[40] https://gmplib.org/
[41] https://mpfr.loria.fr/
[42] http://pari.math.u-bordeaux.fr/
[43] https://gitlab.inria.fr/mpfi/mpfi/
[44] https://arblib.org/
[45] https://www.maplesoft.com/products/Maple/
[46] https://www.sagemath.org/
[47] http://www.mathemagix.org/

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

6. Conclusion

This conclusion focuses on the evolution of floating-point arithmetic and its links to other domains. Section 6.1 is about useful operators that could spread and be standardized in the future. Section 6.2 describes some recent alternative arithmetics. Section 6.3 presents some fruitful links of floating-point arithmetic to formal proof and computer algebra.

6.1. Implemented but not (yet?) standard operators and rounding functions

We have presented many arithmetic operators and rounding functions, the basic ones in Section 2 (including some more exotic ones in Section 2.12) and a few others added by the last revision of the Standard in Section 4.2.3. Nevertheless, some other features might be useful, whether they are rounding functions or operators.

An old rounding function is *round-to-odd*, which was first considered by von Neumann when designing the arithmetic unit of the EDVAC, and later used by Goldberg for converting FP numbers to decimal representation. It was used by GCC for the reverse conversion, and formally studied by Boldo and Melquiond (2008). Its main property related to double rounding is explained in Section 2.10.2.

As far as operators are concerned, we may desire that the augmented operations presented in Section 4.2.3 should also be available in round-to-nearest, ties-to-even. All EFTs (see Section 4.2) would greatly benefit from an efficient hardware implementation of such operations.

Another possibility is to have more complex FP operators. As the FMA instruction can replace and generalize floating-point addition and multiplication, the FDMDA[48] (fused dual multiply dual add) instruction, which computes

$$ab + cd + e,$$

could replace and generalize the FMA. The question of the specification of such an operator is then crucial, in particular whether it is correctly rounded (meaning as if there were a single rounding at the end of the four operations). This may be available in the next generations of processors, due to its usefulness. More than solving the difficult problem of the correctly rounded sum of three numbers (Kornerup *et al.* 2012) and making most EFT computations trivial, it would greatly simplify and make more accurate double-word arithmetic algorithms, some complex arithmetic operations and probably many other algorithms.

6.2. Alternative arithmetics

It is natural and sound to consider that floating-point arithmetic, as defined in the 1970s and 1980s, may now be questioned. As we saw in the Introduction, the processor technology and target applications have drastically changed since the birth of IEEE-754. For some applications, arithmetics other than conventional

[48] See https://patents.google.com/patent/US20220222073A1/en.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

floating-point may be better suited, for instance when the range of numbers we need to represent is known beforehand or very large. The emergence of digital neural networks for AI applications requires arithmetics with tiny precisions (to achieve high bandwidth) yet rather large range (Wang and Kanwar 2019, Bertaccini *et al.* 2022).

In order to obtain a larger range than conventional floating-point while maintaining good precision for numbers of the usual order of magnitude, Gustafson (2015) introduced the *unums*, and later on their successors, the *posits* (Posit Working Group 2022). Posits are still numbers of the form $M \cdot 2^E$, where M and E are integers, but with a cleverly designed encoding that allows for variable sizes of M and E (but with a fixed total word size). Experiments by Cococcioni, Rossi, Ruffaldi and Saponara (2022) suggest that *Posit8* could be an interesting solution for implementing digital neural networks. For general higher-precision numerical computing, however, the price to pay is huge. Indeed, with posits, the standard model (see Section 2.1) no longer applies, that is, the individual arithmetic operations no longer have a bounded relative error (de Dinechin, Forget, Muller and Uguen 2019).

6.3. *Floating-point arithmetic and beyond*

As soon as a numerical algorithm contains more than a few operations, it is a tremendous task to obtain very tight error bounds and to make sure that underflows and overflows will not occur and that NaNs will not propagate throughout the calculation. The proofs we publish are seldom fully satisfactory. Either we just give a rough idea of what makes the algorithm work, coyly hiding all the dirty stuff, or we try to take into account all possible corner cases, ending, as mentioned in the Introduction, with long and tedious proofs, with the consequence that a flaw may remain unnoticed for years. For example, Kahan (2004*b*) proposed an intricate algorithm for computing a correct discriminant.[49] The formal verification of Kahan's algorithm showed a gap in the pen-and-paper proof, as a test could be wrong with respect to its mathematical counterpart, creating unstudied cases. Fortunately, the algorithm was still correct in these special cases and formally proved by Boldo (2009).

And yet, floating-point arithmetic is needed in many critical applications, and for a given application we would like to analyse not just one solution but many of them, in order to choose the best suited. For small formats up to 32 bits, exhaustive testing is achievable for basic operators. But when the precision increases it becomes intractable, and *formal proof* is then a precious tool that can provide a strong validation of the pen-and-paper proof of an algorithm, as seen in several parts of this article.

[49] A naive algorithm may not be accurate enough, as a cancellation may endanger the sign of the result.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Another solution is to try to build algorithms and proofs that are correct by construction. From that point of view, *computer algebra* could considerably help the design and analysis of floating-point programs, in several respects. First, the design of function software could be made easier and the probability of bugs in such software could be made much lower, using tools that automatically find symmetries and build local as well as asymptotic approximations, for instance from the differential equation that defines the function. Second, part of the analysis that uses the standard model could be assisted by computer. This would allow one to explore many variants of a numerical algorithm; this would also allow the analysis of numerical programs significantly larger than the ones we are able to deal with by pen-and-paper calculations. The pioneering work of Mezzarobba (2010, 2020) addresses these two topics.

Acknowledgements

We thank Michael Overton for his comments on an early draft of this article. We are also grateful for the thorough work of the copy-editor.

This work was partly supported by the European Research Council (ERC) under the European Union's Horizon 2020 Research and Innovation Programme – Grant Agreements 810367 and 101001995. It was also partly supported by the NuSCAP project (ANR-20-CE48-0014) of the French national research agency (ANR).

References

A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi and K. Gopalakrishnan (2019), DLFloat: A 16-b floating point format designed for deep learning training and inference, in *26th IEEE Symposium on Computer Arithmetic*, IEEE, pp. 92–95.

C. S. Anderson, J. Zhang and M. Cornea (2018), Enhanced vector math support on the Intel®AVX-512 architecture, in *25th IEEE Symposium on Computer Arithmetic*, pp. 120–124.

I. Babuška (1969), Numerical stability in mathematical analysis, in *Proceedings of the 1968 IFIP Congress*, Vol. 1, pp. 11–23.

R. C. M. Barnes, E. H. Cooke-Yarborough and D. G. A. Thomas (1951), An electronic digital computor using cold cathode counting tubes for storage (Part 1), *Electron. Engng* **23**, 286–291.

T. Bartels, V. Fisikopoulos and M. Weiser (2022), Fast floating-point filters for robust predicates. Available at arXiv:2208.00497.

M. Baudin and R. L. Smith (2012), A robust complex division in Scilab. Available at arXiv:1210.4539.

N. H. F. Beebe (2017), *The Mathematical-Function Computation Handbook*, Springer.

L. Bertaccini, G. Paulin, T. Fischer, S. Mach and L. Benini (2022), MiniFloat-NN and ExSdotp: An ISA extension and a modular open hardware unit for low-precision training on RISC-V cores, in *29th IEEE Symposium on Computer Arithmetic*.

P. Blanchard, N. J. Higham and T. Mary (2020), A class of fast and accurate summation algorithms, *SIAM J. Sci. Comput.* **42**, A1541–A1557.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

G. Bohlender, W. Walter, P. Kornerup and D. Matula (1991), Semantics for exact floating point operations, in *10th IEEE Symposium on Computer Arithmetic*, pp. 22–26.

S. Boldo (2006), Pitfalls of a full floating-point proof: Example on the formal proof of the Veltkamp/Dekker algorithms, in *3rd International Joint Conference on Automated Reasoning* (U. Furbach and N. Shankar, eds), Vol. 4130 of Lecture Notes in Computer Science, Springer, pp. 52–66.

S. Boldo (2009), Kahan's algorithm for a correct discriminant computation at last formally proven, *IEEE Trans. Comput.* **58**, 220–225.

S. Boldo and M. Daumas (2003), Representable correcting terms for possibly underflowing floating point operations, in *16th IEEE Symposium on Computer Arithmetic* (J.-C. Bajard and M. Schulte, eds), pp. 79–86.

S. Boldo and G. Melquiond (2008), Emulation of a FMA and correctly rounded sums: Proved algorithms using rounding to odd, *IEEE Trans. Comput.* **57**, 462–471.

S. Boldo and G. Melquiond (2017), *Computer Arithmetic and Formal Proofs*, ISTE Press / Elsevier.

S. Boldo and J.-M. Muller (2005), Some functions computable with a fused-mac, in *17th IEEE Symposium on Computer Arithmetic*, pp. 52–58.

S. Boldo and J.-M. Muller (2011), Exact and approximated error of the FMA, *IEEE Trans. Comput.* **60**, 157–164.

S. Boldo, S. Graillat and J.-M. Muller (2017), On the robustness of the 2Sum and Fast2Sum algorithms, *ACM Trans. Math. Softw.* **44**, 4:1–4:14.

S. Boldo, C. Lauter and J.-M. Muller (2021), Emulating round-to-nearest ties-to-zero 'augmented' floating-point operations using round-to-nearest ties-to-even arithmetic, *IEEE Trans. Comput.* **70**, 1046–1058.

C. F. Borges (2021), Algorithm 1014: An improved algorithm for Hypot(x, y), *ACM Trans. Math. Softw.* **47**, 1–12.

C. F. Borges, C.-P. Jeannerod and J.-M. Muller (2022), High-level algorithms for correctly-rounded reciprocal square roots, in *29th IEEE Symposium on Computer Arithmetic*, pp. 18–25.

R. P. Brent (1973), On the precision attainable with various floating-point number systems, *IEEE Trans. Comput.* **C-22**, 601–607.

R. P. Brent (1978), Algorithm 524: MP, a Fortran multiple-precision arithmetic package [A1], *ACM Trans. Math. Softw.* **4**, 71–81.

R. Brent, C. Percival and P. Zimmermann (2007), Error bounds on complex floating-point multiplication, *Math. Comp.* **76**, 1469–1481.

N. Brisebarre and S. Chevillard (2007), Efficient polynomial L-approximations, in *18th IEEE Symposium on Computer Arithmetic*, pp. 169–176.

N. Brisebarre and J.-M. Muller (2008), Correctly rounded multiplication by arbitrary precision constants, *IEEE Trans. Comput.* **57**, 165–174.

N. Brisebarre, G. Hanrot and O. Robert (2017), Exponential sums and correctly-rounded functions, *IEEE Trans. Comput.* **66**, 2044–2057.

N. Brisebarre, M. Joldeş, J.-M. Muller, A.-M. Nanes and J. Picot (2020), Error analysis of some operations involved in the Cooley–Tukey fast Fourier transform, *ACM Trans. Math. Softw.* **46**, 11:1–11:27.

N. Brunie, F. de Dinechin, O. Kupriianova and C. Lauter (2015), Code generators for mathematical functions, in *22nd IEEE Symposium on Computer Arithmetic*, pp. 66–73.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

T. R. Cameron and S. Graillat (2022), On a compensated Ehrlich–Aberth method for the accurate computation of all polynomial roots, *Electron. Trans. Numer. Anal.* **55**, 401–423.

A. M. Castaldo, R. C. Whaley and A. T. Chronopoulos (2009), Reducing floating point error in dot product using the superblock family of algorithms, *SIAM J. Sci. Comput.* **31**, 1156–1174.

P. E. Ceruzzi (1981), The early computers of Konrad Zuse, 1935 to 1945, *Ann. Hist. Comput.* **3**, 241–262.

W. P. Champagne (1964), On finding roots of polynomials by hook or by crook. MSc thesis, University of Texas, Austin, TX.

S. Chevillard, J. Harrison, M. Joldeş and C. Lauter (2011), Efficient and accurate computation of upper bounds of approximation errors, *Theoret. Comput. Sci.* **412**, 1523–1543.

S. Chevillard, M. Joldeş and C. Lauter (2010), Sollya: An environment for the development of numerical codes, in *International Conference on Mathematical Software* (K. Fukuda *et al.*, eds), Vol. 6327 of Lecture Notes in Computer Science, Springer, pp. 28–31.

E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao and D. Burger (2018), Serving DNNs in real time at datacenter scale with project brainwave, *IEEE Micro* **38**, 8–20.

J. Cocke and V. Markstein (1990), The evolution of RISC technology at IBM, *IBM J. Res. Dev.* **34**, 4–11.

M. Cococcioni, F. Rossi, E. Ruffaldi and S. Saponara (2022), Small reals representations for deep learning at the edge: A comparison, in *Next Generation Arithmetic* (J. Gustafson and V. Dimitrov, eds), Springer, pp. 117–133.

W. J. Cody and W. Waite (1980), *Software Manual for the Elementary Functions*, Prentice-Hall.

C. Collange, D. Defour, S. Graillat and R. Iakymchuk (2015), Numerical reproducibility for the parallel reduction on multi- and many-core architectures, *Parallel Comput.* **49**, 83–97.

M. P. Connolly and N. J. Higham (2022), Probabilistic rounding error analysis of Householder QR factorization. MIMS EPrint 2022.5, Manchester Institute for Mathematical Sciences, The University of Manchester, UK. Available at http://eprints.maths.manchester.ac.uk/2865/.

M. P. Connolly, N. J. Higham and T. Mary (2021), Stochastic rounding and its probabilistic backward error analysis, *SIAM J. Sci. Comput.* **43**, A566–A585.

M. P. Connolly, N. J. Higham and S. Pranesh (2022), Randomized low rank matrix approximation: Rounding error analysis and a mixed precision algorithm. MIMS EPrint 2022.10, Manchester Institute for Mathematical Sciences, The University of Manchester, UK. Available at http://eprints.maths.manchester.ac.uk/2863/.

M. A. Cornea-Hasegan, R. A. Golliver and P. Markstein (1999), Correctness proofs outline for Newton–Raphson based floating-point divide and square root algorithms, in *14th IEEE Symposium on Computer Arithmetic*, pp. 96–105.

M. Cornea, J. Harrison and P. T. P. Tang (2002), *Scientific Computing on Itanium®-based Systems*, Intel Press.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

M. Croci, M. Fasi, N. J. Higham, T. Mary and M. Mikaitis (2022), Stochastic rounding: implementation, error analysis and applications, *Royal Soc. Open Sci.* **9**, 1–25.

J. Darcy (2017), Restore always-strict floating-point semantics. Technical report JEP 306.

M. Daumas (1999), Multiplications of floating point expansions, in *14th IEEE Symposium on Computer Arithmetic*, pp. 250–257.

M. Daumas, L. Rideau and L. Théry (2001), A generic library of floating-point numbers and its application to exact computing, in *14th International Conference on Theorem Proving in Higher Order Logics* (R. J. Boulton and P. B. Jackson, eds), Vol. 2152 of Lecture Notes in Computer Science, Springer, pp. 169–184.

F. de Dinechin, L. Forget, J.-M. Muller and Y. Uguen (2019), Posits: The good, the bad and the ugly, in *Conference on Next-Generation Arithmetic*, ACM Press, pp. 1–10.

F. de Dinechin, C. Lauter and G. Melquiond (2011), Certifying the floating-point implementation of an elementary function using Gappa, *IEEE Trans. Comput.* **60**, 242–253.

T. J. Dekker (1971), A floating-point technique for extending the available precision, *Numer. Math.* **18**, 224–242.

J. Demmel (1984), Underflow and the reliability of numerical software, *SIAM J. Sci. Statist. Comput.* **5**, 887–919.

J. Demmel, P. Ahrens and H. D. Nguyen (2016), Efficient reproducible floating point summation and BLAS. Technical report UCB/EECS-2016-121, EECS Department, University of California, Berkeley.

J. Demmel and Y. Hida (2004), Fast and accurate floating point summation with application to computational geometry, *Numer. Algorithms* **37**, 101–112.

J. Demmel and H. D. Nguyen (2015), Parallel reproducible summation, *IEEE Trans. Comput.* **64**, 2060–2070.

J. Demmel and J. Riedy (2021), A new IEEE 754 standard for floating-point arithmetic in an ever-changing world, *SIAM News* **54**, 9.

J. Demmel, J. Dongarra, M. Gates, G. Henry, J. Langou, X. Li, P. Luszczek, W. Pereira, J. Riedy and C. Rubio-González (2022), Proposed consistent exception handling for the BLAS and LAPACK. Available at arXiv:2207.09281.

E.-M. El Arar, D. Sohier, P. de Oliveira Castro and E. Petit (2022), The positive effects of stochastic rounding in numerical algorithms, in *29th IEEE Symposium on Computer Arithmetic*, pp. 58–65.

N. Fabiano, J.-M. Muller and J. Picot (2019), Algorithms for triple-word arithmetic, *IEEE Trans. Comput.* **68**, 1573–1583.

M. Fasi and M. Mikaitis (2020), CPFloat: A C library for simulating low-precision arithmetic. MIMS EPrint 2020.22, Manchester Institute for Mathematical Sciences, The University of Manchester, UK. Available at http://eprints.maths.manchester.ac.uk/2873/.

M. Fasi, N. J. Higham, M. Mikaitis and S. Pranesh (2021), Numerical behavior of NVIDIA tensor cores, *PeerJ Comput. Sci.* **7**, e330.

F. Févotte and B. Lathuilière (2016), VERROU: Assessing floating-point accuracy without recompiling. Available at https://hal.archives-ouvertes.fr/hal-01383417.

S. A. Figueroa (1995), When is double rounding innocuous?, *ACM SIGNUM Newsletter* **30**, 21–26.

G. Flegg, C. Hay and B. Moss (1985), *Nicolas Chuquet, Renaissance Mathematician: A Study With Extensive Translation of Chuquet's Mathematical Manuscript Completed in 1484*, Springer.

G. E. Forsythe (1959), Reprint of a note on rounding-off errors, *SIAM Review* **1**, 66–67.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

S. Fortune and C. J. Van Wyk (1993), Efficient exact arithmetic for computational geometry, in *9th Annual Symposium on Computational Geometry*, ACM, pp. 163–172.

L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier and P. Zimmermann (2007), MPFR: A multiple-precision binary floating-point library with correct rounding, *ACM Trans. Math. Softw.* **33**, 13–es.

P. Friedland (1967), Algorithm 312: Absolute value and square root of a complex number, *Commun. Assoc. Comput. Mach.* **10**, 665.

S. Gill (1951), A process for the step-by-step integration of differential equations in an automatic digital computing machine, *Math. Proc. Cambridge Philos. Soc.* **47**, 96–108.

D. Goldberg (1991), What every computer scientist should know about floating-point arithmetic, *ACM Computing Surveys* **23**, 5–48. Edited reprint available at https://docs.oracle.com/cd/E19059-01/fortec6u2/806-7996/806-7996.pdf from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*, also available at http://www.validlab.com/goldberg/addendum.html.

I. B. Goldberg (1967), 27 bits are not enough for 8-digit accuracy, *Commun. Assoc. Comput. Mach.* **10**, 105–106.

F. Goualard (2014), How do you compute the midpoint of an interval?, *ACM Trans. Math. Softw.* **40**, 11:1–11:25.

F. Goualard (2022), Drawing random floating-point numbers from an interval, *ACM Trans. Model. Comput. Simul.* **32**, 16:1–16:24.

S. Graillat and V. Ménissier-Morain (2007), Error-free transformations in real and complex floating-point arithmetic, in *2007 International Symposium on Nonlinear Theory and its Applications*, pp. 341–344.

S. Graillat and V. Ménissier-Morain (2008), Compensated Horner scheme in complex floating point arithmetic, in *8th Conference on Real Numbers and Computer*, pp. 133–146.

S. Graillat and V. Ménissier-Morain (2012), Accurate summation, dot product and polynomial evaluation in complex floating-point arithmetic, *Inform. Comput.* **216**, 57–71.

S. Graillat, V. Lefèvre and J.-M. Muller (2020), Alternative split functions and Dekker's product, in *27th IEEE Symposium on Computer Arithmetic*, pp. 41–47.

R. T. Gregory and J. L. Raney (1964), Floating-point arithmetic with 84-bit numbers, *Commun. Assoc. Comput. Mach.* **7**, 10–13.

J. L. Gustafson (2015), *The End of Error: Unum Computing*, Chapman & Hall / CRC.

E. Hallman and I. C. F. Ipsen (2022), Precision-aware deterministic and probabilistic error bounds for floating point summation. Available at arXiv:2203.15928.

J. Harrison (1999), A machine-checked theory of floating point arithmetic, in *12th International Conference in Theorem Proving in Higher Order Logics* (Y. Bertot *et al.*, eds), Vol. 1690 of Lecture Notes in Computer Science, Springer, pp. 113–130.

J. R. Hauser (1996), Handling floating-point exceptions in numeric programs, *ACM Trans. Program. Lang. Syst.* **18**, 139–174.

Y. He and C. H. Q. Ding (2000), Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications, in *14th International Conference on Supercomputing*, ACM, pp. 225–234.

J. L. Hennessy and D. A. Patterson (2012), *Computer Architecture: A Quantitative Approach*, fifth edition, Morgan Kaufman.

G. Henry, P. T. P. Tang and A. Heinecke (2019), Leveraging the bfloat16 artificial intelligence datatype for higher-precision computations, in *26th IEEE Symposium on Computer Arithmetic*, pp. 69–76.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

Y. Hida, X. S. Li and D. H. Bailey (2001), Algorithms for quad-double precision floating-point arithmetic, in *15th IEEE Symposium on Computer Arithmetic*, pp. 155–162.

N. J. Higham (1993), The accuracy of floating point summation, *SIAM J. Sci. Comput.* **14**, 783–799.

N. J. Higham (2002), *Accuracy and Stability of Numerical Algorithms*, second edition, SIAM.

N. J. Higham (2021*a*), The mathematics of floating-point arithmetic, *LMS Newsletter* **493**, 35–41.

N. J. Higham (2021*b*), Numerical stability of algorithms at extreme scale and low precisions. MIMS EPrint 2021.14, Manchester Institute for Mathematical Sciences, The University of Manchester, UK. Available at http://eprints.maths.manchester.ac.uk/id/eprint/2833.

N. J. Higham and T. Mary (2019), A new approach to probabilistic rounding error analysis, *SIAM J. Sci. Comput.* **41**, A2815–A2835.

N. J. Higham and T. Mary (2020), Sharper probabilistic backward error analysis for basic linear algebra kernels with random data, *SIAM J. Sci. Comput.* **42**, A3427–A3446.

N. J. Higham and T. Mary (2022), Mixed precision algorithms in numerical linear algebra, *Acta Numer.* **31**, 347–414.

N. J. Higham and S. Pranesh (2019), Simulating low precision floating-point arithmetic, *SIAM J. Sci. Comput.* **41**, C585–C602.

A. Hirshfeld (2009), *Eureka Man: The Life and Legacy of Archimedes*, Walker & Company.

T. E. Hull, T. F. Fairgrieve and P. T. P. Tang (1994), Implementing complex elementary functions using exception handling, *ACM Trans. Math. Softw.* **20**, 215–244.

IEEE (2015), *IEEE Standard for Interval Arithmetic (IEEE Std 1788-2015)*, IEEE.

IEEE (2019), *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754-2019)*, IEEE.

G. Iffrah (1999), *The Universal History of Numbers: From Prehistory to the Invention of the Computer*, Wiley.

Y. Ikebe (1965), Note on triple-precision floating-point arithmetic with 132-bit numbers, *Commun. Assoc. Comput. Mach.* **8**, 175–177.

V. Innocente and P. Zimmermann (2022), Accuracy of mathematical functions in single, double, extended double and quadruple precision. Available at hal-03141101.

Intel (2018), BFLOAT16: Hardware numerics definition. White Paper, available at https://www.intel.com/content/dam/develop/external/us/en/documents/bf16-hardware-numerics-definition-white-paper.pdf.

International Organization for Standardization (2010), *Programming Languages – Fortran – Part 1: Base language*, International Standard ISO/IEC 1539-1:2010.

International Organization for Standardization, Geneva, Switzerland (2011), *Programming Languages – C*, International Standard ISO/IEC 9899:2011.

I. C. F. Ipsen and H. Zhou (2020), Probabilistic error analysis for inner products, *SIAM J. Matrix Anal. Appl.* **41**, 1726–1741.

ISO/IEC (2022), C programming language – N3054, working draft of the standard (September 2022). https://en.wikipedia.org/wiki/C2.

C.-P. Jeannerod (2016), A radix-independent error analysis of the Cornea–Harrison–Tang method, *ACM Trans. Math. Softw.* **42**, 19:1–19:20.

C.-P. Jeannerod (2020), The relative accuracy of `(x+y)*(x-y)`, *J. Comput. Appl. Math.* **369**, 112613.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

C.-P. Jeannerod and J.-M. Muller (2017), On the relative error of computing complex square roots in floating-point arithmetic, in *51st Asilomar Conference on Signals, Systems, and Computers*, IEEE, pp. 737–740.

C.-P. Jeannerod and S. M. Rump (2018), On relative errors of floating-point operations: optimal bounds and applications, *Math. Comp.* **87**, 803–819.

C.-P. Jeannerod, P. Kornerup, N. Louvet and J.-M. Muller (2017*a*), Error bounds on complex floating-point multiplication with an FMA, *Math. Comp.* **86**, 881–898.

C.-P. Jeannerod, N. Louvet and J.-M. Muller (2013*a*), Further analysis of Kahan's algorithm for the accurate computation of 2×2 determinants, *Math. Comp.* **82**, 2245–2264.

C.-P. Jeannerod, N. Louvet and J.-M. Muller (2013*b*), On the componentwise accuracy of complex floating-point division with an FMA, in *21st IEEE Symposium on Computer Arithmetic* (A. Nannarelli *et al.*, eds), pp. 83–90.

C.-P. Jeannerod, N. Louvet, J.-M. Muller and A. Plet (2016), Sharp error bounds for complex floating-point inversion, *Numer. Algorithms* **73**, 735–760.

C.-P. Jeannerod, C. Monat and L. Thévenoux (2017*b*), More accurate complex multiplication for embedded processors, in *12th IEEE International Symposium on Industrial Embedded Systems*, pp. 1–4.

C.-P. Jeannerod, J.-M. Muller and P. Zimmermann (2018), On various ways to split a floating-point number, in *25th IEEE Symposium on Computer Arithmetic*, IEEE, pp. 53–60.

H. Jiang, S. Graillat, R. Barrio and C. Yang (2016), Accurate, validated and fast evaluation of elementary symmetric functions and its application, *Appl. Math. Comput.* **273**, 1160–1178.

F. Johansson (2013), Arb: A C library for ball arithmetic, *ACM Commun. Comput. Algebra* **47**, 166–169.

M. Joldeş, J.-M. Muller and V. Popescu (2017), Tight and rigorous error bounds for basic building blocks of double-word arithmetic, *ACM Trans. Math. Softw.* **44**, 1–27.

M. Joldeş, J.-M. Muller, V. Popescu and W. Tucker (2016), CAMPARY: Cuda multiple precision arithmetic library and applications, in *5th International Congress on Mathematical Software* (G. M. Greuel *et al.*, eds), Vol. 9725 of Lecture Notes in Computer Science, Springer, pp. 232–240.

W. Kahan (1965), Pracniques: Further remarks on reducing truncation errors, *Commun. Assoc. Comput. Mach.* **8**, 40.

W. Kahan (1981), Why do we need a floating-point arithmetic standard? Technical report, Computer Science, UC Berkeley. Available at http://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf.

W. Kahan (1987), Branch cuts for complex elementary functions or much ado about nothing's sign bit, in *The State of the Art in Numerical Analysis* (A. Iserles and M. J. D. Powell, eds), Oxford University Press, pp. 165–211.

W. Kahan (1997), Lecture notes on the status of IEEE standard 754 for binary floating-point arithmetic. Available at http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF.

W. Kahan (1998), Matlab's loss is nobody's gain. Available at https://people.eecs.berkeley.edu/~wkahan/MxMulEps.pdf.

W. Kahan (2004*a*), A logarithm too clever by half. Available at http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

W. Kahan (2004*b*), On the cost of floating-point computation without extra-precise arithmetic. Available at http://www.cs.berkeley.edu/~wkahan/Qdrtcs.pdf.

W. Kahan and J. W. Thomas (1991), Augmenting a programming language with complex arithmetic. Technical report UCB/CSD-92-667, EECS Department, University of California, Berkeley.

R. Karpinsky (1985), PARANOIA: A floating-point benchmark, *BYTE* **10**, 223.

D. E. Knuth (1998), *The Art of Computer Programming*, Vol. 2, third edition, Addison-Wesley.

P. Kornerup, V. Lefèvre, N. Louvet and J.-M. Muller (2012), On the computation of correctly rounded sums, *IEEE Trans. Comput.* **61**, 289–298. A proof of Theorems 2 and 3 can be found at https://hal.inria.fr/inria-00475279.

T. Kouya (2019), Performance evaluation of an efficient double-double BLAS1 function with error-free transformation and its application to explicit extrapolation methods, in *26th IEEE Symposium on Computer Arithmetic*, pp. 120–123.

H. Kuki and W. J. Cody (1973), A statistical study of the accuracy of floating point number systems, *Commun. Assoc. Comput. Mach.* **16**, 223–230.

U. Kulisch (1971), An axiomatic approach to rounded computations, *Numer. Math.* **18**, 1–17.

U. Kulisch (2013), *Computer Arithmetic and Validity: Theory, Implementation, and Applications*, Vol. 33 of Studies in Mathematics, De Gruyter.

M. La Porte and J. Vignes (1974), Error analysis in computing, in *Information Processing 74*, North-Holland.

M. Lange (2022), Toward accurate and fast summation, *ACM Trans. Math. Softw.* **48**, 1–39.

M. Lange and S. Oishi (2020), A note on Dekker's FastTwoSum algorithm, *Numer. Math.* **145**, 383–403.

M. Lange and S. M. Rump (2017), Error estimates for the summation of real numbers with application to floating-point summation, *BIT Numer. Math.* **57**, 927–941.

M. Lange and S. M. Rump (2019), Sharp estimates for perturbation errors in summations, *Math. Comp.* **88**, 349–368.

M. Lange and S. M. Rump (2020), Faithfully rounded floating-point computations, *ACM Trans. Math. Softw.* **46**, 1–20.

P. Langlois and N. Louvet (2007), How to ensure a faithful polynomial evaluation with the compensated Horner algorithm, in *18th IEEE Symposium on Computer Arithmetic*, pp. 141–149.

O. Lawlor, H. Govind, I. Dooley, M. Breitenfeld and L. Kale (2005), Performance degradation in the presence of subnormal floating-point values, in *International Workshop on Operating System Interference in High Performance Application*.

V. Lefèvre (2013), SIPE: Small Integer Plus Exponent, in *21th IEEE Symposium on Computer Arithmetic*, pp. 99–106.

V. Lefèvre and J.-M. Muller (2001), Worst cases for correct rounding of the elementary functions in double precision, in *15th IEEE Symposium on Computer Arithmetic*, pp. 111–118.

V. Lefèvre, N. Louvet, J.-M. Muller, J. Picot and L. Rideau (2022), Accurate calculation of Euclidean norms using double-word arithmetic, *ACM Trans. Math. Softw.* https://doi.org/10.1145/3568672.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

X. Li, J. Demmel, D. H. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, A. Kapur, M. Martin, T. Tung and D. J. Yoo (2000), Design, implementation and testing of extended and mixed precision BLAS. Technical report 45991, Lawrence Berkeley National Laboratory. Available at https://netlib.org/lapack/lawnspdf/lawn149.pdf.

C. Lichtenau, A. Buyuktosunoglu, R. Bertran, P. Figuli, C. Jacobi, N. Papandreou, H. Pozidis, A. Saporito, A. Sica and E. Tzortzatos (2022), AI accelerator on IBM Telum processor: Industrial product, in *49th ACM International Symposium on Computer Architecture*, ACM, pp. 1012–1028.

R. J. Lohner (2001), On the ubiquity of the wrapping effect in the computation of error bounds, in *Perspectives on Enclosure Methods* (U. Kulisch *et al.*, eds), Springer, pp. 201–216.

T. Lynch and E. Swartzlander (1992), A formalization for computer arithmetic, in *Computer Arithmetic and Enclosure Methods* (L. Atanassova and J. Hertzberger, eds), Elsevier Science, pp. 137–145.

M. A. Malcolm (1971), On accurate floating-point summation, *Commun. Assoc. Comput. Mach.* **14**, 731–736.

P. Markstein (1990), Computation of elementary functions on the IBM RISC System/6000 processor, *IBM J. Res. Dev.* **34**, 111–119.

W. F. Mascarenhas (2016), Floating point numbers are real numbers. Available at arXiv:1605.09202.

D. W. Matula (1968), In-and-out conversions, *Commun. Assoc. Comput. Mach.* **11**, 47–50.

G. Melquiond (2019), Formal verification for numerical computations, and the other way around. Habilitation à Diriger des Recherches, Université Paris Sud, Orsay.

M. Mezzarobba (2010), NumGfun: A package for numerical and analytic computation with D-finite functions, in *Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation*, ACM, pp. 139–145.

M. Mezzarobba (2020), Rounding error analysis of linear recurrences using generating series. Available at arXiv:2011.00827.

P. Micikevicius, D. Stosic, N. Burgess, M. Cornea, P. Dubey, R. Grisenthwaite, S. Ha, A. Heinecke, P. Judd, J. Kamalu, N. Mellempudi, S. Oberman, M. Shoeybi, M. Siu and H. W (2022), FP8 formats for deep learning. Available at https://paperswithcode.com/paper/fp8-formats-for-deep-learning.

O. Møller (1965), Quasi double-precision in floating-point addition, *BIT* **5**, 37–50.

D. Monniaux (2008), The pitfalls of verifying floating-point computations, *ACM Trans. Program. Lang. Syst.* **30**, 1–41.

J. S. Moore, T. Lynch and M. Kaufmann (1998), A mechanically checked proof of the correctness of the kernel of the AMD5K86 floating point division algorithm, *IEEE Trans. Comput.* **47**, 913–926.

R. E. Moore (1979), *Methods and Applications of Interval Analysis*, SIAM Studies in Applied Mathematics, SIAM.

R. E. Moore, R. B. Kearfott and M. J. Cloud (2009), *Introduction to Interval Analysis*, SIAM.

J.-M. Muller (2015), On the error of computing $ab+cd$ using Cornea, Harrison and Tang's method, *ACM Trans. Math. Softw.* **41**, 7:1–7:8.

J.-M. Muller (2016), *Elementary Functions, Algorithms and Implementation*, third edition, Birkhäuser.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

J.-M. Muller and L. Rideau (2022), Formalization of double-word arithmetic, and comments on 'Tight and rigorous error bounds for basic building blocks of double-word arithmetic', *ACM Trans. Math. Softw.* **48**, 1–24.

J.-M. Muller, N. Brunie, F. de Dinechin, C.-P. Jeannerod, M. Joldeş, V. Lefèvre, G. Melquiond, N. Revol and S. Torres (2018), *Handbook of Floating-Point Arithmetic*, second edition, Birkhäuser.

A. Neumaier (1974), Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen, *ZAMM* **54**, 39–51. In German.

A. Neumaier (1990), *Interval Methods for Systems of Equations*, Cambridge University Press.

Y. Nievergelt (2003), Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit, *ACM Trans. Math. Softw.* **29**, 27–48.

B. Noune, P. Jones, D. Justus, D. Masters and C. Luschi (2022), 8-bit numerical formats for deep neural networks. Available at arXiv:2206.02915.

T. Ogita, S. M. Rump and S. Oishi (2005), Accurate sum and dot product, *SIAM J. Sci. Comput.* **26**, 1955–1988.

F. W. J. Olver (1983), Error analysis of complex arithmetic, in *Computational Aspects of Complex Analysis*, Vol. 102 of NATO Science Series C, D. Reidel, pp. 279–292.

J. Osorio, A. Armejach, E. Petit, G. Henry and M. Casas (2022), A BF16 FMA is all you need for DNN training, *IEEE Trans. Emerg. Topics Comput.* **10**, 1302–1314.

M. L. Overton (2001), *Numerical Computing with IEEE Floating Point Arithmetic*, SIAM.

K. Ozaki, T. Ogita and D. Mukunoki (2021), Interval matrix multiplication using fast low-precision arithmetic on GPU, in *9th International Workshop on Reliable Engineering Computing*, pp. 419–434.

K. Ozaki, T. Ogita, S. Oishi and S. M. Rump (2012), Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications, *Numer. Algorithms* **59**, 95–118.

D. S. Parker, B. Pierce and P. R. Eggert (2000), Monte Carlo arithmetic: How to gamble with floating point and win, *Comput. Sci. Engng* **2**, 58–68.

M. Pichat (1972), Correction d'une somme en arithmétique à virgule flottante, *Numer. Math.* **19**, 400–406. In French.

M. Pichat (1976), Contribution à l'étude des erreurs d'arrondi en arithmétique à virgule flottante. PhD thesis, Université Scientifique et Médicale de Grenoble & Institut National Polytechnique de Grenoble.

S. Pion (1999), De la géométrie algorithmique au calcul géométrique. PhD dissertation, Université Nice Sophia Antipolis.

V. Popescu (2017), Towards fast and certified multiple-precision librairies. PhD dissertation, Université de Lyon, no. 2017LYSEN036.

Posit Working Group (2022), Standard for posit arithmetic. Available at https://posithub.org/docs/posit_standard-2.pdf.

D. M. Priest (1991), Algorithms for arbitrary precision floating point arithmetic, in *10th IEEE Symposium on Computer Arithmetic*, pp. 132–143.

D. M. Priest (1992), On properties of floating-point arithmetics: Numerical stability and the cost of accurate computations. PhD thesis, University of California at Berkeley.

D. M. Priest (2004), Efficient scaling for complex division, *ACM Trans. Math. Softw.* **30**, 389–401.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

N. Revol and F. Rouillier (2005), Motivations for an arbitrary precision interval arithmetic and the MPFI library, *Reliable Computing* **11**, 275–290.
E. J. Riedy and J. Demmel (2018), Augmented arithmetic operations proposed for IEEE-754 2018, in *25th IEEE Symposium on Computer Arithmetic*, pp. 45–52.
P. Roux (2014), Innocuous double rounding of basic arithmetic operations, *J. Formal. Reasoning* **7**, 131–142.
S. M. Rump (2009), Ultimately fast accurate summation, *SIAM J. Sci. Comput.* **31**, 3466–3502.
S. M. Rump (2010), Verification methods: Rigorous results using floating-point arithmetic, *Acta Numer.* **19**, 287–449.
S. M. Rump (2012), Error estimation of floating-point summation and dot product, *BIT Numer. Math.* **52**, 201–220.
S. M. Rump (2015), Computable backward error bounds for basic algorithms in linear algebra, *Nonlinear Theory Appl. IEICE* **6**, 360–363.
S. M. Rump (2017), IEEE754 precision-k base-β arithmetic inherited by precision-m base-β arithmetic for $k < m$, *ACM Trans. Math. Softw.* **43**, 20:1–20:15.
S. M. Rump (2019), Error bounds for computer arithmetics, in *26th IEEE Symposium on Computer Arithmetic*, pp. 1–14.
S. M. Rump, T. Ogita and S. Oishi (2008), Accurate floating-point summation, I: Faithful rounding, *SIAM J. Sci. Comput.* **31**, 189–224.
S. M. Rump, P. Zimmermann, S. Boldo and G. Melquiond (2009), Computing predecessor and successor in rounding to nearest, *BIT Numer. Math.* **49**, 419–431.
C. Severance (1998), IEEE 754: An interview with William Kahan, *Computer* **31**, 114–115.
J. R. Shewchuk (1997), Adaptive precision floating-point arithmetic and fast robust geometric predicates, *Discrete Comput. Geom.* **18**, 305–363.
N. Shibata and F. Petrogalli (2020), SLEEF: A portable vectorized library of C standard mathematical functions, *IEEE Trans. Parallel Distrib. Syst.* **31**, 1316–1327.
A. Sibidanov, P. Zimmermann and S. Glondu (2022), The CORE-MATH project, in *29th IEEE Symposium on Computer Arithmetic*, pp. 26–34.
R. L. Smith (1962), Algorithm 116: Complex division, *Commun. Assoc. Comput. Mach.* **5**, 435.
G. L. Steele Jr and J. L. White (2004), Retrospective: How to print floating-point numbers accurately, *ACM SIGPLAN Notices* **39**, 372–389.
P. H. Sterbenz (1974), *Floating-Point Computation*, Prentice-Hall.
G. W. Stewart (1985), A note on complex division, *ACM Trans. Math. Softw.* **11**, 238–241.
C. Strachey (1959), On taking the square root of a complex number, *Comput. J.* **2**, 89.
X. Sun, N. Wang, C.-Y. Chen, J. Ni, A. Agrawal, X. Cui, S. Venkataramani, K. El Maghraoui, V. V. Srinivasan and K. Gopalakrishnan (2020), Ultra-low precision 4-bit training of deep neural networks, in *Advances in Neural Information Processing Systems 33* (H. Larochelle *et al.*, eds), Curran Associates, pp. 1796–1807.
E. E. Swartzlander and A. G. Alexpoulos (1975), The sign-logarithm number system, *IEEE Trans. Comput.* Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE, 1990.
Y. Uguen and F. de Dinechin (2017), Design-space exploration for the Kulisch accumulator. Available at hal-01488916.
G. W. Veltkamp (1968), ALGOL procedures voor het berekenen van een inwendig product in dubbele precisie. Technical report 22, RC-Informatie, Technische Hogeschool Eindhoven.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press

G. W. Veltkamp (1969), ALGOL procedures voor het rekenen in dubbele lengte. Technical report 21, RC-Informatie, Technische Hogeschool Eindhoven.

S. Wang and P. Kanwar (2019), Bfloat16: The secret to high performance on cloud TPUs. Available at https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus.

R. C. Whaley, A. Petitet and J. J. Dongarra (2001), Automated empirical optimizations of software and the ATLAS project, *Parallel Comput.* **27**, 3–35.

J. H. Wilkinson (1960), Error analysis of floating-point computation, *Numer. Math.* **2**, 319–340.

J. H. Wilkinson (1961), Error analysis of direct methods of matrix inversion, *J. Assoc. Comput. Mach.* **8**, 281–330.

J. H. Wilkinson (1963), *Rounding Errors in Algebraic Processes*, Notes on Applied Science no. 32, HMSO. Also published by Prentice-Hall. Reprinted by Dover, 1994.

J. H. Wilkinson (1965), *The Algebraic Eigenvalue Problem*, Oxford University Press.

J. M. Wolfe (1964), Reducing truncation errors by programming, *Commun. Assoc. Comput. Mach.* **7**, 355–356.

I. Yamazaki, S. Tomov and J. Dongarra (2015), Mixed-precision Cholesky QR factorization and its case studies on multicore CPU with multiple GPUs, *SIAM J. Sci. Comput.* **37**, C307–C330.

A. Ziv (1999), Sharp ULP rounding error bound for the hypotenuse function, *Math. Comp.* **68**, 1143–1148.

https://doi.org/10.1017/S0962492922000101 Published online by Cambridge University Press