Contributions to floating-point arithmetic : Coding and correct rounding of algebraic functions

Adrien Panhaleux

École Normale Supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme

THÈSE

dont la soutenance est prévue le 27 juin 2012 par

Adrien Panhaleux

en vue de l'obtention du grade de

Docteur de l'École Normale Supérieure de Lyon – Université de Lyon
spécialité : Informatique

au titre de l'École Doctorale Informatique et Mathématiques de Lyon

Contributions à l'arithmétique flottante : codages et arrondi correct de fonctions algébriques

Commission d'examen:

Dominique Michelucci	Professeur, Université de Bourgogne	Rapporteur
Jean-Marie Chesneaux	Professeur, Université Paris 6	Rapporteur
Philippe Langlois	Professeur, Université de Perpignan	Examinateur
Christiane Frougny	Professeur, Université Paris 7	Examinatrice
Jean-Michel Muller	Directeur de recherches, ENS de Lyon	Directeur de thèse
Nicolas Louvet	Maître de conférences, Université Lyon 1	Co-encadrant

Contents

List of Figures

List of Tables

List of Algorithms

Remerciements

Je remercie tout d'abord Jean-Michel Muller et Nicolas Louvet, mes encadrants de thèse, dont la grande patience et les nombreux conseils m'ont permis de produire ce manuscrit et les résultats qu'il contient. Je remercie mes rapporteurs Dominique Michelucci et Jean-Marie Chesneaux, ainsi que Philippe Langlois, qui ont accepté de constituer mon jury. Je suis sincèrement désolé que Christiane Frougny n'ait pas pu participer à ce jury, pour des raisons familiales.

Je remercie également mes co-auteurs: Peter Kornerup, qui m'a accueilli dans ce beau pays qu'est le Danemark pour commencer un travail lors de mon stage de master qui a débouché sur de nombreuses reflexions sur les RN-codes lors de ma thèse; et Claude-Pierre Jeannerod qui, grâce à sa volonté d'être toujours plus précis, a beaucoup éclairci les travaux sur les breakpoints des fonctions algébriques simples.

Je remercie également toute l'équipe Arénaire, dont l'accueil fut chaleureux et sympathique. Tout particulièrement, je remercie les différents membres de cette équipe avec qui j'ai partagé un bureau à un moment ou un autre lors de cette thèse: Sylvain Chevillard pour son bon sens et son expérience, Ivan Morel pour sa grande expertise de Magic et Diablo, Érik Martin-Dorel pour sa spontanéité incongrue, Philippe Théveny et Adeline Langlois pour leur bonne humeur constante. Je remercie tout particulièrement Christophe Mouilleron, nos trois années de thèse dans le même bureau ayant donné bien trop de bons souvenirs pour que je puisse les résumer.

Je remercie grandement ma famille, mes parents et mon frère Lionel qui ont toujours été là pour me soutenir et me permettre d'aller jusqu'au bout. Merci à mon cousin Sylvain, qui est comme un frère pour moi. Merci à mes cousins et cousines, à mes grand-parents, et toute ma famille.

Merci aussi à tous mes amis:
- aux grands tacticiens des jeux de plateaux et de figurines pour toutes les parties titanesques passées présentes et à venir: Damien, Alexandre, Lionel et Valentin,
- aux danseurs et danseuses du club rock pour tout les bons moments passés en musique: Clémence, Audrey, Lucille, Antony et Marion, Cyril et Chloé, Toun et Manon, Aurore, Lise, Mac,
- aux wikipédiens en folie pour leur cynisme et leur sagesse: idealites, Poulpy, Ludo, Dereckson, Zil, nojhan, Harmonia, Ælfgar, esby

Bref, merci à toutes les personnes que j'ai pu rencontrer lors de l'élaboration de ce présent manuscrit, et qui ont su bon gré mal gré m'endurer lors de ses quatre années de thèse.

Introduction

Finite precision floating-point numbers are widely used for approximating the set of the reals on general purpose computers. Other ways of representing numbers exist, such as fixed-point numbers that are often used for signal processing and in embedded applications. But floating-point arithmetic provides a good compromise between speed, accuracy, and range of representation for most numerical applications. This includes various applications, such as the computation of approximate solution to partial differential equations that appear in many problems of physics (weather prediction, fluid dynamics, etc.), virtual monetary transfers, and many critical applications (plane control, aerospace systems, etc.).

The first modern implementation of binary floating-point arithmetic was probably in Konrad Zuse's Z3 computer in 1941 [7]. Floating-point arithmetic was already commonly used in the mid 1950s, and during the next two decades, almost every computer manufacturer developed its own floating-point system. Portability of floating-point programs became a real issue, as each processor designer was using a different floating-point hardware. For instance, while most processors during this period were using binary floating-point arithmetic, the IBM 360/370 series were using hexadecimal floating-point arithmetic [47, p. 17]: As it will be recalled later in this document, changing the radix in a sequence of floating-point operations also changes the rounding errors generated. As a consequence, performing the same computation with two computers using different radices may lead to different computed results. Moreover, not all processors incorporated a guard digit, which may lead to serious inaccuracies for instance when performing a floating-point subtraction: This was the case for the early versions of the IBM 360 [47, p. 17 and p. 36], and in the 1990s, CRAY supercomputers still did not have a guard bit [24, p. 44] and [47, p. 17]. That poor situation was described in 1981 in Kahan's paper *Why do we need a floating-point standard* [30].

The IEEE 754 *Standard for Binary Floating-Point Arithmetic* [25] was developed in the late 1970s and early 1980s, mainly as an effort to make floating-point programs easier to write in a portable manner. Released in 1985, IEEE 754 standardizes both the formats of representation of binary floating-point numbers on computers and the semantics of the five basic floating-point operations (addition, subtraction, multiplication, division, and square-root), and some conversions between decimal and binary numbers. A major feature of the standard is the requirement for *correct rounding* of the basic operations: The standard specifies four rounding modes (round to nearest, round toward positive, round toward negative, and round toward zero), and the result of every basic floating-point operation must be rounded according to one of these rounding modes; this constraint is called *correct rounding*, and essentially means that the result of every basic floating-point operation is uniquely defined using a well specified rounding function.

The IEEE 754 standard also specifies non-numerical encodings used in floating-point

arithmetic, such as infinities, or the result of invalid operations (such as 0/0), and how these special values behave with basic operations.

Having a standardized floating-point arithmetic permits to easily write portable software that works across various computers, but it also permits to prove the behavior of numerical software independently of the hardware being used.

Since the 1970s, decimal floating-point arithmetic has mainly been used in banking and financial applications. The IEEE 854 *Standard for Radix Independent Floating-Point Arithmetic* [26], which partly generalized the binary standard 754, was released in 1987 to cope with both binary and decimal floating-point arithmetic.

From 2001 to 2008, IEEE 754 and 854 have been under revision. The new IEEE 754-2008 *Standard for Floating-Point Arithmetic* [27] merges the two previous standards and brings significant improvements. Due to the constant evolution in hardware floating-point units design, computer arithmetic, and numerical algorithm design, new functionalities needed to be added to the floating-point arithmetic standard. The new standard incorporates some features that had become common practice, but that were not present in IEEE 754-1985; It also takes into account new research results that allowed one to easily perform computations that were previously thought impracticable or too expensive. In particular, the new standard incorporates the following features:

- The *Fused Multiply-Add* (FMA) instruction has been introduced in the standard. The FMA evaluates the expression $a \times b + c$ with only one final rounding, and this property can be used to approximately halve the number of roundings in many floating-point algorithms, such as dot products or polynomial evaluation algorithms [24, p. 46]. This instruction was already available in the instruction set of some processors prior to the release of the revised standard, such as the IBM PowerPC, HP/Intel IA-64 and HAL/Fujitsu SPARC64 VI. New *Graphical Processing Units* (GPU) architectures, such as NVIDIA GeForce 200, GeForce 400 and GeForce 500 Series, or the AMD HD 5000 Series [58], now also include correctly rounded floating-point FMA operators. As was first noticed by Markstein [42], the availability of an FMA instruction is also particularly useful for implementing correctly rounded floating-point division, square-root, or reciprocal square-root in software. This was already the case in binary floating-point arithmetic on the HP/Intel IA-64 based computers using Newton-Raphson iterations (see [43] and [46, p. 262]).

- The new standard fully specifies decimal floating-point arithmetic, in particular, the decimal interchange format encodings and the arithmetical properties needed for financial computations. A hardware decimal floating-point unit can already be found in the IBM POWER processors, since the POWER6 processor core (see for example [46, p. 107]). Intel mainly provides a software decimal floating-point library that uses binary hardware [8, 23]. The IEEE 754-2008 standard specifies two encodings for the decimal floating-point numbers: one is more suited for hardware implementation, and the other is more suited for software implementation. Notice that the set of representable numbers is the same in both cases: only the internal encodings differ.

- After more than 20 years of work on floating-point algorithms, it became clear that it was possible to efficiently implement more correctly-rounded functions than the ones

standardized in 1985, like conversion between decimal and binary floating-point numbers in the full range of floating-point numbers, or the evaluation of transcendental functions. In particular, the CRLIBM Library [11, 36] showed that it was possible to efficiently implement correctly rounded elementary functions and the standard now recommends (yet does not require) that some transcendental functions should be correctly rounded.

It is usually easier to compute the correct rounding of a function using a higher internal precision for intermediate computations. For example, to round the quotient of two 24-bit precision floating-point numbers, it is possible to first compute that quotient using 53 bits of precision, and then to round the floating-point result to 24 bits of precision. However, doing so raises another issue, called the *double-rounding* problem, which may in some cases lead to a wrong result. In the first part of this thesis, we present a special encoding of numbers called the RN-coding, different from the IEEE 754 encodings, which does not present the double-rounding issue. We then present how to perform sound and efficient arithmetic with this encoding, for fixed-point and floating-point RN-codings. This work on RN-coding arithmetic was published in *IEEE Transaction on Computers* [35].

When implementing correctly rounded elementary functions, a major difficulty that arises is what is called the Table-Maker's Dilemma [40], [46, chap. 12]. Given a function $f : \mathbb{R} \to \mathbb{R}$, let us assume we want to compute a correct rounding of $f(x)$, where x is a floating-point number. Usually, this is done by computing some "sufficiently accurate" approximation $\hat{f}$ to $f(x)$, and then by rounding $\hat{f}$. This strategy only works if we know that there is no discontinuity point of the rounding function between $\hat{f}$ and $f(x)$, i.e. no breakpoint. Making sure this situation does not happen is called the Table Maker's Dilemma: The closer $f(x)$ is to a breakpoint, the more accurately $\hat{f}$ needs to be computed to ensure correct rounding. In particular, when $f(x)$ is a breakpoint, the only way to ensure a correct rounding is to obtain $\hat{f} = f(x)$ exactly. When there are floating-point numbers x such that $f(x)$ is a breakpoint, we say that f *admits* breakpoints. Hence, to design a fast algorithm to correctly round a function f, two questions arise. For a given function f, does f admit breakpoints? And how close can $f(x)$ be to a breakpoint? Chapter 3 of this thesis focuses on the former question for some algebraic functions:

$$\sqrt{x},\ \frac{1}{\sqrt{x}},\ x^k \text{ for } k \in \mathbb{N}_{>0},\ \frac{x}{\|y\|_2},\ \frac{x}{y},\ \frac{1}{y},\ \frac{1}{\sqrt{x^2+y^2}},\ \frac{x}{\sqrt{x^2+y^2}},\ \sqrt{x^2+y^2}.$$

We study whether these functions admit breakpoints for both binary and decimal floating-point arithmetic. For the functions that admit breakpoints, we provide when it is possible a simple characterization of the entries x for which $f(x)$ is a breakpoint. This study on the breakpoints of some algrebraic functions was published in *IEEE Transaction on Computers* [29].

To compute the quotient of two floating-point numbers, there are essentially two classes of algorithms: the ones based on digit recurrence algorithms, such as the SRT division [13, 52, 55], and the Newton-Raphson division algorithms [42, 43, 46]. The SRT division convergence is known to be linear, whereas the Newton-Raphson algorithm converges quadratically. Despite their linear rate of convergence, SRT-based algorithms are frequently used in hardware for single and double precision floating-point formats, because they provide a good trade-off between speed and the logic area. However, as noticed for instance by Markstein [42] or by Kahan [31], the availability of an FMA instruction makes

it possible to design fast algorithms for correctly rounded division in software based on the Newton-Raphson algorithm. For instance, binary floating-point division was implemented solely in software on the IBM RS/6000 [42], and, later on, on the HP/Intel Itanium [9, 43]. The proofs of correct rounding provided by Markstein and Harisson [22] only handle binary floating-point arithmetic, and do not work in other radices. The introduction of the FMA instruction for decimal arithmetic in the revised IEEE 754-2008 standard raises the issue of implementing decimal floating-point division in software using Newton-Raphson iterations.

In Chapter 4, we study general results on variants of the Newton-Raphson division algorithm using FMA instructions: there are two ways of implementing one step of Newton-Raphson iteration for division using an FMA instruction: the Markstein iteration [43] and the Goldschmidt iteration [14, 19]. They have different advantages and drawbacks, so that it make sense to use both kinds of iteration in a given calculation to compute the quotient of two floating-point numbers [43]. Our study is done for radix β floating-point arithmetic, focusing more particularly on the cases $\beta = 2$ and $\beta = 10$, and on rounding to the nearest. However, the final rounding technique used by Markstein in radix 2 cannot be directly generalized in radix 10: Indeed, while in radix 2 the quotient of two floating-point numbers cannot be a midpoint in the range of normal floating-point numbers, this is no longer the case in radix 10 (see Chapter 3). Therefore, we generalize some results proved by Markstein for radix 2 to radix β, and also provide a tailored final rounding technique in radix 10. This work on Newton-Raphson based divised was presented during the *IEEE Application-specific Systems Architectures and Processors* international conference [41].

Chapter 1

Floating-point arithmetic

This chapter is an introductory chapter: it recalls the definitions related to floating-point arithmetic, and presents some of the features of the IEEE-754-2008 Standard for Floating-Point Arithmetic [27] that are used or studied through the rest of this thesis. In Section 1.3, we recall some elementary facts about rounding functions and breakpoints, and in Section 1.4, we present the concept of correct rounding. We present in Section 1.5 what are the main issues related to correct rounding that we will deal with in the next chapters. This chapter is largely inspired by [46, chap. 1, 2 and 3].

1.1 Floating-point numbers

We first present a general definition of both normal and subnormal floating-point numbers, and introduce the notations that are used in this document.

1.1.1 Scientific notation

Every real number x can be represented using the "scientific notation"

$$\pm\, m_x \cdot \beta^{e_x},$$

where β is the radix in which the number is represented, m_x is a real number called *significand* of the representation, and e_x is an integer, called *exponent*. For example, in radix 10, the real number

$$x_{\text{real}} = 3718.50237499999950000\ldots \cdot 10^{-393}$$

has a significand $m_x = 3718.5023749999995$ and an exponent $e_x = -393$.

However, that scientific notation is not uniquely defined. For every non-null number, the representation is normalized to make it unique. We will say that the representation of a non-null number is *normal* if the significand is such that $1 \leq m_x < \beta$. In that case, the exponent e_x is unique, and its value is $\lfloor \log_\beta |x| \rfloor$. For example, the normal representation of x_{real} is

$$x_{\text{real}} = 3.71850237499999950000\ldots \cdot 10^{-390},$$

and one can check that $-390 = \lfloor \log_{10} |x_{\text{real}}| \rfloor$.

1.1.2 Limited significand

In order to remain efficient and fast in computer arithmetic, arbitrarily large representations of numbers cannot be used. Hence, in computer hardware and software, we use finite-precision floating-point numbers to represent real numbers. The first step is to limit the size of the significand. A *normal floating-point number*, is a non-null number such that its significand m_x is a rational of the form

$$m_x = (m_0.m_1 m_2 \dots m_{p-1})_\beta \,,$$

where p is the *precision* of the format. The set of radix-β, precision-p floating-point numbers will be noted $\mathbb{F}_{\beta,p}$.

For a fixed exponent e, the set of normal floating point numbers comprised in the interval $[\beta^e, \beta^{e+1})$ will be called a *betade*. For any non-null normal floating-point number z in a given betade, i.e. $z \in \mathbb{F}_{\beta,p} \bigcap [\beta^{e_z}, \beta^{e_z+1})$, we define the *unit in the last place*, noted $\mathrm{ulp}(z)$, as $\mathrm{ulp}(z) = \beta^{e_z+1-p}$. We extend this usual definition to all non-null real number: For all $z \in [\beta^{e_z}, \beta^{e_z+1})$, we similarly define $\mathrm{ulp}(z)$ as β^{e_z+1-p}. The unit in the last place is often used to quantify rounding errors, since rounding a real number z to its nearest normal floating-point number z_{float} gives a error smaller than $0.5\mathrm{ulp}(z)$.

For example, if we choose to represent radix 10 numbers using 16 digits of precision, the previous

$$x_{\mathrm{real}} = \underbrace{3.718502374999999}_{\text{16 digits}} 50000 \dots \cdot 10^{-390}$$

will have to be rounded to one of the following two normal floating-point numbers (in $\mathbb{F}_{10,16}$):

$$x_{\mathrm{float}} = 3.718502375000000 \cdot 10^{-390}, \qquad x'_{\mathrm{float}} = 3.718502374999999 \cdot 10^{-390}.$$

The previous number x_{real} is exactly between two consecutive floating-point numbers of the set $\mathbb{F}_{\beta,p}$. This particular case is what we call a *breakpoint*, which will be explained in §1.3.3. In order to be consistent with the standard rounding functions defined in §1.3.1, we will choose to round to x_{float} instead of x'_{float}.

One can also represent the same set of floating-point numbers $\mathbb{F}_{\beta,p}$ using *integral significands*. Instead of using the standard normalization of the significand $1 \leq m_x < \beta$, the same floating-point number can be represented using an integer M_x for its significand, with $\beta^{p-1} \leq M_x < \beta^p$:

$$x = \pm\, m_0 m_1 m_2 \dots m_{p-1} m_p \cdot 10^{e_x - p + 1}.$$

For example, x_{float} can be represented with an integral significand:

$$x_{\mathrm{float}} = 3718502375000000 \cdot 10^{-405}.$$

1.1.3 Limited exponent range

The second step toward floating-point formats is to limit the range of the exponent. To avoid arbitrarily large exponents, a *minimum exponent* $e_{\min}$ and *maximum exponent* $e_{\max}$ need to be specified.

For example, the IEEE-754-2008 standard specifies for the decimal64 format (Section 1.2) 16 digits of precision in radix 10, with $e_{\min} = -383$ and $e_{\max} = 384$.

However, if we only have normal floating-point numbers, with limited exponent range, we observe a gap around 0, as shown in Figure 1.1. In order to avoid that gap, and smooth the repartition of floating-point numbers around 0, floating-point formats introduced the notion of "subnormal numbers". A *subnormal floating-point number* x is of the form

$$\pm m_x \cdot \beta^{e_{\min}},$$

where $m_x = (0.m_1 m_2 \dots m_{p-1})_\beta < 1$. To remain consistent with subnormal numbers, zero is represented as $0 \cdot \beta^{e_{\min}}$ in floating-point formats.

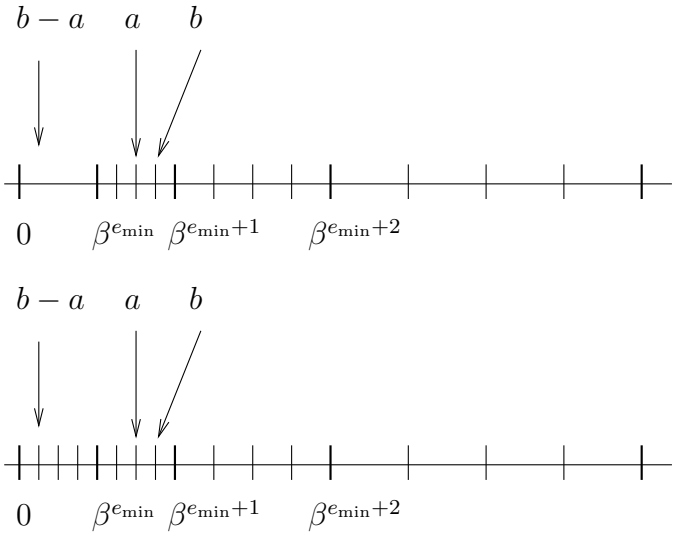


Figure 1.1: The positive floating-point numbers in the toy system $\beta = 2$ and $p = 3$. Above: normal floating-point numbers only. In that set, $b - a$ cannot be represented, so that the computation of $b - a$ will be rounded to 0. Below: the subnormal numbers are included in the set of floating-point numbers.

Our previous example is such that

$$x_{\text{real}} = \underbrace{0.000000371850237}_{\text{16 digits}}499999950000\ldots \cdot 10^{-383}.$$

If we want to represent this real number in the decimal64 format ($\mathbb{F}_{10,16}$, with $e_{\min} = -383$), it will be rounded as a subnormal number:

$$x_{\text{subnormal}} = 0.000000371850237 \cdot 10^{-383}.$$

Notice that if, instead of rounding x_{real}, we round the previously already rounded number

$$x_{\text{float}} = \underbrace{0.000000371850237}_{\text{16 digits}}5000000 \cdot 10^{-383},$$

we can round this number to one of the two following subnormal floating-point numbers:

$$x_{\text{subnormal}} = 0.000000371850237 \cdot 10^{-383}, \quad \text{or} \quad x_{\text{wrong}} = 0.000000371850238 \cdot 10^{-383}.$$

In fact, according to rounding rules (§1.3.1), x_{float} will be rounded to x_{wrong}. The fact that rounding twice a number can give a wrong result is a well-known issue called *Double rounding* [5, 17, 18, 44]: more details on that topic will be given in §1.5.2.

1.2 The IEEE-754-2008 standard

Since 1985, floating-point formats have been specified by the IEEE-754 standard [25], in order to have the same formats with the same behavior on every computer. Due to the constant evolution in computers and algorithms, the standard had to be revised. In 2008, a revised standard for floating-point arithmetic, IEEE-754-2008 [27], was then published.

1.2.1 Floating-point formats

The IEEE-754-2008 standard defines five basic floating-point formats, as well as several interchange floating-point formats, extended formats and extendable formats [27, section 3]. Basic formats are formats used for computing in floating-point arithmetic. Interchange formats are used for the exchange of floating-point data between implementations. Extended formats, while not required by the standard, can be used to extend a basic format with wider precision and exponent range. Extendable formats are the same as extended formats, except that precision and range are defined by the user. We will here specify basic and interchange formats, using our notations. A floating-point format is fully characterized by four parameters:

- The radix β, which is an integer larger than or equal to 2;
- The precision p, which is an integer larger than or equal to 1;
- The integer $e_{\min}$;
- The integer $e_{\max}$.

Once these parameters are specified, a floating-point number conforming the IEEE-754-2008 standard is either:

- A *normal floating-point number*, which is a number of the form

$$x = \pm \left(m_0.m_1 m_2 \dots m_{p-1}\right)_\beta \cdot \beta^{e_x},$$

  with $1 \leq m_x < \beta$ and $e_{\min} \leq e_x \leq e_{\max}$.

- A *subnormal floating-point number*, which is a number of the form

$$x = \pm \left(0.m_1 m_2 \dots m_{p-1}\right)_\beta \cdot \beta^{e_{\min}},$$

  with $m_x < 1$.

- A *special value*: qNaN, sNaN, $\pm\infty$ (briefly described in §1.2.2).

The standard specifies format only for radices 2 and 10. It also fixes $e_{\min}$ to the value $1 - e_{\max}$, for all formats. The binary interchange formats parameters are listed in Table 1.1, while the decimal interchange formats parameters are given in Table 1.2. The five basic formats correspond to the interchange formats **binary32, binary64, binary128, decimal64, decimal128**. It should be noticed that binary32 and binary64 correspond exactly to the single and double floating-point formats that were defined in the IEEE-754-1985 Standard [25].

Table 1.1: Parameters of the binary interchange formats in the standard IEEE-754-2008

	Binary ($\beta = 2$)				
	binary16	**binary32**	**binary64**	**binary128**	**binary**$\{k\}$
p	11	24	53	113	$k - \lfloor 4\log_2(k) \rceil + 13$
$e_{\max}$	+15	+127	+1023	+16383	$2^{k-p-1} - 1$
$e_{\min}$	-14	-126	-1022	-16382	-2^{k-p-1}

Table 1.2: Parameters of the decimal interchange formats in the standard IEEE-754-2008

	Decimal ($\beta = 10$)			
	decimal32	**decimal64**	**decimal128**	**decimal**$\{k\}$
p	7	16	34	$9k/32 - 2$
$e_{\max}$	+96	+384	+6144	$3 \times 2^{k/16+3}$
$e_{\min}$	-95	-383	-6143	$1 - 3 \times 2^{k/16+3}$

1.2.2 Special values

In addition to normal and subnormal numbers, the IEEE-754-2008 standard defines special values [27, ch. 6], in order to fully specify all operations on all floating-point entries. There are three types of special values: *NaNs*, *Infinities* and *Zeroes*.

- A NaN (*Not a Number*) is mainly used to represent the result of invalid operations, for example $\sqrt{-5}$ or $0/0$. The standard in fact defines two types of NaNs, *quiet* NaN and *signaling* NaN. For more information on NaNs, see Muller et al. [46, p. 69] or the IEEE-754-2008 standard [27, ch. 6].

- When the result of a floating-point operation overflows, the standard also specifies that the result is a special value called infinity ($+\infty$ or $-\infty$).

- Since the standard chose to keep the sign of the result in the infinities, and in order to remain mathematically consistent, the standard also defines two signed zeroes, namely $+0$ and -0. These two zeroes can lead to different results on some operations. For example, $1/+0 = +\infty$ while $1/-0 = -\infty$.

1.2.3 Implementation of floating-point formats

Only the definition of the floating-point formats given in §1.2.1 will be used in this document. However, the IEEE-754-2008 standard also specifies the actual bit-level representation of floating-point numbers, both for binary and decimal arithmetic. For the sake of completeness, these implementation issues and improvements are briefly explained in this subsection.

Binary

The first implementation issue in binary floating-point is the representation of the exponent field. To represent that signed integer, the standard has chosen to use a biased representation. For normal floating-point numbers, the exponent e_x is encoded as a biased exponent E_x, with $e_x = E_x - b$, b being the bias. Assuming the biased exponent is encoded on W_E bits, the bias for each binary format of the standard is $b = 2^{W_E-1}$. The value $E_x = 0$ is used to represent subnormal numbers, while $E_x = 2^{W_E} - 1$ is used to represent special values (§1.2.2).

Since the first bit of the significand in binary floating-point arithmetic is always a 1 when the number is normal, and always a 0 when it is subnormal, storing the first bit of the significand is not needed. Indeed, it suffices to check whether $E_x = 0$ to determine if the number is subnormal, and to determine the first bit of the significand. Hence, a binary floating-point number of precision p has its significand stored on only $p - 1$ bits. This unstored bit of the significand is called the *implicit bit*.

Given this two implementation requirements, binary floating-point formats are encoded as follows.

1 bit	W_E bits	$p-1$ bits
S (sign)	e'_x (biased exponent)	m'_x (trailing significand)

Example 1.1. Consider the following bit string, interpreted as a binary32 floating-point number x:

sign	exponent	trailing significand
0	01101011	01010101010101010101010

The biased exponent is neither 00000000 nor 11111111, so x is a normal floating-point number. The biased exponent's value is $e'_x = 107$. Since the bias in binary32 is 127, the exponent of x is $e_x = 107 - 127 = -20$.

Since x is a normal floating-point number, the implicit bit is a one. Hence, the significand of x is

$$m_x = (1.01010101010101010101010)_2 = \frac{5592405}{4194304}.$$

Also, the sign bit is 0, so $x \geq 0$. Hence, x is equal to

$$\frac{5592405}{4194304} \cdot 2^{-20} = 0.000001271565679417108185589313507080078125.$$

Decimal

The two possible bit-string representations of decimal floating-point numbers defined by the standard are a bit more complex than binary. Since the understanding of these implementations is not necessary for the topics of this thesis, we will only outline the key ideas behind these representations. For a more detailed presentation, see the standard [27] or Muller et al. [46, pp. 82-92].

- The standard defines two encodings for decimal floating-point numbers, named *decimal* and *binary* encodings. The binary encoding, more suited for a software implementation, encodes the decimal integral significand as a binary number. The decimal encoding, more suited for a hardware implementation, encodes a group of three consecutive decimal digits as a 10-bit string (a *declet*).

- The exponent range defined for the decimal formats (Table 1.2) is not a power of two. As a consequence, the exponent field and significand field in the bit string are not as clearly separated as in the binary formats.

- It is sometimes allowed to encode non-canonical floating-point numbers in decimal. For example, the decimal32 canonical floating-point number $5.296800 \cdot 10^{22}$ can also be represented as $0.052968 \cdot 10^{24}$.

1.3 Roundings

We present in this section the roundings defined by the IEEE-754-2008 standard, as well as the faithful rounding, and define breakpoints.

1.3.1 Definition

Since floating-point numbers are used to approximate real numbers on a computer, it is essential to define functions that transform a real number into a floating-point approximation of precision p. Such functions are called *rounding functions*. A "general" rounding function will be noted $\circ_p()$ in this manuscript. For any real number z, there is a floating-point number $x = \circ_p(z)$ which is the rounded value of z.

The IEEE-754-2008 defines 5 rounding functions:

- *roundTiesToEven* rounds the real number z to the nearest floating-point number. If z is exactly the middle of two consecutive floating-point numbers, it is rounded to the floating-point number whose least significand digit is even. We will note this rounding function $x = \mathrm{RN}_p(z)$. Notice that *roundTiesToEven* is the default rounding mode.

- *roundTiesToAway* rounds the real number z to the nearest floating-point number. If z is exactly the middle of two consecutive floating-point numbers, it is rounded to the floating-point number with the larger significand. We will note this rounding function $x = \mathrm{RA}_p(z)$. Also notice that *roundTiesToAway* is only required in decimal arithmetic (it is only used for financial calculations). [1]

- *roundTowardPositive* rounds the real number z to the nearest floating-point number that is no less than z. We will note this rounding function $x = \mathrm{RU}_p(z)$.

- *roundTowardNegative* rounds the real number z to the nearest floating-point number that is no greater than z. We will note this rounding function $x = \mathrm{RD}_p(z)$.

- *roundTowardZero* rounds the real number z to the nearest floating-point number that is no greater in magnitude than z. We will note this rounding function $x = \mathrm{RZ}_p(z)$.

[1] This rounding was not defined in the IEEE-754-1985 Standard.

Sometimes, when there is no ambiguity on the value of the precision p being used for the rounding, we will write

$$\mathrm{RN}(), \mathrm{RA}(), \mathrm{RU}(), \mathrm{RD}(), \mathrm{RZ}()$$

instead of

$$\mathrm{RN}_p(), \mathrm{RA}_p(), \mathrm{RU}_p(), \mathrm{RD}_p(), \mathrm{RZ}_p().$$

Two major properties of rounding functions are really important, and the first step towards most of the proofs on floating-point arithmetic. All the previously defined rounding functions are monotone, and the rounding of a floating-point number is always that floating-point number. More formally:

- If $x < y$, then for any of the five rounding functions of the IEEE-754-2008 standard, $\circ_p(x) \leq \circ_p(y)$.
- If x is a floating-point number of precision p, then for any $q \geq p$, and for any of the five rounding functions of the IEEE-754-2008 standard, $\circ_q(x) = x$.

1.3.2 Faithful rounding

Another rounding named the *faithful rounding* is often used in floating-point arithmetic. While this rounding is not a mathematical function, it is however often used in many proofs involving floating-point arithmetic [46, p. 22].

The floating-point number $x \in \mathbb{F}_{\beta,p}$ is said to be a *faithful rounding* of the real number z if:

- When z is a floating-point number ($z \in \mathbb{F}_{\beta,p}$), then $x = z$,
- When z is not in $\mathbb{F}_{\beta,p}$, then x is one of the two consecutive floating-point numbers surrounding z.

In this thesis, we use the faithful rounding only in Chapter 4, which will help us prove the correct rounding of the algorithms computing the quotient of two floating-point numbers.

1.3.3 Breakpoints

Another property that is shared by all five rounding functions defined in previous §1.3.1 is the discontinuities of the rounding functions over the reals. For example, Figure 1.2 shows the discontinuities of $\mathrm{RN}_2(x)$ on a small set of positive binary floating-point numbers : $p = 3$, $e_{\min} = -1$ and $e_{\max} = 1$.

Given the definition of each rounding functions, we see that there are two possibilities for the discontinuities.

- For the *roundTiesToEven* and *roundTiesToAway* functions, the discontinuities lie exactly halfway between two consecutive floating-point numbers. We will call *midpoints* the numbers at which there is a discontinuity for these two rounding functions. The set of midpoints, which depends on the radix β and the precision p, will be noted $\mathbb{M}_{\beta,p}$.

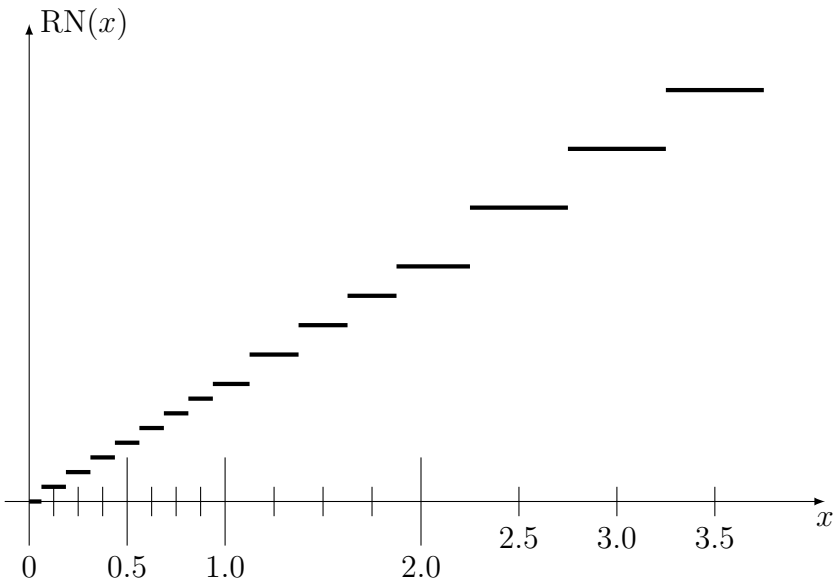


Figure 1.2: The $\mathrm{RN}_2(x)$ function for radix $\beta = 2$ and precision $p = 3$.

- For the *roundTowardPositive*, *roundTowardNegative* and *roundTowardZero* functions, the discontinuities lie on the floating-point numbers, except for *roundTowardZero* which is continuous at 0.

The existence of these discontinuities for the rounding functions yields serious problems when trying to evaluate a function, especially when we want to compute a correctly rounded function, as we will see later in §1.5.1.

1.4 Floating-point operations

We present in this section most functions that are considered in the IEEE-754-2008 standard: First, the five basic arithmetic operations that all IEEE implementations need to provide correctly rounded, and next the elementary functions that, if present, are recommended to be correctly rounded by the IEEE-754-2008 standard.

1.4.1 Basic operations

Several operations on floating-point formats are required by the IEEE-754-2008 standard [27, section 5]. In this thesis, we will call basic operations the six following required arithmetic operations:

- Addition(a, b) computes $x = \circ_p(a + b)$.
- Subtraction(a, b) computes $x = \circ_p(a - b)$.
- Multiplication(a, b) computes $x = \circ_p(a \times b)$.
- Division(a, b) computes $x = \circ_p(a/b)$.

- Square-root(a) computes $\circ_p(\sqrt{(a)})$.
- Fused Multiply-Add(a, b, c) (or FMA(a, b, c)) computes $\circ_p(a \times b + c)$.

It is required that all the basic operations yield a correctly rounded result. An operation is said to be correctly rounded if the computed floating-point number is the same as the one obtained if we round the real result of the operation according to the current rounding mode.

For example the real number resulting from the operation 2/3 would be

$$0.666666666666666666\ldots$$

Hence, the correctly rounded result of Division$(2, 3)$ in decimal64 ($p = 16$) in round-to-nearest mode is

$$0.6666666666666667 = \circ_{16}(0.666666666666666666\ldots)$$

1.4.2 Function evaluation

While only the basic operations are mandatory in the IEEE-754-2008 standard, one might want to compute the correctly rounded result of some other functions. The standard [27, section 9] recommends to provide an implementation of many functions, including exp, log, trigonometric functions, hyperbolic functions, power, ... (see Table 1.3 for a list of those functions): We will call these functions *elementary functions*. Also, in Chapter 3, we study some other functions that are linked to the normalization of vectors. While these functions are not mentioned by the IEEE-754-2008 standard, they are often used in several domains. Hence it would be a good idea to provide a fast correct rounding of these functions.

1.5 Issues related to correct rounding

In this last section, we finally present the mains issues related to correct rounding, which are the core of this work: the Table Maker's Dilemma, and the double-rounding error.

1.5.1 Correctly rounded function evaluation

A common way of evaluating the correctly rounded result of elementary functions over floating-point numbers is to first use *range reduction*, and then use a good polynomial or rational approximation to the targeted function. Range reduction is used to change the problem of evaluating the function on all floating-point numbers into evaluating the function on a small subset of floating-point numbers. A detailed explanation of the whole process is given in Muller et al. [46, chap. 11, 12].

Determining a good approximation can be done by separating the problem in two parts: computing an approximation, and proving the result of the approximation is the correctly rounded function. Given a function $f : \mathbb{R}^d \to \mathbb{R}$, the two parts can be expressed more precisely:

- Computing the approximation is, given a target accuracy ϵ, finding an algorithm $\hat{f} : (\mathbb{F}_{\beta,p})^d \to \mathbb{R}$ such that for any floating-point number x, $|\hat{f}(x) - f(x)| < \epsilon$.

Table 1.3: Recommended correctly rounded functions

Operation	Function	Domain
exp	e^x	$[-\infty, +\infty]$
expm1	$e^x - 1$	
exp2	2^x	
exp2m1	$2^x - 1$	
exp10	10^x	
exp10m1	$10^x - 1$	
log	$\log_e(x)$	$[0, +\infty]$
log2	$\log_2(x)$	
log10	$\log_{10}(x)$	
logp1	$\log_e(1+x)$	$[-1, +\infty]$
log2p1	$\log_2(1+x)$	
log10p1	$\log_{10}(1+x)$	
hypot(x, y)	$\sqrt{x^2+y^2}$	$[-\infty, +\infty] \times [-\infty, +\infty]$
rSqrt	$1/\sqrt{x}$	$[0, +\infty]$
compound(x, n)	$(1+x)^n$	$[-1, +\infty] \times \mathbb{Z}$
root(x, n)	$x^{\frac{1}{n}}$	$[-\infty, +\infty] \times \mathbb{Z}$
pown(x, n)	x^n	$[-\infty, +\infty] \times \mathbb{Z}$
pow(x, y)	x^y	$[-\infty, +\infty] \times [-\infty, +\infty]$
powr(x, y)	x^y	$[0, +\infty] \times [-\infty, +\infty]$
sin	$\sin(x)$	$(-\infty, +\infty)$
cos	$\cos(x)$	$(-\infty, +\infty)$
tan	$\tan(x)$	$(-\infty, +\infty)$
sinPi	$\sin(\pi \times x)$	$(-\infty, +\infty)$
cosPi	$\cos(\pi \times x)$	$(-\infty, +\infty)$
atanPi	$\frac{\operatorname{atan}(x)}{\pi}$	$[-\infty, +\infty]$
atan2Pi(y, x)	$\frac{2}{\pi} \operatorname{atan}\left(\frac{y}{\sqrt{x^2+y^2}+x}\right)$	$[-\infty, +\infty] \times [-\infty, +\infty]$
asin	$\operatorname{asin}(x)$	$[-1, +1]$
acos	$\operatorname{acos}(x)$	$[-1, +1]$
atan	$\operatorname{atan}(x)$	$[-\infty, +\infty]$
atan2(y, x)	$2 \operatorname{atan}\left(\frac{y}{\sqrt{x^2+y^2}+x}\right)$	$[-\infty, +\infty] \times [-\infty, +\infty]$
sinh	$\sinh(x)$	$[-\infty, +\infty]$
cosh	$\cosh(x)$	$[-\infty, +\infty]$
tanh	$\tanh(x)$	$[-\infty, +\infty]$
asinh	$\operatorname{asinh}(x)$	$[-\infty, +\infty]$
acosh	$\operatorname{acosh}(x)$	$[+1, +\infty]$
atanh	$\operatorname{atanh}(x)$	$[-1, +1]$

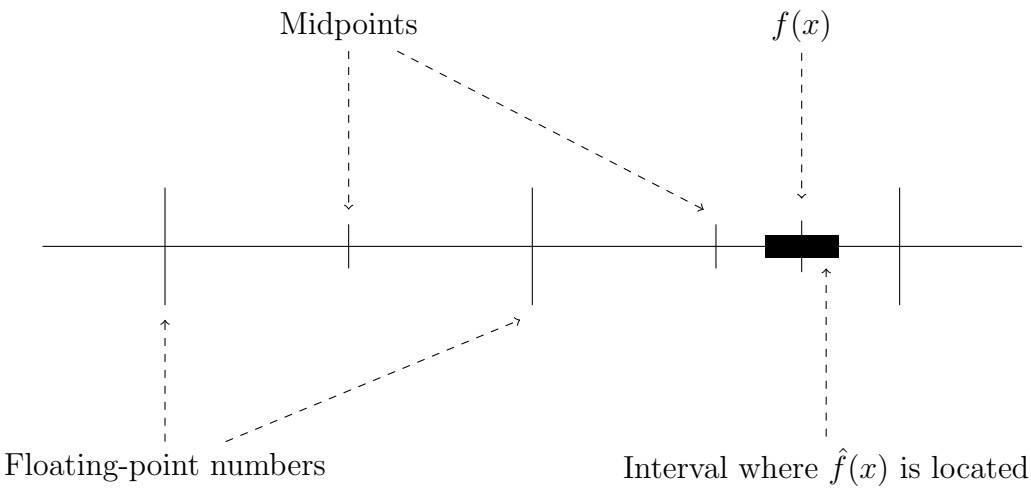


Figure 1.3: In this example (assuming rounding to nearest), the interval around $\hat{f}(x)$ where $f(x)$ is known to be located contains no breakpoint. Hence, $\mathrm{RN}(f(x)) = \mathrm{RN}(\hat{f}(x))$: we can safely return a correctly rounded result.

- Proving the correctly rounded result is, given a target precision p_{to}, proving that for all floating-point numbers x of input precision p_{in} there exists an accuracy ϵ such that if $|\hat{f}(x) - f(x)| < \epsilon$, then $\mathrm{RN}_p(\hat{f}(x)) = \mathrm{RN}_p(f(x))$.

If these two conditions are fulfilled, then using the approximation $\hat{f}$ yields the correctly rounded result of the function f for all floating-point numbers. The latter problem, also known as *solving the Table Maker's Dilemma*, is the main topic of chapters 3 and 4.

Solving the Table Maker's Dilemma for a given function f can be done by determining, for any floating-point number $x \in \mathbb{F}_{\beta,p}$, how close from a breakpoint can $f(x)$ be. If we know that $f(x)$ cannot be closer to a breakpoint than a distance δ, then it suffices to have $|\hat{f}(x) - f(x)| < \delta$ to prove a correctly-rounded result, as shown in Figures 1.3 and 1.4. This method will be mainly used for the computation of the quotient in Chapter 4.

However, solving the Table Maker's Dilemma can be done this way only if for any floating-point number x, $f(x)$ is not a breakpoint. Hence, determining for a given function if $f(x)$ can be a breakpoint is a crucial problem, for both proving the correctly rounded result and designing an approximation algorithm. Chapter 3 address this issue in details for some algebraic functions. Chapter 4 shows how to round correctly the division operation using the correctly rounded FMA operation, by searching a good approximation algorithm, solving the Table Maker's Dilemma, and dealing with the case where $f(x)$ is a breakpoint.

1.5.2 Double-rounding

In some cases, rounding several times the same number using decreasing successive precisions can yield a bigger error than when rounding only once with the smallest precision. For example, we saw previously that for

$$x_{\mathrm{real}} = 3.71850237499999950000\ldots \cdot 10^{-390},$$

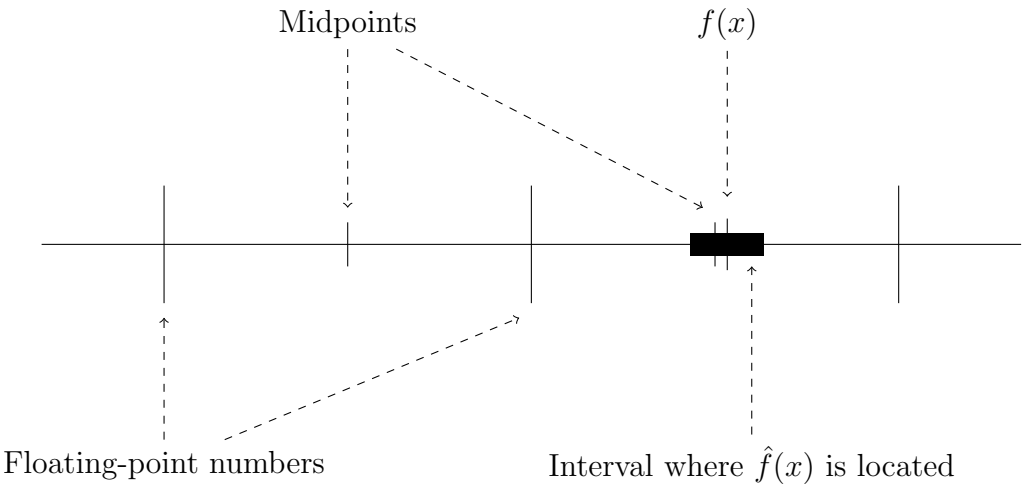


Figure 1.4: In this example (assuming rounding to nearest), the interval around $\hat{f}(x)$ where $f(x)$ is known to be located contains a breakpoint. We do not have enough information to provide a correctly rounded result.

one has

$$x_{\text{wrong}} = \text{RN}_9\left(\text{RN}_{16}\left(x_{\text{real}}\right)\right) = 3.71850238{\cdot}10^{-390} \neq 3.71850237{\cdot}10^{-390} = \text{RN}_9\left(x_{\text{real}}\right) = x_{\text{right}}.$$

During this thesis, we proposed the implementation of some basic operations on a new encoding of numbers named *RN-coding*. This encoding is specially designed to avoid this extra error due to double-rounding, and we provided an efficient and mathematically coherent implementation of addition, subtraction, multiplication and division for fixed-point and floating-point RN-codes.

Chapter 2

RN-coding

Recently a class of number representations denoted RN-Codings was introduced [34, 35], allowing rounding-to-nearest to take place by a simple truncation, with the property that errors due to double-roundings are avoided. They are based on a generalization of the observation that certain radix representations are known to possess this property. An example is the classical (binary) Booth recoding. Another one is the following: we can write a number in decimal using the digits $\{-5, -4, \ldots, 4, 5\}$ instead of the usual digit set $\{0, \ldots, 9\}$, with the constraint that the next non-null digit to the right of a 5 (resp. -5) is negative (resp. positive).

When addressing the issue of double-roundings in §1.5.2, we saw with the number

$$x_{\text{real}} = 3.71850237499999950000\ldots \cdot 10^{-390}$$

that rounding it first to 16 digits, and then to 9 digits, is not the same as rounding x_{real} only once to 9 digits:

$$x_{\text{wrong}} = \text{RN}_9\left(\text{RN}_{16}\left(x_{\text{real}}\right)\right) = 3.71850238 \cdot 10^{-390} \neq 3.71850237 \cdot 10^{-390} = \text{RN}_9\left(x_{\text{real}}\right).$$

However, if we write x_{real} using the digit set $\{\bar{5}, \bar{4}, \ldots, 4, 5\}$ with the constraint presented above ($\bar{5}$ representing the digit whose value is -5),

$$x_{\text{rn-coding}} = 4.\bar{3}2\bar{1}\bar{5}024\bar{3}500000\bar{5}0000\ldots \cdot 10^{-390}.$$

Now, rounding $x_{\text{rn-coding}}$ is simply done by truncating:

$$\text{RN}_{16}\left(x_{\text{rn-coding}}\right) = 4.\bar{3}2\bar{1}\bar{5}024\bar{3}5000000 \cdot 10^{-390},$$

$$\text{RN}_9\left(\text{RN}_{16}\left(x_{\text{rn-coding}}\right)\right) = 4.\bar{3}2\bar{1}\bar{5}024\bar{3} \cdot 10^{-390}.$$

This means that using this special representation, we have no double-rounding error: $\text{RN}_9\left(\text{RN}_{16}\left(x_{\text{rn-coding}}\right)\right) = \text{RN}_9\left(x_{\text{rn-coding}}\right)$. Notice that $\text{RN}_9\left(x_{\text{rn-coding}}\right) = 4.\bar{3}2\bar{1}\bar{5}024\bar{3} \cdot 10^{-390}$ represents as expected the correct result $3.71850237 \cdot 10^{-390}$, i.e., without the double-rounding error that previously occurred in the ordinary decimal representation.

In this chapter, we investigate in Section 2.1 a particularly efficient encoding of the binary representation. This encoding is generalized to any radix and digit set; however radix complement representations for odd values of the radix turn out to be particularly impractical. The encoding is essentially an ordinary radix complement representation with an appended

round-bit, but still allowing rounding to nearest by truncation and thus avoiding problems with double-roundings. Conversions from radix complement to these round-to-nearest representations can be performed in constant time, whereas conversion from round-to-nearest representation to radix complement in general takes at least logarithmic time.

In Section 2.2, we develop arithmetic operations for fixed-point RN-codings. Not only is rounding-to-nearest a constant time operation, but so is also sign inversion, both of which are at best log-time operations on ordinary 2's complement representations. Addition and multiplication on such fixed-point representations are first analyzed and defined in such a way that the rounding information can be carried along in a meaningful way, at minimal cost. The analysis is carried through for a compact (canonical) encoding using 2's complement representation, supplied with a round-bit. Based on the fixed-point encoding it is shown possible in Section 2.3 to define floating point representations, and a sketch of the implementation of an FPU is presented.

2.1 Definition

When a number x is written as a digit sequence in radix β

$$x = d_n d_{n-1} d_{n-2} \cdots d_0 . d_{-1} d_{-2} \cdots = \sum_{i=-\infty}^{n} d_i \beta^i,$$

we usually use digits d_i that are integers such that $0 \leq d_i \leq \beta - 1$. The conventional binary representation uses the digits 0 and 1, whereas the conventional decimal representation uses the digit set $\{0, \ldots, 9\}$.

However, it is possible to use different digit sets, for example $0 \leq d_i \leq \beta$ (*carry-save*), or $\left\lfloor \frac{-\beta+1}{2} \right\rfloor \leq d_i \leq \left\lceil \frac{\beta-1}{2} \right\rceil$ (*symmetrical signed-digit*). Using a different representation for numbers can yield interesting properties, like faster, fully parallel, additions for carry-save and signed-digit [1, 2, 21, 48, 49]. In this chapter, we present another way of representing numbers, for which rounding to the nearest number at precision p is done by truncating at precision p. Since rounding is done by truncating in this representation, no additional error can occur from double-rounding.

Definition 2.1 (RN-codings)**.** *Let β be an integer greater than or equal to 2. The digit sequence $d_n d_{n-1} d_{n-2} \cdots$ with $-\beta + 1 \leq d_i \leq \beta - 1$ is a radix β RN-representation of x iff*

1. *$x = \sum_{i=-\infty}^{n} d_i \beta^i$ (that is $d_n d_{n-1} d_{n-2} \cdots$ is a radix-β representation of x);*
2. *for any $j \leq n$,*

$$\left| \sum_{i=-\infty}^{j-1} d_i \beta^i \right| \leq \frac{1}{2} \beta^j,$$

that is, if the digit sequence is truncated to the right at any position j, the remaining sequence is always the number (or one of the two numbers in case of a tie) of the form $d_n d_{n-1} d_{n-2} d_{n-3} \ldots d_j$ that is closest to x.

Hence, truncating the RN-coding of a number at any position is equivalent to rounding it to the nearest.

Using the conversion algorithms from conventional digit sets into RN-codings of §2.1.2, it is possible to prove that these RN-codings are correct encodings of numbers, i.e., every number x can be written as a RN-coding in radix β.

2.1.1 Characterization of RN-codings

Theorem 2.2 (Characterization of RN-codings)**.**

- *if $\beta \geq 3$ is odd, then $d_n d_{n-1} d_{n-2} \cdots$ is an RN-coding iff*

$$\forall i, \frac{-\beta+1}{2} \leq d_i \leq \frac{\beta-1}{2},$$

- *if $\beta \geq 2$ is even then $d_n d_{n-1} d_{n-2} \cdots$ is an RN-coding iff*

  1. *all digits have absolute value less than or equal to $\frac{\beta}{2}$;*
  2. *if $|d_i| = \frac{\beta}{2}$, then the first non-zero digit that follows on the right has the opposite sign, that is, the largest $j < i$ such that $d_j \neq 0$ satisfies $d_i \times d_j < 0$.*

Proof. Consider first $\beta \geq 2$ even. If $d_n d_{n-1} d_{n-2} \cdots$ is a RN-representation of x, then for any $i \leq n$,

$$\left|d_i \beta^i\right| = \left| \sum_{k=-\infty}^{i} d_k \beta^k - \sum_{k=-\infty}^{i-1} d_k \beta^k \right| \leq \left| \sum_{k=-\infty}^{i} d_k \beta^k \right| + \left| \sum_{k=-\infty}^{i-1} d_k \beta^k \right| \leq \frac{\beta+1}{2} \beta^i.$$

Hence for any $i \leq n$, we have $|d_i| \leq \frac{\beta+1}{2}$. Since d_i is an integer, we then have $|d_i| \leq \frac{\beta}{2}$. Furthermore, if $d_i = \frac{\beta}{2}$, the largest $j < i$ such that $d_j \neq 0$ satisfies

$$d_j \beta^j = \sum_{k=-\infty}^{i} d_k \beta^k - d_i \beta^i - \sum_{k=-\infty}^{j-1} d_k \beta^k \leq \frac{\beta^j}{2}.$$

Since d_j is an integer, we know that $d_j \leq 0$. For the same reason, if $d_i = -\frac{\beta}{2}$ then the largest $j < i$ such that $d_j \neq 0$ gives $d_j \geq 0$.

Conversely, if the digit sequence $d_n d_{n-1} d_{n-2} \cdots$ satisfies the characterization of Theorem 2.2 for β even, we need to consider two cases:

- If $|d_{j-1}| < \frac{\beta}{2}$, then from $|d_{j-1}| \leq \frac{\beta-2}{2}$ and $|d_i| \leq \frac{\beta}{2}$ for all $i \leq j-1$, we obtain

$$\left| \sum_{i=-\infty}^{j-1} d_i \beta^i \right| \leq \frac{1}{2}\beta^j + \underbrace{\left(\frac{2\beta^{j-1} - \beta^j}{2(\beta-1)} \right)}_{\leq 0 \text{ since } \beta \geq 2} \leq \frac{1}{2}\beta^j.$$

- If $d_{j-1} = \frac{\beta}{2}$ and the largest $k < j-1$ such that $d_k \neq 0$ satisfies $d_k \leq -1$ (the same holds for $d_{j-1} = -\frac{\beta}{2}$), then

$$\left| \sum_{i=-\infty}^{j-1} d_i \beta^i \right| \leq \frac{\beta^j}{2} - \beta^k + \sum_{i=-\infty}^{k-1} \left|d_i \beta^i\right|,$$

  and since for all i, $|d_i| \leq \frac{\beta}{2}$, we finally get

$$\left| \sum_{i=-\infty}^{j-1} d_i \beta^i \right| \leq \frac{1}{2}\beta^j - \beta^k \left(\frac{\beta-2}{2\beta-2} \right) \leq \frac{1}{2}\beta^j,$$

  which concludes the proof for β even.

Now, let us consider $\beta \geq 3$ with β odd. If $d_n d_{n-1} d_{n-2} \cdots$ is a RN-representation, then one can easily see from the inequality in the definition of RN-representations that $|d_j| \leq \frac{\beta+1}{2}$. However, if we assume that for some j, $d_j = \frac{\beta+1}{2}$, then from $\left|\sum_{i=-\infty}^{j} d_i \beta^i\right| \leq \frac{1}{2}\beta^{j+1}$, we deduce that

$$\sum_{i=-\infty}^{j-1} d_i \beta^i \leq -\frac{1}{2}\beta^j,$$

which is a contradiction with the definition of RN-representations. Hence, for all j, $d_j \neq \frac{\beta+1}{2}$. Similarly, $d_j \neq -\frac{\beta+1}{2}$, hence $|d_j| \leq \frac{\beta-1}{2}$.

Conversely, if $|d_i| \leq \frac{\beta-1}{2}$ one can see by summing the terms that

$$\left|\sum_{i=-\infty}^{j-1} d_i \beta^i\right| \leq \frac{1}{2}\beta^j,$$

which concludes the proof for β odd. □

2.1.2 Conversion algorithms

Converting from the usual representation of a number x in radix β into its RN-coding representation in the same radix is basically computing the operation $2x - x$, whose result is of course x.

Example 2.3. Representing π in RN-coding in radix 10 is done as follows:

$$\begin{array}{rl} 2\pi & 6.2831853071795\cdots \\ -\pi & 3.1415926535897\cdots \\ \hline \pi & 3.142\bar{4}\bar{1}33\bar{5}54\bar{4}\bar{1}0\bar{2} \end{array}$$

More formally, converting x from its usual radix β representation $x = x_n x_{n-1} x_{n-2} \cdots x_l$ into its radix β RN-representation $x = d_{n+1} d_n d_{n-1} \cdots d_l$ can be done by first defining the carries $c_i \in \{0, 1\}$ as

$$c_{i+1} = \begin{cases} 1 & \text{if} \quad 2x_i + c_i \geq \beta \\ 0 & \text{if} \quad 2x_i + c_i \leq \beta - 1 \end{cases},$$

and then computing the digit sequence $d_{n+1} d_n d_{n-1} \cdots d_l$ defined by

$$d_i = x_i + c_i - \beta c_{i+1},$$
$$\text{and } d_{n+1} = c_{n+1}.$$

If β is odd, this can lead to arbitrarily large carry-ripple, as we can see when converting the radix 3 numbers $11111111 \to 11111111$ and $11111112 \to 1\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}\bar{1}$.

However, if β is even, the carries c_i can equivalently be defined as

$$c_{i+1} = \begin{cases} 1 & \text{if} \quad x_i \geq \beta/2 \\ 0 & \text{if} \quad x_i < \beta/2 \end{cases}.$$

To prove this, it suffices to remark that

if $x_i \geq \beta/2$, then $2x_i + c_i \geq \beta$,

if $x_i < \beta/2$, then since β is even, $2x_i \leq \beta - 2$ and $2x_i + c_i \leq \beta - 1$.

This shows that, when β is even, carries cannot ripple, and conversion from regular radix β into RN-coding can be done fully in parallel.

2.1.3 Binary encoding

In binary, multiplying by 2 can be done with a left shift. When using our previously defined conversion algorithm on binary 2's complement numbers, this conversion algorithm corresponds to the well-known Booth recoding [6].

Example 2.4. Let $x = 110100110010$ be a sign-extended 2's complement number and write the digits of $2x$ above the digits of x.

$2x$	1	0	1	0	0	1	1	0	0	1	0	0
x	1	1	0	1	0	0	1	1	0	0	1	0
RN-repr. x		$\bar{1}$	1	$\bar{1}$	0	1	0	$\bar{1}$	0	1	$\bar{1}$	0

In any column the two upper-most bits provide the encoding defined above of the signed-digit below in the column. Since the signed digit in the leftmost position will always be 0, there is no need to include the most significant position otherwise found in the two top rows.

Rounding this value by truncating off the two least significant digits we obtain:

$\mathrm{RZ}_9(2x)$	1	0	1	0	0	1	1	0	0	1
$\mathrm{RZ}_9(x)$	1	1	0	1	0	0	1	1	0	0
RN-repr. $\mathrm{RN}_9(x)$		$\bar{1}$	1	$\bar{1}$	0	1	0	$\bar{1}$	0	1

The bit of value 1 in the upper rightmost corner (in red) acts as a round bit, assuring a round-up in cases there is a tie-situation as here.

From this example, one can imagine two possible ways of encoding the RN-coding in binary:

- encoding each signed digit as two bits.

$$\begin{aligned} -1 &\sim (0,1) \\ 0 &\sim (0,0) \text{ or } (1,1) \\ 1 &\sim (1,0), \end{aligned} \tag{2.1}$$

  where the value of the digit is the difference between the first and the second component,

- using a more compact form, noticing that in the example above, the first row is simply the second row, shifted to the left, except for the *round-bit*. This more compact form is what we call *Binary canonical RN-coding*.

Definition 2.5 (Binary canonical RN-coding)**.**
Let the number x be given in 2's complement representation as the bit string $b_n \cdots b_{\ell+1} b_\ell$, such that $x = -b_n 2^n + \sum_{i=\ell}^{n-1} b_i 2^i$. Then the binary canonical encoding *of the RN-representation of x is defined as the pair*

$$x \sim (b_n b_{n-1} \cdots b_{\ell+1} b_\ell, r) \text{ where the round-bit is } r = 0$$

and after truncation at position k, for $n \geq k > \ell$

$$\mathrm{RN}_{n-k}(x) \sim (b_n b_{n-1} \cdots b_{k+1} b_k, r) \text{ with round-bit } r = b_{k-1}.$$

Example 2.6. The binary canonical RN-coding of the previous truncated example correspond to the bits set in red:

$\mathrm{RZ}_9(2x)$	1	0	1	0	0	1	1	0	0	1
$\mathrm{RZ}_9(x)$	1	1	0	1	0	0	1	1	0	0
RN-repr. $\mathrm{RN}_9(x)$		$\bar{1}$	1	$\bar{1}$	0	1	0	$\bar{1}$	0	1

Definition 2.7 (Value of a binary canonical RN-coding)**.**
A fixed-point number $x = \sum_{i=l}^{n} b_i 2^i$ has two binary canonical RN-codings, $(x, 0)$ and $(x - u, 1)$, where u is the unit in the last place : $u = 2^l$. We will note $\mathcal{V}(x, r_x)$ the value of the number represented by the RN-coding x, r_x, and we have

$$\mathcal{V}(x, r_x) = x + r_x u.$$

It is important to notice that although from a "value perspective" the representation is redundant ($\mathcal{V}(a, 1) = \mathcal{V}(a+u, 0)$), it is not so when considering the signed-digit representation. In this interpretation the sign of the least significant digit carries information about the sign of the part which possibly has been rounded away. Hence, the two representations that have the same value might round to different numbers, which have different values. For example, we have $\mathcal{V}(0.1101001111, 1) = \mathcal{V}(0.1101010000, 0)$, but

$$\begin{gathered}\mathcal{V}(\mathrm{RN}_5(0.1101001111, 1)) \neq \mathcal{V}(\mathrm{RN}_5(0.1101010000, 0)) \\ \mathcal{V}(0.11010, 0) \neq \mathcal{V}(0.11010, 1) \\ 0.11010 \neq 0.11011\end{gathered}$$

Lemma 2.8. *Provided that a RN-represented number with canonical encoding (a, r_a) is non-zero, then $r_a = 1$ implies that the least significant non-zero digit in its signed-digit representation is 1 (the number was possibly rounded up), and $r_a = 0$ implies it is -1 (the number was possibly rounded down).*

Proof. The result is easily seen when listing the trailing bits of the 2's complement representation of $2a + r_a$ (with r_a in red) above those of a together with the signed-digit representation:

$$\begin{array}{cc} \ldots\ 0\ 1\ 1\ \ldots\ 1 & \ldots\ 1\ 0\ 0\ \ldots\ 0 \\ \ldots\ .\ 0\ 1\ \ldots\ 1 & \ldots\ .\ 1\ 0\ \ldots\ 0 \\ \hline \ldots\ .\ 1\ 0\ \ldots\ 0 & \ldots\ .\ \bar{1}\ 0\ \ldots\ 0 \end{array}$$

□

If (x, r_x) is the binary canonical encoding of $X = \mathcal{V}(x, r_x) = x + r_x u$ then it follows that

$$-X = -x - r_x u = \bar{x} + u - r_x u = \bar{x} + (1 - r_x)u = \bar{x} + \bar{r}_x u,$$

which can also be seen directly from the encoding of the negated signed digit representation.

Lemma 2.9. *If (x, r_x) is the canonical RN-encoding of a value X, then $(\bar{x}, \bar{r}_x)$ is the canonical RN-encoding of $-X$, where $\bar{x}$ is the 1's complement of x. Hence negation of a canonically encoded value is a constant time operation.*

2.1.4 Interval interpretation

We may interpret the representation (a, r_a) as an interval $\mathcal{I}(a, r_a)$ of length $u/2$:

$$\mathcal{I}(a,1) = \left[a + \tfrac{u}{2}\ ;\ a + u\right],\ \mathcal{I}(a+u,0) = \left[a+u\ ;\ a + \tfrac{3u}{2}\right]$$

or

$$\mathcal{I}(a, r_a) = \left[a + r_a\tfrac{u}{2}\ ;\ a + (1 + r_a)\tfrac{u}{2}\right], \tag{2.2}$$

when interpreting them as intervals according to what may have been thrown away when rounding by truncation.

Hence even though $(a, 1)$ and $(a + u, 0)$ represent the same value $a + u$, as intervals they are essentially disjoint, except for sharing a single point. In general we may express the interval interpretation as pictured in Fig. 2.1.

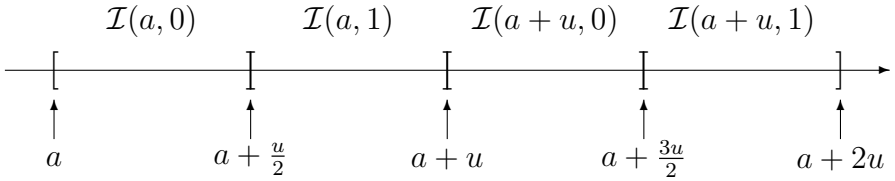


Figure 2.1: Binary Canonical RN-representations as Intervals

We do not intend to define an interval arithmetic, but only require that the interval representation of the result of an arithmetic operation $\odot$ satisfies [1]

$$\mathcal{I}(A \odot B) \subseteq \mathcal{I}(A) \odot \mathcal{I}(B) = \{a \odot b | a \in A, b \in B\}.$$

In Section 2.2, we find several ways of defining arithmetic operations such as addition, and this constraint on intervals permits to ensure the uniqueness of the result of the operations.

2.2 Computing with fixed-point RN-coding

We will first consider fixed-point representations for some fixed value of u of the unit in the last place. We want to operate directly on the components of the encoding (a, r_a), not on the signed-digit representation, but we will not discuss overflow problems, as we assume that we have enough bits to represent the result in canonically encoded representation.

2.2.1 Addition of RN-Represented Values

Employing the value interpretation of encoded operands (a, r_a) and (b, r_b) we have for addition:

$$\begin{array}{rcl} \mathcal{V}(a, r_a) & = & a + r_a u \\ + \mathcal{V}(b, r_b) & = & b + r_b u \\ \hline \mathcal{V}(a, r_a) + \mathcal{V}(b, r_b) & = & a + b + (r_a + r_b)u \end{array}$$

The resulting value has two possible representations, depending on the choice of the rounding bit of the result. To determine what the rounding bit of the result should be,

[1] Note that this is the reverse inclusion of that required for ordinary interval arithmetic

	$\mathcal{I}(\mathcal{V}(a,r_a)+\mathcal{V}(b,r_b))$	$\mathcal{I}(a,r_a)+\mathcal{I}(b,r_b)$
$\mathbf{r_a = r_b = 0}$	$\mathcal{I}(a+b-u,1) = \left[a+b-\frac{u}{2}\ ;\ a+b\right]$	$\not\subseteq\ [a+b\ ;\ a+b+u]$
	$\mathcal{I}(a+b,0) = \left[a+b\ ;\ a+b+\frac{u}{2}\right]$	$\subseteq\ [a+b\ ;\ a+b+u]$
$\mathbf{r_a \oplus r_b = 1}$	$\mathcal{I}(a+b,1) = \left[a+b+\frac{u}{2}\ ;\ a+b+u\right]$ $\mathcal{I}(a+b+u,0) = \left[a+b+u\ ;\ a+b+\frac{3u}{2}\right]$	$\subseteq\ \left[a+b+\frac{u}{2}\ ;\ a+b+\frac{3u}{2}\right]$
$\mathbf{r_a = r_b = 1}$	$\mathcal{I}(a+b+u,1) = \left[a+b+\frac{3u}{2}\ ;\ a+b+2u\right]$	$\subseteq\ [a+b+u\ ;\ a+b+2u]$
	$\mathcal{I}(a+b+2u,0) = \left[a+b+2u\ ;\ a+b+\frac{5u}{2}\right]$	$\not\subseteq\ [a+b+u\ ;\ a+b+2u]$

Table 2.1: Interpretations of additions as intervals

we consider interval interpretations of the two possible representations of the result in Table 2.1, depending on the rounding bits of the operands. In order keep the property that $\mathcal{I}((a,r_a)\oplus(b,r_b)) \subseteq \mathcal{I}(a,r_a)+\mathcal{I}(b,r_b)$, the rounding bit of the result should be 0 (resp. 1) when the rounding bits of the operands are both 0 (resp. 1).

In order to keep the addition symmetric and to have $(a,r_a)\oplus(0,0) = (a,r_a)$ we define addition of RN encoded numbers as follows.

Definition 2.10 (Addition)**.** *If u is the unit in the last place of the operands, let us define addition as:*

$$(a,r_a)\oplus(b,r_b) = ((a+b+(r_a\wedge r_b)u), r_a\vee r_b),$$

where $r_a \wedge r_b$ may be used as carry-in to the 2's complement addition.

Example 2.11. Let us take two examples adding two numbers that were previously rounded to the nearest integer.

	Addition not rounded	Addition on rounded canonical representations
a_1	01011.1110	$(01011, 1)$
b_1	01001.1101	$(01001, 1)$
$a_1 + b_1$	010101.1011	$(010101, 1)$
a_2	01011.1010	$(01011, 1)$
b_2	01001.1001	$(01001, 1)$
$a_2 + b_2$	010101.0011	$(010101, 1)$

Using the definition above, $\mathrm{RN}_0(a_1 + b_1) = \mathrm{RN}_0(a_1) + \mathrm{RN}_0(b_1)$ holds in the first case. Obviously, since some information may be lost during rounding, there are cases like in the second example where $\mathrm{RN}_0(a_2+b_2) \neq \mathrm{RN}_0(a_2)+\mathrm{RN}_0(b_2)$. Note that due to the information loss, $a_2 + b_2$ is not in $\mathcal{I}((a_2, r_{a_2}) + (b_2, r_{b_2}))$.

Recalling that $-(x, r_x) = (\bar{x}, \bar{r_x})$, we observe that using Definition 2.10 for subtraction yields $(x, r_x) \ominus (x, r_x) = (-u, 1)$, with $\mathcal{V}(-u, 1) = 0$. It is possible alternatively to define addition on RN-encoded numbers as

$$(a, r_a) \oplus_2 (b, r_b) = ((a + b + (r_a \vee r_b)u), r_a \wedge r_b).$$

Using this definition, one has

$$(x, r_x) \ominus_2 (x, r_x) = (0, 0),$$

but then the neutral element for addition is $(-u, 1)$ instead of $(0, 0)$, i.e.,

$$(x, r_x) \oplus_2 (-u, 1) = (x, r_x).$$

2.2.2 Multiplying RN-Represented Values

By definition we have for the value of the product

$$\begin{array}{rcl} \mathcal{V}(a, r_a) & = & a + r_a u \\ \mathcal{V}(b, r_b) & = & b + r_b u \\ \hline \mathcal{V}(a, r_a)\mathcal{V}(b, r_b) & = & ab + (ar_b + br_a)u + r_a r_b u^2, \end{array}$$

noting that the unit of the result is u^2, assuming that $u \leq 1$ and a and b are greater than u. Since negation of canonical (2's complement) RN-encoded values can be obtained by constant-time bit inversion (Lemma 2.9), multiplication can be realized by multiplication of the absolute values of the operands, the result being supplied with the correct sign by a conditional inversion.

When multiplying positive values, the resulting value has two possible representations, depending on the choice of the rounding bit of the result. To determine what the rounding bit of the result should be, we consider interval interpretations of the two possible representations of the result in Table 2.2, depending on the rounding bits of the operands. In order keep the property that

$$\mathcal{I}((a, r_a) \otimes (b, r_b)) \subseteq \mathcal{I}(a, r_a) \times \mathcal{I}(b, r_b),$$

the rounding bit of the result should be 0 (resp. 1) when the rounding bits of the operands are both 0 (resp. 1).

In order to keep the multiplication symmetric and to have $(a, r_a) \otimes (1, 0) = (a, r_a)$ we define multiplication of RN encoded numbers as follows.

Definition 2.12 (Multiplication)**.** *If u is the unit in the last place, with $u \leq 1$, we define for non-negative operands:*

$$(a, r_a) \otimes (b, r_b) = (ab + u(ar_b + br_a), r_a r_b)\,,$$

and for general operands by appropriate sign inversions of the operands and result. If $u < 1$ the unit is $u^2 < u$ and the result may often have to be rounded to unit u, which can be done by truncation.

The product can be returned as (p, r_p) with

$$p = ab + u(ar_b + br_a) = a(b + r_b u) + br_a u,$$

where the terms of $br_a u$ may be added into the array of partial products.

For a 5-bit integer example let $(a, r_a) = (a_4a_3a_2a_1a_0, r_a)$ and $(b, r_b) = (b_4b_3b_2b_1b_0, r_b)$, or in signed-digit $b = d_4d_3d_2d_1d_0$, $d_i \in \{-1, 0, 1\}$, we note that $a_4 = b_4 = 0$ since $a \geq 0$ and $b \geq 0$. It is then possible to add the terms of $br_a u$ (shown framed) into the array of partial products[2]:

					a_3	a_2	a_1	a_0	
					a_3d_0	a_2d_0	a_1d_0	a_0d_0	d_0
				a_3d_1	a_2d_1	a_1d_1	a_0d_1	$\boxed{b_0r_a}$	d_1
			a_3d_2	a_2d_2	a_1d_2	a_0d_2	$\boxed{b_1r_a}$		d_2
		a_3d_3	a_2d_3	a_1d_3	a_0d_3	$\boxed{b_2r_a}$			d_3
	a_3d_4	a_2d_4	a_1d_4	a_0d_4	$\boxed{b_3r_a}$				d_4
p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0	

Thus the product is (p, r_p) with $p = ab + u(ar_b + br_a) = a(b + r_b u) + br_a u$ and $r_p = r_a r_b$.

Multiplication of RN-represented values can therefore be implemented on their canonical encodings at about the same cost as ordinary 2's complement multiplication. The result may have to be rounded, which by truncation in constant time will define the rounded product as some (p', r'_p). Note that when recoding the multiplier into a higher radix like 4 and 8, similar kinds of modification may be applied.

[2]We do not show the possible rewriting of negative partial products, requiring an additional row.

	$\mathcal{I}(\mathcal{V}(a,r_a)\times\mathcal{V}(b,r_b))$	$\mathcal{I}(a,r_a)\times\mathcal{I}(b,r_b)$
$\mathbf{r_a=r_b=0}$	$\mathcal{I}(ab-u^2,1)=\left[ab-\frac{u^2}{2}\ ;\ ab\right]$	$\not\subseteq\left[ab\ ;\ ab+\frac{(a+b)u}{2}+\frac{u^2}{4}\right]$
	$\mathcal{I}(ab,0)=\left[ab\ ;\ ab+\frac{u^2}{2}\right]$	$\subseteq\left[ab\ ;\ ab+\frac{(a+b)u}{2}+\frac{u^2}{4}\right]$
$\mathbf{r_a=1,r_b=0}$	$\mathcal{I}(ab+bu-u^2,1)=\left[ab+bu-\frac{u^2}{2}\ ;\ ab+bu\right]$ $\mathcal{I}(ab+bu,0)=\left[ab+bu\ ;\ ab+bu+\frac{u^2}{2}\right]$	$\subseteq\left[ab+\frac{bu}{2}\ ;\ ab+(\frac{a}{2}+b)u+\frac{u^2}{2}\right]$
$\mathbf{r_a=0,r_b=1}$	$\mathcal{I}(ab+au-u^2,1)=\left[ab+au-\frac{u^2}{2}\ ;\ ab+au\right]$ $\mathcal{I}(ab+au,0)=\left[ab+au\ ;\ ab+au+\frac{u^2}{2}\right]$	$\subseteq\left[ab+\frac{au}{2}\ ;\ ab+(a+\frac{b}{2})u+\frac{u^2}{2}\right]$
$\mathbf{r_a=r_b=1}$	$\mathcal{I}(ab+(a+b)u,1)$	$\subseteq\left[ab+\frac{(a+b)u}{2}+\frac{u^2}{4}\ ;\ ab+(a+b)u+u^2\right]$
	$\mathcal{I}(ab+(a+b)u+u^2,0)$	$\not\subseteq\left[ab+\frac{(a+b)u}{2}+\frac{u^2}{4}\ ;\ ab+(a+b)u+u^2\right]$

Table 2.2: Interpretations as intervals of multiplications of positive operands

Example 2.13. For $a = (01011, 1)$ and $b = (01001, 1)$ (with $\mathcal{V}(b) = 1\bar{1}010$), the array of partial products is:

					1	0	1	1	
					0	0	0	0	0
				1	0	1	1	$\boxed{1}$	1
			0	0	0	0	$\boxed{0}$		0
		-1	0	-1	-1	$\boxed{0}$			-1
	1	0	1	1	$\boxed{1}$				1
0	0	1	1	1	0	1	1	1	

Hence $(01011, 1) \otimes (01001, 1) = (001110111, 1)$, where we note that $(001110111, 1)$ corresponds to the interval $[01110111.1\ ;\ 01111000.0]$, which is a subset of the interval

$$[01011.1 \times 01001.1\ ; 01100 \times 01010] = [01101101.01\ ; 01111000.00].$$

2.2.3 Dividing RN-Represented Values

As for multiplication, we assume that negative operands have been sign-inverted, and that the signs are treated separately. Employing our interval interpretation (2.2), we must require the result of dividing (x, r_x) by (y, r_y) to be in the interval:

$$\left[\frac{x + r_x\frac{u}{2}}{y + (1 + r_y)\frac{u}{2}}\ ;\ \frac{x + (1 + r_x)\frac{u}{2}}{y + r_y\frac{u}{2}}\right],$$

where it is easily seen that the rational value[3]

$$q = \frac{x + r_x\frac{u}{2}}{y + r_y\frac{u}{2}}$$

belongs to that interval. Note that the dividend and divisor to obtain the quotient q are then constructed simply by appending the round bits to the 2's complement parts, i.e., simply using the "extended" bit-strings as operands. To determine the accuracy needed in an approximate quotient $q' = q + \varepsilon$ consider the requirement

$$\frac{x + r_x\frac{u}{2}}{y + (1 + r_y)\frac{u}{2}} < q + \varepsilon < \frac{x + (1 + r_x)\frac{u}{2}}{y + r_y\frac{u}{2}}. \tag{2.3}$$

Generally division algorithms require that the operands are scaled, hence assume that the operands satisfy $1 \le x < 2$ and $1 \le y < 2$, implying $\frac{1}{2} < q < 2$. Furthermore assume that x and y have p fractional digits, so $u = 2^{-p}$. To find sufficient bounds on the error ε in (2.3) consider first for $\varepsilon \ge 0$ the right bound. Here we must require

$$(x + r_x\tfrac{u}{2}) + \varepsilon(y + r_y\tfrac{u}{2}) < x + (1 + r_x)\tfrac{u}{2} \quad \text{or} \quad \varepsilon(y + r_y\tfrac{u}{2}) < \tfrac{u}{2},$$

which is satisfied for $\varepsilon < \frac{u}{4}$, since $y + r_y\frac{u}{2} < 2$. For the other bound (for negative ε) we must require

$$\frac{x + r_x\frac{u}{2}}{y + (1 + r_y)\frac{u}{2}} < \frac{x + r_x\frac{u}{2}}{y + r_y\frac{u}{2}} + \varepsilon$$

[3] We might also have chosen to evaluate the quotient $\frac{x + r_x u}{y + r_y u}$. However dividing (x, r_x) by the neutral element $(1, 0)$ would then yield the result $(x + r_x u, 0)$, whereas with the chosen quotient the result becomes (x, r_x). The difference between these two expressions evaluated to some precision p is at most one ulp(p).

or

$$-\varepsilon(y + (1 + r_y)\tfrac{u}{2}) < (x + r_x\tfrac{u}{2})\tfrac{u}{2},$$

which is satisfied for $-\varepsilon < \frac{u}{4}$, since $x \geq 1$ and $y + u \leq 2$.

Hence $|\varepsilon| < \frac{u}{4}$ assures that (2.3) is satisfied, and any standard division algorithm may be used to develop a binary approximation to q with $p+2$ fractional bits, $x = q'y + r$ with $|r| < y2^{-p-2}$. Note that since q may be less than 1, a left shift may be required to deliver a $p+1$ signed-digit result, the same number of digits as in the operands. Hence a bit of weight 2^{-p-1} will always be available to determine the round bit.

The sign of the remainder determines the sign of the tail beyond the bits determined. Recall from Lemma 2.8 that when the round bit is 1, the error is assumed non-positive, and non-negative when the round bit is 0. If this is not the case then the resulting round bit must be inverted, hence rounding is also here a constant time operation.

2.3 Computing with floating-point RN-coding

For an implementation of a binary floating point arithmetic unit (FPU) it is necessary to define an encoding of an operand $(2^e m, r_m)$, based on the canonical encoding of the significand part (say m encoded in $p+1$ bits, 2's complement), supplied with the round bit r_m and the exponent e in some biased binary encoding. It is then natural to pack the components into a computer word (32, 64 or 128 bits), employing the same principles as used in the IEEE-754 standard [27]. For normal values it is also possible here to use a "hidden bit," noting that the 2's complement encoding of the normalized significand will have complementary first and second bits. Thus representing the leading bit as a (separate) sign bit, the next bit (of weight 2^0) need not be represented and can be used as "hidden-bit." Hence let f_m be the fractional part of m, assumed to be normalized such that $1 \leq |m| < 2$, and let s_m be the sign-bit of m. The "hidden bit" is then the complement $\bar{s}_m$ of the sign-bit. The components can then be allocated in the fields of a word as:

with the round bit in immediate continuation of the significand part. The exponent e can be represented in biased form as in the IEEE-754 standard. The number of bits allocated to the individual fields may be chosen as in the different IEEE-754 formats, of course with the combined f_m, r_m together occupying the fraction field of those formats. The value of a floating point number encoded this way can then be expressed as

$$2^{e-bias}\left([s_m\bar{s}_m.f_1f_2\cdots f_{p-1}]_{2c} + r_m 2^{-p-1}\right),$$

where $f_1, f_2, \cdots, f_{p-1}$ are the (fractional) bits of f_m.

Subnormal and exceptional values may be encoded as in the IEEE-754 standard, noting that negative subnormals have leading ones. Observe that the representation is sign-symmetric and that negation is obtained by inverting the bits of the significand.

We shall now sketch how the fundamental operations may be implemented on such floating point RN-representations, not going into details on overflow, underflow and exceptional values as these situations can be treated exactly as known for the standard binary IEEE-754 representation. [27]

2.3.1 Multiplication and Division

Since the exponents are handled separately, forming the product or quotient of the significands is precisely as described previously for fixed point representations: sign-inverting negative operands by bit-wise inversion, forming the double-length product, normalizing and rounding it, and supplying it with the proper sign by possibly negating the result.

2.3.2 Addition

Before addition or subtraction there is in general a need of alignment of the two operand significands, according to the difference of their exponents (too large a difference is treated as a special case). The operand having the larger exponent must be left-shifted, with appropriate digit values appended at the least significant end, to overlap with the significand of the smaller operand. In effective subtractions, after cancellation of leading digits it may be necessary to left normalize, so we also need here to consider what to append at the right, recalling that we want to operate on the significands encoded in 2's complement.

Thinking of the value as represented in binary signed-digit, obviously zeroes have to be shifted in. In our encoding, say for a positive result (d, r_d) we may have a 2's complement bit pattern:

$$d \sim 0\ 0\ \cdots\ 0\ 1\ b_k \cdots b_{p-1} \text{ and round bit } r_d$$

to be left normalized. If we were encoding the number with two bits for each signed digit, as in (2.1), the least significant signed digit would be encoded as (r_d, b_{p-1}). Zero-valued digits to be shifted in may then naturally be encoded as (r_d, r_d), as confirmed from applying the addition rule for obtaining $2 \times (x, r_x)$ by $(x, r_x) + (x, r_x) = (2x + r_x u, r_x)$.

It then follows that shifting in bits of value r_d will precisely achieve the effect of shifting in zeroes in the signed-digit interpretation:

$$2^k d \sim 0\ 1\ b_k \cdots b_{p-1} r_d \cdots r_d \text{ with round bit } r_d.$$

Subtraction, the "near case"

Addition is traditionally now handled in an FPU as two cases [16], where the "near case" is dealing with effective subtraction of operands whose exponents differ by no more than one. Here, a significant cancellation of leading digits may occur, and thus a variable amount of normalization shifts are required. This left shifting is handled by shifting in copies of the round-bit. Figure 2.2 shows a possible pipelined implementation of this case, where `lzd`(d) is a log-time algorithm for "leading zeroes determination" of the difference (see e.g., [33]) to determine the necessary normalization shift amount. This determination is based on a redundant representation of the difference (obtained in constant time by pairing the aligned operands), taking place in parallel with the subtraction (conversion from redundant to non-redundant representation). Normalization can take place on the non-redundant difference, without need for sign inversion, as in the case of the sign-magnitude representation used in the IEEE-754 encoding. For simplicity in the figure we assume that m_a, m_b and m_r are the 2's complement operands, respectively the result, including their appended round-bits.

Addition, the "far case"

The remaining case dealt with are the situations where the result of adding or subtracting the aligned significands at most requires normalization by a single right or left shift. Since

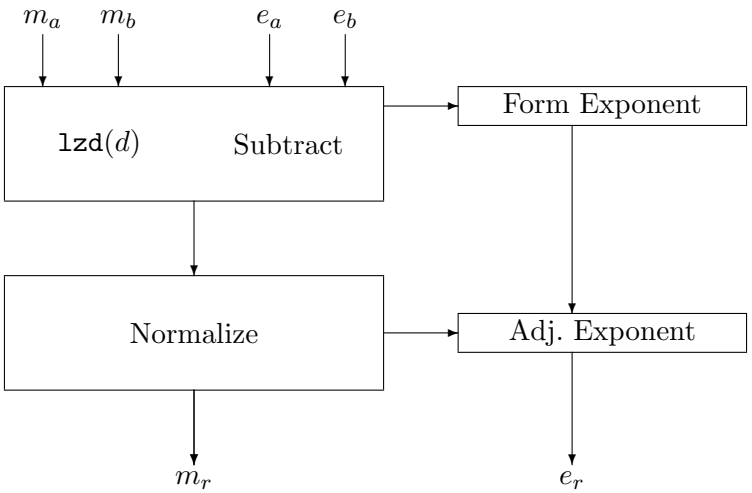


Figure 2.2: Near Path, effective subtraction when $|e_a - e_b| \leq 1$

negation is a constant time operation we may assume that an effective addition is to be performed of the larger operand (appended with copies of the round-bit) and the sign extended smaller operand. Rounding can then be performed as usual by truncation, noting that here there are only two log-time operations: the variable amount of alignment shifts and addition/subtraction. Compared with standard number representations we avoid the "expensive" determination of a sticky bit and rounding incrementation. Figure 2.3 shows a possible two-stage pipeline implementation.

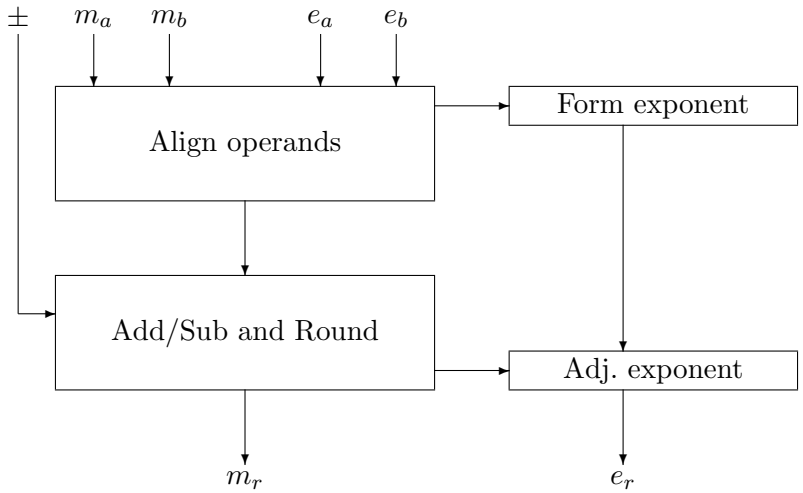


Figure 2.3: Far Path, add or subtract when $|e_a - e_b| \geq 2$

Note that in the case where the exponent difference exceeds the number of operand bits is not necessary to form the exact sum. It is sufficient to deliver the larger operand as the result, but following Lemma 2.8 possibly inverting the round-bit of the result so that it reflects the sign of the smaller (discarded) operand.

2.3.3 Discussion of the Floating Point RN-representation

As seen above it is possible to define binary floating point representations, where the significand is encoded in the canonical 2's complement encoding with the round-bit appended at the end. An FPU implementation of the basic arithmetic operations is feasible in about

the same complexity as one based on the IEEE-754 standard for binary floating point. But since the round-to-nearest functionality is achieved at less hardware complexity, the arithmetic operations will generally be faster, by avoiding the usual log-time "sticky-bit" determination and rounding incrementation. Benefits are obtained through faster rounding to nearest, also note that the domain of representable values is sign-symmetric. The directed roundings can also be realized at minimal cost; however, they require the calculation of a "sticky bit," as also needed for the directed roundings of the IEEE-754 representation, but no rounding incrementation is needed here, as shown in the next result.

Theorem 2.14. *Let a number after truncation of tail t have encoding (a, r_a) with sign-bit s_a, then the directed roundings can be realized by changing the resulting round-bit if the truncated tail t (the "sticky-bit") is non-zero as follows:*

$$\begin{aligned} \text{RU}: &\quad r_a := 1 \\ \text{RD}: &\quad r_a := 0 \\ \text{RZ}: &\quad r_a := s_a \\ \text{RA}: &\quad r_a := \bar{s}_a, \end{aligned}$$

Proof. Consider the case of RU when $r_a = 0$. By Lemma 2.8 the least significant non-zero signed-digit of the truncated (a, r_a) is -1, and if $t \neq 0$ the value was effectively rounded down, thus r_a should be changed to $r_a = 1$, whereas it should not be changed when $r_a = 1$. The other cases follow similarly. □

2.4 Conclusion

We have analyzed a general class of number representations for which truncation of a digit string yields the effect of rounding to nearest.

Concentrating on binary RN-represented operands, we have shown how a simple encoding, based on the ordinary 2's complement representation, allows trivial (constant time) conversion from 2's complement representation to the binary RN-representation. A simple parallel prefix (log time) algorithm is needed for conversion the other way. We have demonstrated how operands in this particular canonical encoding can be used at hardly any penalty in many standard calculations, e.g., addition and multiplication, with negation even being a constant time operation, which often simplifies the implementation of arithmetic algorithms.

Similarly to what have been done for division in §2.2.3, function evaluations like squaring, square root and even the evaluation of "well behaved" transcendental functions may be defined and implemented, just considering canonical RN-represented operands as 2's complement values with a "carry-in" not yet absorbed, possibly using interval interpretation to define the resulting round bit.

The particular feature of the RN-representation, that rounding-to-nearest is obtained by truncation, implies that repeated roundings ending in some precision yields the same result, as if a single rounding to that precision was performed. To deal with double-rounding errors, it was previously proposed [39] to attach some state information (2 bits) to a rounded result, allowing subsequent roundings to be performed in such a way, that multiple roundings yields the same result as a single rounding to the same precision. It was shown that this property holds for any specific IEEE-754 [27] rounding mode, including

in particular for the round-to-nearest-even mode. But these roundings may still require log-time incrementations, which are avoided with the proposed RN-representation.

The fixed point encoding immediately allows for the definition of corresponding floating point representations, which in a comparable hardware FPU implementation will be simpler and faster than an equivalent IEEE standard conforming implementation.

Thus in applications where conformance to the IEEE-754 standard is not required, it is possible to avoid the penalty of intermediate log-time roundings by employing the RN-representation. Signal processing may be an application area where specialized hardware (ASIC or FPGA) is often used anyway, and the RN-representation can provide faster arithmetic with round to nearest operations at reduced area and delay.

Chapter 3

Breakpoints for some algebraic functions

As we reminded in the Introduction (Section 1.3), in a floating-point system that follows the IEEE 754-1985 standard for radix-2 floating-point arithmetic [25], the user can choose an active rounding mode, also called *rounding-direction attribute* in the newly revised IEEE 754-2008 standard [27]:

- rounding toward $-\infty$ ($\mathrm{RD}_p(z)$),
- rounding toward $+\infty$ ($\mathrm{RU}_p(z)$),
- rounding toward 0 ($\mathrm{RZ}_p(z)$),
- and rounding to nearest ($\mathrm{RN}_p(z)$), which is the default rounding mode.

We are interested here in facilitating the delivery of correctly-rounded results for various simple algebraic functions that are frequently used in numerical analysis or signal processing. Given a function $f : \mathbb{R}^d \to \mathbb{R}$ and a floating-point number x, the problem of computing $\mathrm{RN}_p(f(x))$ is closely related to the knowledge of the midpoints of the function f, as we presented in §1.3.3. Let us remind that we call *breakpoints* the discontinuities of the rounding functions: More specifically, the *midpoints* are the discontinuities of the RN() rounding function, and $\mathbb{M}_{\beta,p}$ denotes the set of all midpoints, namely:

$$\mathbb{M}_{\beta,p} = \left\{ \pm \left(Z + \frac{1}{2} \right) \cdot \beta^{e_z - p + 1} \;\middle|\; Z \in \mathbb{N},\ \beta^{p-1} \leq Z < \beta^p \text{ and } e_z \in \mathbb{Z} \right\}.$$

The *exact points* are the discontinuities of the other rounding functions, which basically correspond to the set of floating point numbers $\mathbb{F}_{\beta,p}$.

In this chapter, we present results on the existence of midpoints and exact points for some algebraic functions: beyond division, inversion, and square root, we studied the reciprocal square root $1/\sqrt{y}$, the 2D Euclidean norm $\sqrt{x^2+y^2}$ and its reciprocal $1/\sqrt{x^2+y^2}$, and the 2D-normalization function $x/\sqrt{x^2+y^2}$. A part of the results presented on division and square root have been known for some time in binary arithmetic; see for instance the work by Markstein [43], as well as studies by Iordache and Matula [28] and Parks [50, 51]. Let us also mention the work by Lauter and Lefèvre [38] on the function x^y, which thus covers integer powers. We present these results for completeness, and we extend some of them to other radices, in particular to radix 10.

Table 3.1 summarizes the results presented in the chapter. In this table, "many" indicates that the techniques we used did not allow us to find a simple characterization of the midpoints or of the exact points of the function, and that an exhaustive enumeration was impractical because of the too large number of cases to consider. Most of the results displayed here for $\beta = 2$ are in fact obtained in a more general setting, namely for $\beta = 2^q$ with $q \in \mathbb{N}_{>0}$.

Table 3.1: Summary of the results given in this chapter.

	Midpoints		Exact points	
Function	Radix 2	Radix 10	Radix 2	Radix 10
$\sqrt{y}$	none	none	many	many
$1/\sqrt{y}$	none	Theorem 3.5	$y = 2^{2k}$	Theorem 3.8
x^k for $k \in \mathbb{N}_{>0}$	Theorem 3.11	Theorem 3.11	Theorem 3.11	Theorem 3.11
$x/\|\|y\|\|_2$	none	many	many	many
x/y	none	many	many	many
$1/y$	none	Theorem 3.15	$y = \pm 2^k$	Theorem 3.16
$1/\sqrt{x^2+y^2}$	none	Theorem 3.20	$\{x, y\} = \{0, \pm 2^k\}$	Theorem 3.22
$x/\sqrt{x^2+y^2}$	none	none	$x = 0$ or $y = 0$	many
$\sqrt{x^2+y^2}$	many	many	many	many

We start with extensions to radices 2^q and 10 of classical, radix-2 results for square roots (Section 3.1), reciprocal square roots (Section 3.2), and positive integer powers (Section 3.3). In Section 3.4 we move to the function that maps a real x and a d-dimensional real vector $y = [y_k]_{1 \le k \le d}$ to $x/\|y\|_2$. Here $\|\cdot\|_2$ denotes the Euclidean norm of vectors:

$$\|y\|_2 = \sqrt{y_1^2 + \cdots + y_d^2}.$$

The function $x/\|y\|_2$ is interesting for it covers several important special cases, each of them being detailed in a subsequent section: for $d = 1$, division and reciprocal (Sections 3.5 and 3.6); for $d = 2$, reciprocal two-dimensional Euclidean norm $1/\sqrt{x^2+y^2}$ and normalization of two-dimensional vectors $x/\sqrt{x^2+y^2}$ (Sections 3.7 and 3.8). We comment on the two-dimensional Euclidean norm in Section 3.9. Finally, we discuss the issue of breakpoints located in the range of subnormal floating-point numbers in Section 3.10.

3.1 Square-root

3.1.1 Midpoints for square root

The following Theorem 3.1 can be viewed as a consequence of a result of Markstein [43, Theorem 9.4]. It says that the square root function has no midpoints, whatever the radix β is. A detailed proof is given here for completeness.

Theorem 3.1 (Markstein [43]). *Let $y \in \mathbb{F}_{\beta,p}$ be positive. Then $\sqrt{y} \notin \mathbb{M}_{\beta,p}$.*

Proof. Let $z = \sqrt{y}$ and assume that z is in $\mathbb{M}_{\beta,p}$. Then there exist some integers Z and e_z such that $z = (Z + 1/2) \cdot \beta^{e_z - p + 1}$ and $\beta^{p-1} \leq Z < \beta^p$. Using $y = z^2$ and $y = Y \cdot \beta^{e_y - p + 1}$, we deduce that

$$4Y \cdot \beta^{e_y - 2e_z + p - 1} = (2Z + 1)^2. \tag{3.1}$$

Now, one may check that $e_z = \lfloor e_y/2 \rfloor$, so that

$$e_y - 2e_z = e_y \bmod 2, \tag{3.2}$$

which is non-negative. Thus, for $p \geq 1$, the left-hand side of (3.1) is an even integer. This contradicts the fact that the right-hand side is an odd integer. □

3.1.2 Exact points for square root

We saw in the previous subsection that the square root function has no midpoints. The situation for exact points is just opposite: for a given input exponent, the number N of floating-point numbers having this exponent and whose square root is also a floating-point number is huge. This number grows essentially like $\beta^{p/2}$. In this section, we make this claim precise for $\beta = 2^q$ ($q \in \mathbb{N}_{>0}$) and $\beta = 10$ by giving an explicit expression for N in Theorem 3.4. To establish this counting formula, we need the following two lemmata. Lemma 3.2 counts the number of integer multiple of c in a positive interval $[a, b]$, while Lemma 3.3 gives a characterization of the exact points for the square root.

Lemma 3.2. *Given $a, b \in \mathbb{R}$ such that $0 \leq a \leq b$, and $c \in \mathbb{N}_{>0}$, let $N_{a,b}$ be the cardinal of $\{d \in \mathbb{N} \mid c \text{ divides } d \in [a, b)\}$. Then $N_{a,b} = \lceil b/c \rceil - \lceil a/c \rceil$.*

Proof. Since $[a, b) = [0, b) \backslash [0, a)$, one has $N_{a,b} = N_{0,b} - N_{0,a}$. Hence it remains to check that $N_{0,a} = \lceil a/c \rceil$. If $a \notin \mathbb{N}$, since $c \in \mathbb{N}_{>0}$, then $N_{0,a} = 1 + \lfloor a/c \rfloor$. If $a \in \mathbb{N}$, either c divides a in which case $N_{0,a} = a/c$, otherwise $N_{0,a} = 1 + \lfloor a/c \rfloor$. □

Lemma 3.3. *Let y be a positive number in $\mathbb{F}_{\beta,p}$. The real number $\sqrt{y}$ is also in $\mathbb{F}_{\beta,p}$ if and only if the integral significand Y of y satisfies $\beta^{p-1} \leq Y < \beta^p$ and $Y = Z^2 \cdot \beta^{1-p-(e_y \bmod 2)}$ for some integer Z such that $\beta^{p-1} \leq Z < \beta^p$.*

Proof. Let $z = \sqrt{y}$. Assume first that $z \in \mathbb{F}_{\beta,p}$. Then there exists an integer Z such that $z = Z \cdot \beta^{e_z - p + 1}$ and $\beta^{p-1} \leq Z < \beta^p$. Using $y = z^2$ and $y = Y \cdot \beta^{e_y - p + 1}$, we deduce that

$$Y = Z^2 \cdot \beta^{1-p-(e_y - 2e_z)}. \tag{3.3}$$

The "only if" statement then follows from (3.2). Conversely, using $y = Y \cdot \beta^{e_y - p + 1}$, we may rewrite the equality $Y = Z^2 \cdot \beta^{1-p-(e_y \bmod 2)}$ as

$$\sqrt{y} = Z \cdot \beta^{e_z - p + 1}, \quad \text{where } e_z = (e_y - (e_y \bmod 2))/2.$$

By definition, e_z is an integer and, by assumption, Z is an integer lying in $[\beta^{p-1}, \beta^p)$. Therefore, $\sqrt{y}$ belongs to $\mathbb{F}_{\beta,p}$. □

Theorem 3.4. *Let $\delta_y = (e_y + p - 1) \bmod 2$. For a given e_y, the number N of positive values $y \in \mathbb{F}_{\beta,p}$ such that $\sqrt{y} \in \mathbb{F}_{\beta,p}$ is given by*

$$N = \begin{cases} \left\lceil 2^{(qp-\delta_y(q \bmod 2))/2} \right\rceil - \left\lceil 2^{(q(p-1)-\delta_y(q \bmod 2))/2} \right\rceil, & \text{if } \beta = 2^q,\ q \in \mathbb{N}_{>0}; \\ \left\lceil 10^{(p-\delta_y)/2} \right\rceil - \left\lceil 10^{(p-1-\delta_y)/2} \right\rceil, & \text{if } \beta = 10. \end{cases}$$

Proof. Let $\gamma = p - 1 + (e_y \bmod 2)$. From Lemma 3.3, N is the number of integers Y in $[\beta^{p-1}, \beta^p)$ and of the form $Z^2 \cdot \beta^{-\gamma}$ for some integer Z such that $\beta^{p-1} \le Z < \beta^p$. Rewriting $Y = Z^2 \cdot \beta^{-\gamma}$ as

$$Y \cdot \beta^{\delta_y} \cdot \beta^{\gamma-\delta_y} = Z^2,$$

we see that $\beta^{\gamma-\delta_y}$ divides Z^2. Since $\delta_y = \gamma \bmod 2$, we know that $\gamma - \delta_y$ is even and, for $p \ge 1$, nonnegative. Using for instance the factorizations of $\beta^{(\gamma-\delta_y)/2}$ and Z into primes, we deduce that $\beta^{(\gamma-\delta_y)/2}$ divides Z. Consequently, there exists a positive integer X such that

$$Y \cdot \beta^{\delta_y} = X^2 \qquad \text{and} \qquad Z = X \cdot \beta^{(\gamma-\delta_y)/2}.$$

Now, the assumption $\beta^{p-1} \le Y < \beta^p$ is equivalent to

$$\beta^{(p-1+\delta_y)/2} \le X < \beta^{(p+\delta_y)/2}, \tag{3.4}$$

while the same assumption on Z is equivalent to $\beta^{p-1-(\gamma-\delta_y)/2} \le X < \beta^{p-(\gamma-\delta_y)/2}$. The latter interval contains the former because $p - 1 \le \delta \le p$. Hence, N is the number of integers X satisfying (3.4) and whose square is an integer multiple of β^{δ_y}. We distinguish between the two cases $\delta_y = 0$ and $\delta_y = 1$.

- If $\delta_y = 0$ then N is the number of integers X satisfying (3.4). Consequently, $N = \lceil \beta^{p/2} \rceil - \lceil \beta^{(p-1)/2} \rceil$ (using either Lemma 3.2 with $c = 1$, or [20, (3.12)]).
- If $\delta_y = 1$ then X^2 is a multiple of β: When β has linear factors only (like $\beta = 2$ or $\beta = 10 = 2 \cdot 5$), this implies that X is a multiple of β. In this case, N is the number of integers X that are multiples of β and satisfy

$$\beta^{p/2} \le X < \beta^{(p+1)/2}.$$

Hence, using Lemma 3.2,

$$N = \lceil \beta^{(p-1)/2} \rceil - \lceil \beta^{(p-2)/2} \rceil.$$

Assume now that $\beta = 2^q$ for some positive integer q. If q is even then 2^q divides X^2 implies $2^{q/2}$ divides X, so that we take the number of X's being an integer multiple of $2^{q/2}$. Lemma 3.2 thus gives

$$N = \lceil 2^{qp/2} \rceil - \lceil 2^{q(p-1)/2} \rceil.$$

If q is odd then $Y \cdot 2 = (X \cdot 2^{-\lfloor q/2 \rfloor})^2$, which means that $X \cdot 2^{-\lfloor q/2 \rfloor}$ is even. Hence we keep all the X's that are an integer multiple of $2^{1+\lfloor q/2 \rfloor}$. Using Lemma 3.2, this gives

$$N = \lceil 2^{(qp-1)/2} \rceil - \lceil 2^{(q(p-1)-1)/2} \rceil.$$

□

For a fixed e_y, using Theorem 3.4, one can count the number of input floating-point numbers y whose square root is an exact point. Tables 3.2 and 3.3 give the number N of exact points for the basic formats of the IEEE 754-2008.

Also, for a fixed exponent e_y, one can see from Theorem 3.4 that the number of exact points for the square root function is $\Theta(2^{qp/2})$ when $\beta = 2^q$, and $\Theta(10^{p/2})$ when $\beta = 10$ (see for instance Graham, Knuth, and Patashnik [20, p. 448] for a precise definition of the Θ notation). Except for small precisions, Theorem 3.4 implies therefore that it can be regarded as impractical to enumerate the exact points for the square root. The exponential growth of the number of exact points is also displayed in Figure 3.1.

Table 3.2: Number of exact point for the square-root function for various binary formats

Format	binary16	binary32	binary64	binary128
p	11	24	53	113
$\delta_y = 0$	14	1199	$\approx 2.78 \cdot 10^7$	$\approx 2.98 \cdot 10^{16}$
$\delta_y = 1$	9	849	$\approx 1.97 \cdot 10^7$	$\approx 2.11 \cdot 10^{16}$

Table 3.3: Number of exact point for the square-root function for various decimal formats

Format	decimal32	decimal64	decimal128
p	7	16	34
$\delta_y = 0$	2163	$\approx 6.84 \cdot 10^7$	$\approx 6.84 \cdot 10^{16}$
$\delta_y = 1$	683	$\approx 2.16 \cdot 10^7$	$\approx 2.16 \cdot 10^{16}$

3.2 Reciprocal square root

3.2.1 Midpoints for reciprocal square root

Theorem 3.5. *Let $y \in \mathbb{F}_{\beta,p}$ be positive and let δ_y denote $e_y \bmod 2$. If $\beta = 2^q$ ($q \in \mathbb{N}_{>0}$) then $1/\sqrt{y} \notin \mathbb{M}_{\beta,p}$. If $\beta = 10$, one has $1/\sqrt{y} \in \mathbb{M}_{\beta,p}$ if and only if the integral significand Y of y has the form*

$$Y = 2^{3p-\delta_y+1} \cdot 5^{3p-2\ell-\delta_y-1},$$

with $\ell \in \mathbb{N}$ such that $\ell \le (3p - \delta_y - 1)/2$ and

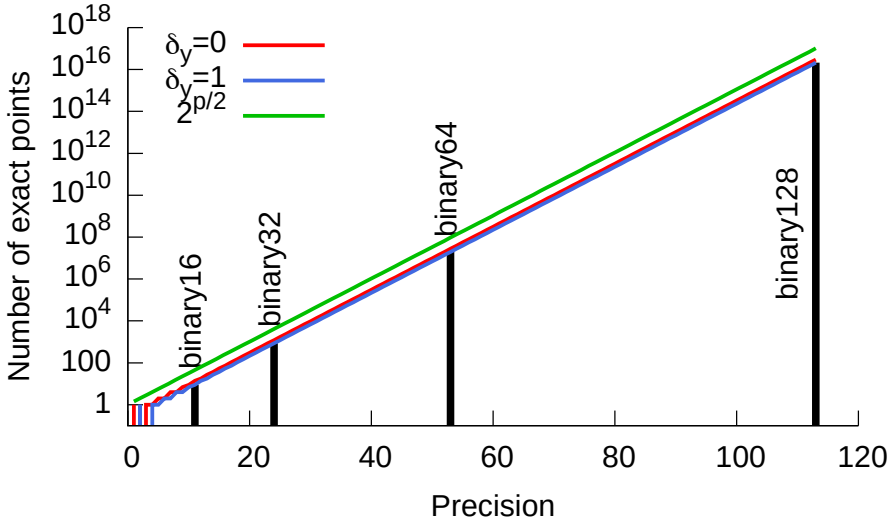
$$\begin{cases} 2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^{p-1/2}, & \text{if } e_y \text{ is odd,} \\ 2 \cdot 10^{p-1/2} < 5^\ell < 2 \cdot 10^p, & \text{if } e_y \text{ is even.} \end{cases} \tag{3.5}$$

Proof. Let $z = 1/\sqrt{y}$ and assume $z \in \mathbb{M}_{\beta,p}$. Let $y = Y \cdot \beta^{e_y-p+1}$ and $z = (Z+1/2) \cdot \beta^{e_z-p+1}$ be the normalized representations of y and z. From $yz^2 = 1$ we deduce that

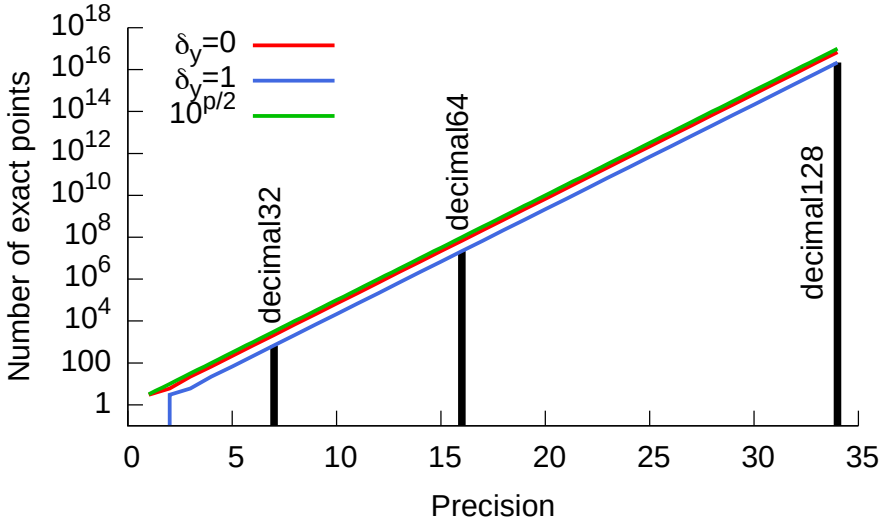
$$Y(2Z+1)^2 = 4 \cdot \beta^{-e_y-2e_z+3p-3}. \tag{3.6}$$

Since z is a midpoint, one has $\beta^{e_z} < z < \beta^{e_z+1}$ and so $\beta^{-2e_z-2} < y < \beta^{-2e_z}$. From this, one may check that $1 \le -e_y - 2e_z \le 2$. If e_y is even, then we have $-e_y - 2e_z = 2$, otherwise $-e_y - 2e_z = 1$. Hence,

$$-e_y - 2e_z = 2 - \delta_y, \qquad \delta_y = e_y \bmod 2. \tag{3.7}$$



(a) binary



(b) decimal

Figure 3.1: Number of exact points for the square-root function.

Hence we obtain from Equations (3.6) and (3.7)

$$Y(2Z+1)^2 = 4 \cdot \beta^{3p-\delta_y-1}. \tag{3.8}$$

When $\beta = 2^q$, Equation (3.8) has no solution, since the right-hand side of the equality is a power of two while the left-hand side has an odd factor $(2Z+1)^2$.

Let us now consider the case where $\beta = 10$. Equation (3.8) then becomes

$$Y(2Z+1)^2 = 2^{3p-\delta_y+1} \cdot 5^{3p-\delta_y-1}. \tag{3.9}$$

Since $2Z+1$ is odd, we deduce from (3.9) that $2Z+1 = 5^\ell$ for some $\ell \in \mathbb{N}$. Hence

$$Y = 2^{3p-\delta_y+1} \cdot 5^{3p-2\ell-\delta_y-1}$$

and it remains to prove the bounds on ℓ. Since Y is an integer, we have $3p-2\ell-\delta_y-1 \geq 0$, and the first bound $\ell \leq (3p-\delta_y-1)/2$ follows. To prove the bounds in (3.5), note first that $10^{e_y} \leq y < 10^{e_y+1}$ and (3.7) give $10^{e_z+(1-\delta_y)/2} < z = 1/\sqrt{y} \leq 10^{e_z+1-\delta_y/2}$. Then, using $z = (Z+1/2) \cdot 10^{e_z-p+1}$, we obtain

$$2 \cdot 10^{p-(\delta_y+1)/2} < 2Z+1 = 5^\ell \leq 2 \cdot 10^{p-\delta_y/2}.$$

In fact, the upper bound is strict, for 5^ℓ is an odd integer while $2 \cdot 10^{p-\delta_y/2}$ is either an even integer ($\delta_y = 0$) or an irrational number ($\delta_y = 1$).

Conversely, let $Y = 2^{3p-\delta_y+1} \cdot 5^{3p-2\ell-\delta_y-1}$, with ℓ as in (3.5), and let $z = 1/\sqrt{y}$. From (3.8) we deduce that $y = 2^{2p-2e_z} \cdot 5^{2p-2\ell-2e_z-2}$ and $z = \big((5^\ell-1)/2+1/2\big) \cdot 10^{1-p+e_z}$. Now $2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^p$ implies $10^{p-1} \leq (5^\ell-1)/2 < 10^p$ and thus $z \in \mathbb{M}_{10,p}$. □

To find in radix 10 the significands Y of all the inputs y such that $1/\sqrt{y}$ is a midpoint, it suffices to find the at most two $\ell \in \mathbb{N}$ such that $2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^p$, and to determine from the bounds (3.5) whether e_y is even or odd. Table 3.4 gives the integral significands Y and the parity of the exponent e_y such that $z = 1/\sqrt{y}$ is a midpoint in the basic decimal formats of IEEE 754-2008.

Table 3.4: Integral significands Y of $y \in \mathbb{F}_{10,p}$ such that $1/\sqrt{y} \in \mathbb{M}_{10,p}$, for the decimal formats of the IEEE 754-2008 standard [27].

Format	Integral significand Y	e_y
decimal32 ($p=7$)	$2^{22} \cdot 5^0 = 4194304$	even
decimal64 ($p=16$)	$2^{48} \cdot 5^2 = 7036874417766400$	odd
	$2^{49} \cdot 5^1 = 2814749767106560$	even
decimal128 ($p=34$)	$2^{102} \cdot 5^4 = 3169126500570573503741758013440000$	odd
	$2^{103} \cdot 5^3 = 1267650600228229401496703205376000$	even

Notice that for radices different from 10 or a power of 2, we do not have general results (which is in contrast with square root; see Section 3.1.1). Equation (3.8) may have solutions:

Example 3.6. In radix 3, $\frac{1}{\sqrt{4}} = \frac{1}{2}$ is a midpoint for the reciprocal square-root function for any precision $p \geq 2$. For example, with $p = 6$, one has

$$\big((110000)_3 \cdot 3^{-4}\big)^{-1/2} = ((111111)_3 + 1/2) \cdot 3^{-6}.$$

3.2.2 Exact points for reciprocal square root

The following theorem gives a characterization of the exact points of the square-root reciprocal when the radix is a power of a prime number, which includes the most frequent case $\beta = 2$ and $\beta = 2^q$. The case $\beta = 10$ is treated separately in Theorem 3.8.

Theorem 3.7. *Let $y \in \mathbb{F}_{\beta,p}$ be positive. Let $\beta = m^q$ with m a prime number. One has $1/\sqrt{y} \in \mathbb{F}_{\beta,p}$ if and only if $y = m^{2k}$ with $k \in \mathbb{Z}$.*

Proof. Taking $z = 1/\sqrt{y}$, note first that (3.7) still holds. Now assume that $z \in \mathbb{F}_{\beta,p}$ and let Y and Z be the integral significands of y and z. From $yz^2 = 1$ and (3.7), we deduce that

$$YZ^2 = \beta^{3p-\delta_y-1}. \tag{3.10}$$

If $\beta = m^q$, we deduce from (3.10) that $Z = m^\ell$ for some $\ell \in \mathbb{N}$. Hence $Y = m^{q(3p-\delta_y-1)-2\ell}$ and, using (3.7), $y = m^{2(qp-q-qe_z-\ell)}$ is indeed an even power of m. Conversely, if $y = m^{2k}$, then $z = m^{-k}$ is in $\mathbb{F}_{\beta,p}$. □

All the floating-point numbers y such that $1/\sqrt{y}$ is an exact point can be deduced from the ones lying in the interval $[1, \beta^2)$. In radix 2^q, Theorem 3.7 implies that at most q values of y in $[1, 2^{2q})$ suffice to characterize the exact points for the reciprocal square root. In radix $16 = 2^4$ for instance, the only exact points for input values $y \in [1, 256)$ are:

y	1	4	16	64
$1/\sqrt{y}$	1	$1/2 = 0.8_{16}$	$1/4 = 0.4_{16}$	$1/8 = 0.2_{16}$

Theorem 3.8. *Let $y \in \mathbb{F}_{10,p}$ be positive and let δ_y denote $e_y \bmod 2$. One has $1/\sqrt{y} \in \mathbb{F}_{10,p}$ if and only if either $y = 10^{-2e_z}$ or the integral significand Y of y differs from 10^{p-1} and has the form*

$$Y = 2^{3p-1-\delta_y-2k} \cdot 5^{3p-1-\delta_y-2\ell},$$

with $k, \ell \in \mathbb{N}$ such that $0 \le k, \ell \le (3p-1-\delta_y)/2$.

Proof. Let $z = 1/\sqrt{y}$ and assume $z \in \mathbb{F}_{10,p}$. If $z = 10^{e_z}$ then obviously $y = 10^{-2e_z}$. On the other hand, z must differ from the irrational number $10^{e_z+1/2}$. Hence we now assume

$$z \in (10^{e_z}, 10^{e_z+1/2}) \cup (10^{e_z+1/2}, 10^{e_z+1}).$$

This implies

$$y \in (10^{-2e_z-2}, 10^{-2e_z-1}) \cup (10^{-2e_z-1}, 10^{-2e_z}).$$

Therefore, y is not a power of 10 and its normalized representation $y = Y \cdot 10^{e_y-p+1}$ is such that $Y \neq 10^{p-1}$. Note now that (3.7) and (3.10) still hold here, so that $yz^2 = 1$ implies $YZ^2 = 10^{3p-1-\delta_y}$. In particular, Z must have the form $Z = 2^k \cdot 5^\ell$ for some k, ℓ in $\mathbb{N}$. Thus

$$Y = 2^{3p-1-\delta_y-2k} \cdot 5^{3p-1-\delta_y-2\ell},$$

where, since Y is an integer, $0 \le k, \ell \le (3p-1-\delta_y)/2$.

Conversely, the case $y = 10^{-2e_z}$ being trivial, let $Y = 2^{3p-1-\delta_y-2k} \cdot 5^{3p-1-\delta_y-2\ell}$ be the integral significand of y such that $10^{p-1} < Y < 10^p$, and let $z = 1/\sqrt{y}$. Using (3.7) further leads to $z = 2^k \cdot 5^\ell \cdot 10^{e_z-p+1}$. One has $2^k \cdot 5^\ell \in \mathbb{N}$ and, from $10^{p-1} < Y < 10^p$, we get $10^{p-(1+\delta_y)/2} < 2^k \cdot 5^\ell < 10^{p-\delta_y/2}$. Hence $z \in \mathbb{F}_{10,p}$. □

Enumerating the integral significands

$$Y = 2^{3p-1-\delta_y-2k} \cdot 5^{3p-1-\delta_y-2\ell},$$

with $k, \ell \in \mathbb{N}$ such that $0 \leq k, \ell \leq (3p-1-\delta_y)/2$ and $10^{p-1} < Y < 10^p$ is easily done by a simple program, like for example Algorithm 1.

Data: Precision p, $\delta_y = e_y \bmod 2$
Result: Integral significands Y such that $y = Y \cdot 10^{e_y-p+1}$ yields an exact point for the reciprocal square-root function

print 10^{p-1}; `/* First, take into account the case` $y = 10^{-2e_x}$ `*/`
$\min_Y = 10^{p-1}$;
$\max_Y = 10^p$;
$\max_{k\ell} = \lfloor (3p-1-\delta_y)/2 \rfloor$;
for k **from** 0 **to** $\max_{k\ell}$ **do**
  for ℓ **from** 0 **to** $\max_{k\ell}$ **do**
    $Y = 2^{3p-1-\delta_y-2k} \cdot 5^{3p-1-\delta_y-2\ell}$; `/* Y as in Theorem 3.8 */`
    if $\min_Y < Y < \max_Y$ **then**
      print Y ; `/* Y is an integral significand */`
    end
  end
end

Algorithm 1: Reciprocal_Sqrt_Exactpoints (int p, int δ_y)

Tables 3.5, 3.6, 3.7 and 3.8 give all the integral significands Y of y, and the parity of the exponent e_y, such that $1/\sqrt{y}$ is a floating-point number too, in the decimal32, decimal64 and decimal128 formats. Figures 3.2, 3.3 and 3.4 also display the distribution of the inputs that leads to exact points for the reciprocal square-root on the decade $[1, 100)$.

For the basic decimal formats of the IEEE 754-2008, the table below gives the number of significands Y such that $1/\sqrt{y}$ is an exact point, with respect to the parity δ_y of the exponent of y.

Format	decimal32	decimal64	decimal128
p	7	16	34
$\delta_y = 0$	9	17	37
$\delta_y = 1$	7	17	36

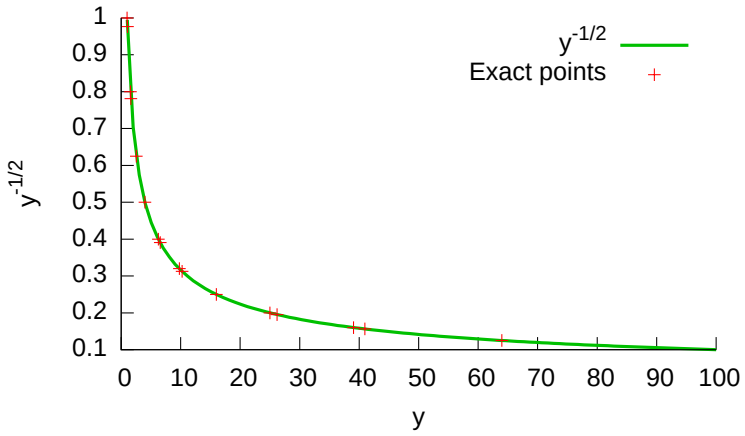


Figure 3.2: The exactpoints of the reciprocal square-root function for the decimal32 format.

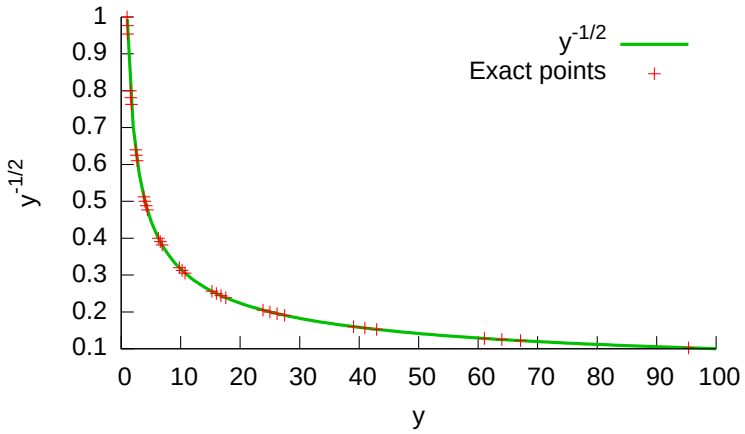


Figure 3.3: The exactpoints of the reciprocal square-root function for the decimal64 format.

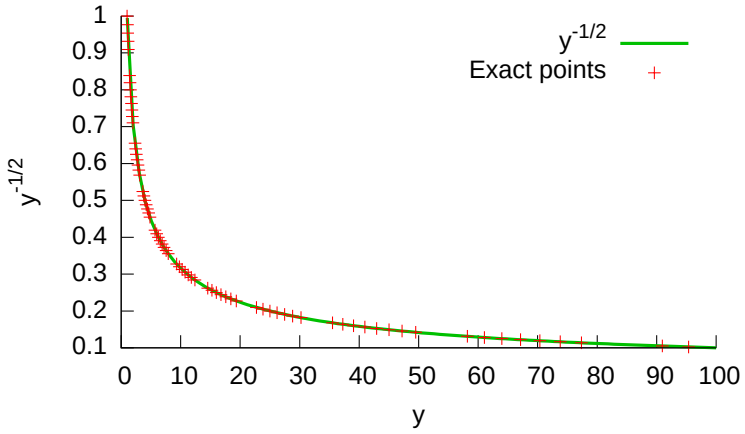


Figure 3.4: The exactpoints of the reciprocal square-root function for the decimal128 format.

Table 3.5: Integral significands Y of $y \in \mathbb{F}_{10,7}$, such that $1/\sqrt{y} \in \mathbb{F}_{10,7}$.

Y	$1/\sqrt{Y \cdot 10^{\delta_y - p + 1}}$	e_y
$2^6 \cdot 5^6 = 1000000$	$1.000000 \cdot 10^0$	even
$2^{20} \cdot 5^0 = 1048576$	$9.765625 \cdot 10^{-1}$	even
$2^{18} \cdot 5^2 = 6553600$	$3.906250 \cdot 10^{-1}$	even
$2^{16} \cdot 5^2 = 1638400$	$7.812500 \cdot 10^{-1}$	even
$2^{12} \cdot 5^4 = 2560000$	$6.250000 \cdot 10^{-1}$	even
$2^8 \cdot 5^6 = 4000000$	$5.000000 \cdot 10^{-1}$	even
$2^4 \cdot 5^8 = 6250000$	$4.000000 \cdot 10^{-1}$	even
$2^2 \cdot 5^8 = 1562500$	$8.000000 \cdot 10^{-1}$	even
$2^0 \cdot 5^{10} = 9765625$	$3.200000 \cdot 10^{-1}$	even
$2^{19} \cdot 5^1 = 2621440$	$1.953125 \cdot 10^{-1}$	odd
$2^{15} \cdot 5^3 = 4096000$	$1.562500 \cdot 10^{-1}$	odd
$2^{13} \cdot 5^3 = 1024000$	$3.125000 \cdot 10^{-1}$	odd
$2^{11} \cdot 5^5 = 6400000$	$1.250000 \cdot 10^{-1}$	odd
$2^9 \cdot 5^5 = 1600000$	$2.500000 \cdot 10^{-1}$	odd
$2^5 \cdot 5^7 = 2500000$	$2.000000 \cdot 10^{-1}$	odd
$2^1 \cdot 5^9 = 3906250$	$1.600000 \cdot 10^{-1}$	odd

Table 3.6: Integral significands Y of $y \in \mathbb{F}_{10,16}$, such that $1/\sqrt{y} \in \mathbb{F}_{10,16}$.

Y	$1/\sqrt{Y \cdot 10^{\delta_y - p + 1}}$	e_y
$2^{15}{\cdot}5^{15} = 1000000000000000$	$1.000000000000000 \cdot 10^{0}$	even
$2^{45}{\cdot}5^{3} = 4398046511104000$	$4.768371582031250 \cdot 10^{-1}$	even
$2^{43}{\cdot}5^{3} = 1099511627776000$	$9.536743164062500 \cdot 10^{-1}$	even
$2^{41}{\cdot}5^{5} = 6871947673600000$	$3.814697265625000 \cdot 10^{-1}$	even
$2^{39}{\cdot}5^{5} = 1717986918400000$	$7.629394531250000 \cdot 10^{-1}$	even
$2^{35}{\cdot}5^{7} = 2684354560000000$	$6.103515625000000 \cdot 10^{-1}$	even
$2^{31}{\cdot}5^{9} = 4194304000000000$	$4.882812500000000 \cdot 10^{-1}$	even
$2^{29}{\cdot}5^{9} = 1048576000000000$	$9.765625000000000 \cdot 10^{-1}$	even
$2^{27}{\cdot}5^{11} = 6553600000000000$	$3.906250000000000 \cdot 10^{-1}$	even
$2^{25}{\cdot}5^{11} = 1638400000000000$	$7.812500000000000 \cdot 10^{-1}$	even
$2^{21}{\cdot}5^{13} = 2560000000000000$	$6.250000000000000 \cdot 10^{-1}$	even
$2^{17}{\cdot}5^{15} = 4000000000000000$	$5.000000000000000 \cdot 10^{-1}$	even
$2^{13}{\cdot}5^{17} = 6250000000000000$	$4.000000000000000 \cdot 10^{-1}$	even
$2^{11}{\cdot}5^{17} = 1562500000000000$	$8.000000000000000 \cdot 10^{-1}$	even
$2^{9}{\cdot}5^{19} = 9765625000000000$	$3.200000000000000 \cdot 10^{-1}$	even
$2^{7}{\cdot}5^{19} = 2441406250000000$	$6.400000000000000 \cdot 10^{-1}$	even
$2^{3}{\cdot}5^{21} = 3814697265625000$	$5.120000000000000 \cdot 10^{-1}$	even
$2^{46}{\cdot}5^{2} = 1759218604441600$	$2.384185791015625 \cdot 10^{-1}$	odd
$2^{42}{\cdot}5^{4} = 2748779069440000$	$1.907348632812500 \cdot 10^{-1}$	odd
$2^{38}{\cdot}5^{6} = 4294967296000000$	$1.525878906250000 \cdot 10^{-1}$	odd
$2^{36}{\cdot}5^{6} = 1073741824000000$	$3.051757812500000 \cdot 10^{-1}$	odd
$2^{34}{\cdot}5^{8} = 6710886400000000$	$1.220703125000000 \cdot 10^{-1}$	odd
$2^{32}{\cdot}5^{8} = 1677721600000000$	$2.441406250000000 \cdot 10^{-1}$	odd
$2^{28}{\cdot}5^{10} = 2621440000000000$	$1.953125000000000 \cdot 10^{-1}$	odd
$2^{24}{\cdot}5^{12} = 4096000000000000$	$1.562500000000000 \cdot 10^{-1}$	odd
$2^{22}{\cdot}5^{12} = 1024000000000000$	$3.125000000000000 \cdot 10^{-1}$	odd
$2^{20}{\cdot}5^{14} = 6400000000000000$	$1.250000000000000 \cdot 10^{-1}$	odd
$2^{18}{\cdot}5^{14} = 1600000000000000$	$2.500000000000000 \cdot 10^{-1}$	odd
$2^{14}{\cdot}5^{16} = 2500000000000000$	$2.000000000000000 \cdot 10^{-1}$	odd
$2^{10}{\cdot}5^{18} = 3906250000000000$	$1.600000000000000 \cdot 10^{-1}$	odd
$2^{6}{\cdot}5^{20} = 6103515625000000$	$1.280000000000000 \cdot 10^{-1}$	odd
$2^{4}{\cdot}5^{20} = 1525878906250000$	$2.560000000000000 \cdot 10^{-1}$	odd
$2^{2}{\cdot}5^{22} = 9536743164062500$	$1.024000000000000 \cdot 10^{-1}$	odd
$2^{0}{\cdot}5^{22} = 2384185791015625$	$2.048000000000000 \cdot 10^{-1}$	odd

Table 3.7: Integral significands Y of $y \in \mathbb{F}_{10,34}$, such that $1/\sqrt{y} \in \mathbb{F}_{10,34}$ and e_y even.

Y	$1/\sqrt{Y \cdot 10^{\delta_y - p + 1}}$
$2^{33}.5^{33} = 1000000000000000000000000000000000$	$1.000000000000000000000000000000000 \times 10^{0}$
$2^{101}.5^{5} = 7922816251426433759354395033600000$	$3.552713678800500929355621337890625 \times 10^{-1}$
$2^{99}.5^{5} = 1980704062856608439838598758400000$	$7.105427357601001858711242675781250 \times 10^{-1}$
$2^{95}.5^{7} = 3094850098213450687247810560000000$	$5.684341886080801486968994140625000 \times 10^{-1}$
$2^{91}.5^{9} = 4835703278458516698824704000000000$	$4.547473508864641189575195312500000 \times 10^{-1}$
$2^{89}.5^{9} = 1208925819614629174706176000000000$	$9.094947017729282379150390625000000 \times 10^{-1}$
$2^{87}.5^{11} = 7555786372591432341913600000000000$	$3.637978807091712951660156250000000 \times 10^{-1}$
$2^{85}.5^{11} = 1888946593147858085478400000000000$	$7.275957614183425903320312500000000 \times 10^{-1}$
$2^{81}.5^{13} = 2951479051793528258560000000000000$	$5.820766091346740722656250000000000 \times 10^{-1}$
$2^{77}.5^{15} = 4611686018427387904000000000000000$	$4.656612873077392578125000000000000 \times 10^{-1}$
$2^{75}.5^{15} = 1152921504606846976000000000000000$	$9.313225746154785156250000000000000 \times 10^{-1}$
$2^{73}.5^{17} = 7205759403792793600000000000000000$	$3.725290298461914062500000000000000 \times 10^{-1}$
$2^{71}.5^{17} = 1801439850948198400000000000000000$	$7.450580596923828125000000000000000 \times 10^{-1}$
$2^{67}.5^{19} = 2814749767106560000000000000000000$	$5.960464477539062500000000000000000 \times 10^{-1}$
$2^{63}.5^{21} = 4398046511104000000000000000000000$	$4.768371582031250000000000000000000 \times 10^{-1}$
$2^{61}.5^{21} = 1099511627776000000000000000000000$	$9.536743164062500000000000000000000 \times 10^{-1}$
$2^{59}.5^{23} = 6871947673600000000000000000000000$	$3.814697265625000000000000000000000 \times 10^{-1}$
$2^{57}.5^{23} = 1717986918400000000000000000000000$	$7.629394531250000000000000000000000 \times 10^{-1}$
$2^{53}.5^{25} = 2684354560000000000000000000000000$	$6.103515625000000000000000000000000 \times 10^{-1}$
$2^{49}.5^{27} = 4194304000000000000000000000000000$	$4.882812500000000000000000000000000 \times 10^{-1}$
$2^{47}.5^{27} = 1048576000000000000000000000000000$	$9.765625000000000000000000000000000 \times 10^{-1}$
$2^{45}.5^{29} = 6553600000000000000000000000000000$	$3.906250000000000000000000000000000 \times 10^{-1}$
$2^{43}.5^{29} = 1638400000000000000000000000000000$	$7.812500000000000000000000000000000 \times 10^{-1}$
$2^{39}.5^{31} = 2560000000000000000000000000000000$	$6.250000000000000000000000000000000 \times 10^{-1}$
$2^{35}.5^{33} = 4000000000000000000000000000000000$	$5.000000000000000000000000000000000 \times 10^{-1}$
$2^{31}.5^{35} = 6250000000000000000000000000000000$	$4.000000000000000000000000000000000 \times 10^{-1}$
$2^{29}.5^{35} = 1562500000000000000000000000000000$	$8.000000000000000000000000000000000 \times 10^{-1}$
$2^{27}.5^{37} = 9765625000000000000000000000000000$	$3.200000000000000000000000000000000 \times 10^{-1}$
$2^{25}.5^{37} = 2441406250000000000000000000000000$	$6.400000000000000000000000000000000 \times 10^{-1}$
$2^{21}.5^{39} = 3814697265625000000000000000000000$	$5.120000000000000000000000000000000 \times 10^{-1}$
$2^{17}.5^{41} = 5960464477539062500000000000000000$	$4.096000000000000000000000000000000 \times 10^{-1}$
$2^{15}.5^{41} = 1490116119384765625000000000000000$	$8.192000000000000000000000000000000 \times 10^{-1}$
$2^{13}.5^{43} = 9313225746154785156250000000000000$	$3.276800000000000000000000000000000 \times 10^{-1}$
$2^{11}.5^{43} = 2328306436538696289062500000000000$	$6.553600000000000000000000000000000 \times 10^{-1}$
$2^{7}.5^{45} = 3637978807091712951660156250000000$	$5.242880000000000000000000000000000 \times 10^{-1}$
$2^{3}.5^{47} = 5684341886080801486968994140625000$	$4.194304000000000000000000000000000 \times 10^{-1}$
$2^{1}.5^{47} = 1421085471520200371742248535156250$	$8.388608000000000000000000000000000 \times 10^{-1}$

Table 3.8: Integral significands Y of $y \in \mathbb{F}_{10,34}$, such that $1/\sqrt{y} \in \mathbb{F}_{10,34}$ and e_y odd.

Y	$1/\sqrt{Y \cdot 10^{\delta_y - p + 1}}$
$2^{98}\cdot 5^{6} = 4951760157141521099596496896000000$	$1.421085471520200371742248535156250 \times 10^{-1}$
$2^{96}\cdot 5^{6} = 1237940039285380274899124224000000$	$2.842170943040400743484497070312500 \times 10^{-1}$
$2^{94}\cdot 5^{8} = 7737125245533626718119526400000000$	$1.136868377216160297393798828125000 \times 10^{-1}$
$2^{92}\cdot 5^{8} = 1934281311383406679529881600000000$	$2.273736754432320594787597656250000 \times 10^{-1}$
$2^{88}\cdot 5^{10} = 3022314549036572936765440000000000$	$1.818989403545856475830078125000000 \times 10^{-1}$
$2^{84}\cdot 5^{12} = 4722366482869645213696000000000000$	$1.455191522836685180664062500000000 \times 10^{-1}$
$2^{82}\cdot 5^{12} = 1180591620717411303424000000000000$	$2.910383045673370361328125000000000 \times 10^{-1}$
$2^{80}\cdot 5^{14} = 7378697629483820646400000000000000$	$1.164153218269348144531250000000000 \times 10^{-1}$
$2^{78}\cdot 5^{14} = 1844674407370955161600000000000000$	$2.328306436538696289062500000000000 \times 10^{-1}$
$2^{74}\cdot 5^{16} = 2882303761517117440000000000000000$	$1.862645149230957031250000000000000 \times 10^{-1}$
$2^{70}\cdot 5^{18} = 4503599627370496000000000000000000$	$1.490116119384765625000000000000000 \times 10^{-1}$
$2^{68}\cdot 5^{18} = 1125899906842624000000000000000000$	$2.980232238769531250000000000000000 \times 10^{-1}$
$2^{66}\cdot 5^{20} = 7036874417766400000000000000000000$	$1.192092895507812500000000000000000 \times 10^{-1}$
$2^{64}\cdot 5^{20} = 1759218604441600000000000000000000$	$2.384185791015625000000000000000000 \times 10^{-1}$
$2^{60}\cdot 5^{22} = 2748779069440000000000000000000000$	$1.907348632812500000000000000000000 \times 10^{-1}$
$2^{56}\cdot 5^{24} = 4294967296000000000000000000000000$	$1.525878906250000000000000000000000 \times 10^{-1}$
$2^{54}\cdot 5^{24} = 1073741824000000000000000000000000$	$3.051757812500000000000000000000000 \times 10^{-1}$
$2^{52}\cdot 5^{26} = 6710886400000000000000000000000000$	$1.220703125000000000000000000000000 \times 10^{-1}$
$2^{50}\cdot 5^{26} = 1677721600000000000000000000000000$	$2.441406250000000000000000000000000 \times 10^{-1}$
$2^{46}\cdot 5^{28} = 2621440000000000000000000000000000$	$1.953125000000000000000000000000000 \times 10^{-1}$
$2^{42}\cdot 5^{30} = 4096000000000000000000000000000000$	$1.562500000000000000000000000000000 \times 10^{-1}$
$2^{40}\cdot 5^{30} = 1024000000000000000000000000000000$	$3.125000000000000000000000000000000 \times 10^{-1}$
$2^{38}\cdot 5^{32} = 6400000000000000000000000000000000$	$1.250000000000000000000000000000000 \times 10^{-1}$
$2^{36}\cdot 5^{32} = 1600000000000000000000000000000000$	$2.500000000000000000000000000000000 \times 10^{-1}$
$2^{32}\cdot 5^{34} = 2500000000000000000000000000000000$	$2.000000000000000000000000000000000 \times 10^{-1}$
$2^{28}\cdot 5^{36} = 3906250000000000000000000000000000$	$1.600000000000000000000000000000000 \times 10^{-1}$
$2^{24}\cdot 5^{38} = 6103515625000000000000000000000000$	$1.280000000000000000000000000000000 \times 10^{-1}$
$2^{22}\cdot 5^{38} = 1525878906250000000000000000000000$	$2.560000000000000000000000000000000 \times 10^{-1}$
$2^{20}\cdot 5^{40} = 9536743164062500000000000000000000$	$1.024000000000000000000000000000000 \times 10^{-1}$
$2^{18}\cdot 5^{40} = 2384185791015625000000000000000000$	$2.048000000000000000000000000000000 \times 10^{-1}$
$2^{14}\cdot 5^{42} = 3725290298461914062500000000000000$	$1.638400000000000000000000000000000 \times 10^{-1}$
$2^{10}\cdot 5^{44} = 5820766091346740722656250000000000$	$1.310720000000000000000000000000000 \times 10^{-1}$
$2^{8}\cdot 5^{44} = 1455191522836685180664062500000000$	$2.621440000000000000000000000000000 \times 10^{-1}$
$2^{6}\cdot 5^{46} = 9094947017729282379150390625000000$	$1.048576000000000000000000000000000 \times 10^{-1}$
$2^{4}\cdot 5^{46} = 2273736754432320594787597656250000$	$2.097152000000000000000000000000000 \times 10^{-1}$
$2^{0}\cdot 5^{48} = 3552713678800500929355621337890625$	$1.677721600000000000000000000000000 \times 10^{-1}$

3.3 Positive integer powers

We consider here the function $(x, k) \mapsto x^k$ with $x \in \mathbb{R}$ and $k \in \mathbb{N}_{>0}$, assuming that each prime factor appears only once in the prime decomposition of β, which is the case for $\beta = 2$ and $\beta = 10$. We provide a sufficient condition for the nonexistence of midpoints in such radices. In the particular case $\beta = 2$, the results given in this section can be deduced easily from Lauter and Lefèvre's study of the power function $(x, y) \mapsto x^y$ [37, 38], which shows how to check quickly if x^y is a midpoint or an exact point, in double precision (binary64 format).

Definition 3.9. *A number* fits in n digits exactly *in radix β if it is a precision-n floating-point number that cannot be exactly represented in precision $n-1$. More precisely, it is a number of the form $x = X \cdot \beta^{e_x}$, where $e_x, X \in \mathbb{Z}$, $\beta^{n-1} < |X| < \beta^n$, and X is not a multiple of β.*

Lemma 3.10. *Let $k \in \mathbb{N}_{>0}$ be given. If each factor of β appears only once in its prime number decomposition (which is true for β equal to 2 or 10), and if x fits in n digits exactly then x^k fits in m digits exactly, with $m \in \mathbb{N}$ such that $k(n-1) < m \leq kn$.*

Proof. Let $x = X \cdot \beta^{e_x}$ be a number that fits in n digits exactly. From $\beta^{n-1} < |X| < \beta^n$ it follows that $\beta^{k(n-1)} < |X^k| < \beta^{kn}$. Consequently, there exists $m \in \mathbb{N}$ such that $k(n-1) < m \leq kn$ and $\beta^{m-1} < |X^k| < \beta^m$. Moreover, the assumption on the prime factor decomposition of β and the fact that β does not divide X imply that X^k is not a multiple of β. □

An immediate consequence of the previous lemma is the following result.

Theorem 3.11. *Assume the radix β is such that each factor appears only once in its prime number decomposition, and let p be the precision. If x fits in n digits exactly then x^k cannot be a midpoint as soon as $k(n-1) > p$, and it cannot be an exact point as soon as $k(n-1) + 1 > p$.*

Theorem 3.11 is not very helpful when k is small. For large values of k, however, it allows to quickly determine the possible midpoints and exact points. For instance, in the binary64 format ($\beta = 2$ and $p = 53$), the only floating-point numbers x such that x^{10} can be an exact point are those that fit in n bits exactly, where $n \leq 6$. For a given value of the exponent, there are at most $2^6 = 64$ such points: it therefore suffices to check these 64 values to know all the exact points. By accurately computing x^{10} for these 64 points, we easily find that the exact points for function x^{10} in the binary64 format correspond to the input values of the form $x = X \cdot 2^{e_x}$, where X is an integer between 0 and 40.

3.4 The function $(x, y) \mapsto x \,/\, \|y\|_2$

Given $d \in \mathbb{N}_{>0}$, the number of exact points of the function that maps $(x, y) \in \mathbb{R} \times (\mathbb{R}^d \backslash \{0\})$ to

$$\frac{x}{\|y\|_2} = \frac{x}{\sqrt{\sum_{1 \leq k \leq d} y_k^2}}$$

is huge. Indeed, all the exact points for the division operation, whose number is huge as we will see later in §3.5.2, are exact points for the function $x/\|y\|_2$ as well. Therefore, we shall focus here exclusively on midpoints: our aim is to decide whether there exist floating-point inputs $x, y_1, \ldots, y_d \in \mathbb{F}_{\beta,p}$ such that $x/\|y\|_2 \in \mathbb{M}_{\beta,p}$. We start with the following theorem, which says that midpoints cannot exist in radix 2.

Theorem 3.12. *Let $x \in \mathbb{F}_{\beta,p}$ and, for $d \in \mathbb{N}_{>0}$, let y be a nonzero, d-dimensional vector of elements of $\mathbb{F}_{\beta,p}$. If $\beta = 2$ then $x/\|y\|_2 \notin \mathbb{M}_{\beta,p}$.*

Proof. Because of the symmetries of the function that maps (x, y) to $x/\|y\|_2$, we can restrict to the case where x and all the entries of $y = [y_k]$ are positive. Hence $x = X \cdot \beta^{e_x - p+1}$ and $y_k = Y_k \cdot \beta^{e_{y_k} - p+1}$ for some integers X and Y_k such that $\beta^{p-1} \leq X, Y_k < \beta^p$. Let $z = x/\|y\|_2$ and assume z is a midpoint, that is, $z = (Z + 1/2) \cdot \beta^{e_z - p+1}$ for some integer Z in the same range as X and the Y_k above. The identity $x^2 = \|y\|_2^2 \, z^2$ thus becomes

$$4X^2 \cdot \beta^{2(e_x - e_z + p-1)} = \Big(\sum_k Y_k^2 \cdot \beta^{2e_{y_k}} \Big) (2Z + 1)^2. \tag{3.11}$$

In order to have integers on both sides, it suffices to multiply (3.11) by β^{-2e^*}, where $e^* = \min_k e_{y_k}$. This gives

$$4X^2 \cdot \beta^{2(e_x - e_z - e^* + p-1)} = \Big(\sum_k Y_k^2 \cdot \beta^{2(e_{y_k} - e^*)} \Big) (2Z + 1)^2. \tag{3.12}$$

Now, the power of β involved in the left-hand side of (3.12) is itself an integer. This is is due to the fact that the integer $e_x - e_z - e^*$ is non-negative, which can be seen as follows. Since $d \geq 1$ and $y_k \geq \beta^{e^*}$ for $k = 1, \ldots, d$, one has $z \leq x/\beta^{e^*}$. Using $x < \beta^{e_x+1}$ and $\beta^{e_z} \leq z$ (in fact this lower bound is strict, for z is a midpoint), we deduce that $\beta^{e_z} < \beta^{e_x - e^* + 1}$. The exponents on both sides of the latter inequality being integers, we conclude that $e_z \leq e_x - e^*$.

When $\beta = 2$, Equation (3.12) becomes

$$X^2 \cdot 2^{2(e_x - e_z - e^* + p)} = \Big(\sum_k Y_k^2 \cdot 2^{2(e_{y_k} - e^*)} \Big) (2Z + 1)^2. \tag{3.13}$$

The left-hand side of (3.13) is a multiple of the odd integer $(2Z + 1)^2$. Since $e_x - e_z - e^*$ is non-negative, this implies that X is a multiple of $2Z + 1$ and thus $X \geq 2Z + 1$. However, recalling that $2^{p-1} \leq X, Z < 2^p$, we have $X < 2Z + 1$.

Hence a contradiction, which concludes the proof. □

Theorem 3.12 implies the non-existence of midpoints in radix $\beta = 2$ for a number of important special cases: division x/y (see following Corollary 3.13) and thus reciprocal $1/y$ as well; reciprocal 2D Euclidean norm $1/\sqrt{x^2 + y^2}$ and 2D-vector normalization $x/\sqrt{x^2 + y^2}$.

However, when $\beta > 2$, the function $x/\|y\|_2$ does have midpoints and some examples will be given in §3.5.1 for $\beta \in \{3, 4, 10\}$. Thus, in the sequel, rather than trying to characterize all the midpoints of that general function, we focus from Sections 3.5 to 3.8 on the four special cases just mentioned.

3.5 Division

3.5.1 Midpoints for division

Concerning midpoints for division, Theorem 3.12 gives an answer for the far most frequent case in practice: the radix is 2, the input precision equals the output precision, and the results are above the underflow threshold. Indeed, choosing $d = 1$ in Theorem 3.12, we obtain the following corollary.

Corollary 3.13. *In binary arithmetic, the quotient of two floating-point numbers cannot be a midpoint in the same precision.*

In radix-2 floating-point arithmetic, Corollary 3.13 can be seen as a consequence of a result presented by Markstein in [43, Theorem 8.4, p. 114]. Note that this result only holds when $\beta = 2$ and when the input precision is less than or equal to the output precision. Nevertheless, it is sometimes believed that it holds in any prime radices: the first example given below shows that this is not the case. The following examples also illustrate the existence of midpoints when $\beta > 2$.

- In radix 3, with precision $p = 4$,
$$\frac{28_{10}}{56_{10}} = \frac{1001_3}{2002_3} = 0.1111_3 + \frac{1}{2} \cdot 3^{-4}.$$
- In radix 4, with $p = 4$
$$\frac{129_{10}}{128_{10}} = \frac{2001_4}{2000_4} = 1.000_4 + \frac{1}{2} \cdot 4^{-3}.$$
- In radix 10, midpoint quotients are quite frequent. For instance, with $p = 2$ we have 181 midpoints for X/Y with $10 \le X, Y \le 99$ (e.g., $10/16 = 0.625$), and with $p = 3$, we have 2633 cases with $100 \le X, Y \le 999$.

We now briefly discuss the case of different input (p_i) and output (p_o) precisions. If $p_i > p_o$, many quotients can be midpoints, even in radix-2 arithmetic. For example, we can compute the quotient $x/1$ in precision p_o. Since x is in precision $p_i > p_o$, x can be a midpoint in precision p_o. It is also possible to find less trivial cases.

Example 3.14. If x and y are binary64 numbers ($p_i = 53$) with

$$\begin{aligned} x &= 1.00000000000000000000000011111111111111111111110100000, \\ y &= 1.11111111111111111111110000000000000000000000000000000, \end{aligned}$$

then one has

$$x/y = 0.\underbrace{100000000000000000000001}_{p_o=24}1,$$

which is a midpoint in the binary32 floating-point format ($p_o = 24$).

A typical case in binary floating-point arithmetic arises when the output falls in the subnormal range. However, since division is one of the basic operations of the IEEE-754-2008 norm (§1.4.1), it is mandatory to handle that midpoint case correctly.

3.5.2 Exact points for division

Let x and y be two numbers in $\mathbb{F}_{\beta,p}$, and assume that the quotient $z = x/y$ is also in $\mathbb{F}_{\beta,p}$. Using the normalized representations $x = X \cdot \beta^{e_x-p+1}$, $y = Y \cdot \beta^{e_y-p+1}$ then z can be written $z = Z \cdot \beta^{e_x-e_y+\delta-p}$, with $\delta \in \{0, 1\}$. Hence from $x = yz$ it follows that

$$\beta^{p-\delta} X = YZ, \tag{3.14}$$

with $\delta \in \{0, 1\}$. In other words, if z is an exact point then Equation (3.14) must be satisfied. For any radix β, Equation (3.14) has many solutions: for each value of X there is at least the straightforward solution $(X, Y) = (Z, \beta^{p-1})$, which corresponds to x/β^{e_y}. As a consequence, the number of exact points of the function $(x, y) \mapsto x/y$ grows at least like $\beta^{p-1}(\beta - 1)$ for any given exponents e_x, e_y. This is too large to enumerate all the exact points of division in practice.

3.6 Reciprocal

As we have seen in the previous section, except in radix 2, division admits many midpoints. Moreover, whatever the radix is, division also admits a lot of exact points. Consequently, we now focus on a special case, the reciprocal function $y \mapsto 1/y$, for which more useful results can be obtained.

3.6.1 Midpoints for reciprocal

Theorem 3.15. *Let $y \in \mathbb{F}_{\beta,p}$ be nonzero. If $\beta = 2^q$ ($q \in \mathbb{N}_{>0}$) then $1/y \notin \mathbb{M}_{\beta,p}$. If $\beta = 10$, one has $1/y \in \mathbb{M}_{\beta,p}$ if and only if the integral significand Y of y has the form*

$$Y = 2^{2p} \cdot 5^{2p-1-\ell}, \tag{3.15}$$

with $\ell \in \mathbb{N}$ such that $2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^p$.

Proof. Without loss of generality, we assume $y > 0$. Let $z = 1/y$. First, one may check that

$$e_z = -e_y - 1. \tag{3.16}$$

Now, if $z \in \mathbb{M}_{\beta,p}$ then $z = (Z + 1/2) \cdot \beta^{e_z-p+1}$ for some integer Z such that $\beta^{p-1} \le Z < \beta^p$. Using $yz = 1$ thus gives

$$Y(2Z + 1) = 2 \cdot \beta^{2p-1}. \tag{3.17}$$

When $\beta = 2^q$, Equation (3.17) has no solution, since the right-hand side of the equality is a power of two while the left-hand side has an odd factor $2Z + 1$.

When $\beta = 10$, Equation (3.17) becomes

$$Y(2Z + 1) = 2^{2p} \cdot 5^{2p-1}. \tag{3.18}$$

As $2Z + 1$ is odd, we deduce from Equation (3.18) that $2Z + 1$ is necessarily a power of 5, and since $2 \cdot 10^{p-1} < 2Z + 1 < 2 \cdot 10^p$, we deduce that there are at most two such powers of 5. Hence y is necessarily as in (3.15).

Conversely, if $y = Y \cdot 10^{e_y-p+1}$ with Y as in (3.15) then, using (3.16), $y = 5^{-\ell-1} \cdot 10^{-e_z+p}$. It follows that z can be written $z = ((5^\ell - 1)/2 + 1/2) \cdot 10^{e_z-p+1}$. Since $(5^\ell - 1)/2$ is an integer, and by hypothesis $10^{p-1} \le (5^\ell - 1)/2 < 10^p$, we deduce that $z \in \mathbb{M}_{10,p}$, which concludes the proof. □

In radix 10, notice that there are at most two values of $\ell \in \mathbb{N}$ such that $2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^p$. Therefore, to determine all inputs y that give a midpoint $1/y$ for a fixed exponent e_y, it suffices to find the at most two ℓ such that $2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^p$. This is easily done, even when the precision p is large. For example, Table 3.9 gives the integral significands Y of the floating-point numbers y such that $1/y$ is a midpoint, for the decimal formats of the IEEE 754-2008 standard [27].

Table 3.9: Integral significands Y of $y \in \mathbb{F}_{10,p}$ such that $1/y \in \mathbb{M}_{10,p}$, for the decimal formats of the IEEE 754-2008 standard [27].

Format	Integral significand Y
decimal32 ($p = 7$)	$2^{14} \cdot 5^3 = 2048000$
decimal64 ($p = 16$)	$2^{32} \cdot 5^8 = 1677721600000000$ $2^{32} \cdot 5^9 = 8388608000000000$
decimal128 ($p = 34$)	$2^{68} \cdot 5^{18} = 1125899906842624000000000000000000$ $2^{68} \cdot 5^{19} = 5629499534213120000000000000000000$

3.6.2 Exact points for reciprocal

For radices either 10 or a positive power of 2, the exact points of the reciprocal function can all be enumerated according to the following theorem.

Theorem 3.16. *Let $y \in \mathbb{F}_{\beta,p}$ be nonzero. One has $1/y \in \mathbb{F}_{\beta,p}$ if and only if the integral significand Y of y satisfies $\beta^{p-1} \le Y < \beta^p$ and*

$$Y = \begin{cases} 2^k, & 0 \le k \le q(2p-1), \quad \text{if } \beta = 2^q,\ q \in \mathbb{N}_{>0}; \\ 2^k \cdot 5^\ell, & 0 \le k, \ell \le 2p-1, \quad \text{if } \beta = 10. \end{cases}$$

Proof. For the "only if" statement, let $y > 0$ in $\mathbb{F}_{\beta,p}$ be given, let $z = 1/y$, and assume that $z \in \mathbb{F}_{\beta,p}$. First, one may check that the exponent of z satisfies $e_z = -e_y - \delta$ with $\delta \in \{0, 1\}$. Then, using the identity $yz = 1$ together with the normalized representations $y = Y \cdot \beta^{e_y - p + 1}$ and $z = Z \cdot \beta^{e_z - p + 1}$, we get

$$YZ = \beta^{2p-2+\delta}, \qquad \beta^{p-1} \le Y, Z < \beta^p. \tag{3.19}$$

If $\beta = 2^q$ for some integer $q \ge 1$ then (3.19) implies that $Y = 2^k$ for some integer k such that $0 \le k \le q(2p-1)$. If $\beta = 10$ then (3.19) implies that $Y = 2^k \cdot 5^\ell$ for some integers k and ℓ such that $0 \le k, \ell \le 2p-1$.

Let us now prove the "if" statement. If $Y = \beta^{p-1}$ then y is a power of the radix and thus $1/y$ belongs to $\mathbb{F}_{\beta,p}$. If $\beta^{p-1} < Y < \beta^p$ then, defining $Z = Y^{-1} \cdot \beta^{2p-1}$, we obtain

$$1/y = Z \cdot \beta^{-e_y - p}, \qquad \beta^{p-1} < Z < \beta^p. \tag{3.20}$$

To conclude that $1/y$ belongs to $\mathbb{F}_{\beta,p}$ it remains to show that Z is an integer: If $\beta = 2^q$ and $Y = 2^k$, one has $Z = 2^{q(2p-1)-k}$, which is an integer for $k \le q(2p-1)$; If $\beta = 10$ and $Y = 2^k \cdot 5^\ell$ then $Z = 2^{2p-1-k} \cdot 5^{2p-1-\ell}$, which is an integer for $k, \ell \le 2p-1$. Hence Z is an integer in both cases, showing that $1/y$ is indeed an exact point. □

In radix $16 = 2^4$ for instance, the exact points $1/y$ with y in the interval $[1, 16)$ are listed below:

y	1	2	4	8
$1/y$	1	$1/2 = 0.8_{16}$	$1/4 = 0.4_{16}$	$1/8 = 0.2_{16}$

In radix 10, all the integers $Y = 2^k \cdot 5^\ell$ with $0 \leq k, \ell \leq 2p-1$ and $10^{p-1} \leq Y < 10^p$ can be enumerated by the simple Algorithm 2, and each one of them gives an exact point. Tables 3.10, 3.11, 3.12, and 3.13 give the integral significands Y such that $1/y$ is an exact point, in the case of the decimal32, decimal64 and decimal128 formats. Figures 3.5, 3.6 and 3.7 also display the distribution of the inputs that leads to exact points for the reciprocal function on the decade $[1, 10)$.

Data: Precision p
Result: Integral significands Y such that $y = Y \cdot 10^{e_y - p + 1}$ yields an exact point for the reciprocal

$\min_Y = 10^{p-1}$; $\max_Y = 10^p - 1$;
for k **from** 0 **to** $2p-1$ **do**
  $\ell_1 = \lceil \log_5 (\min_Y \cdot 2^{-k}) \rceil$; `/* At most two ℓ for each k */`
  $\ell_2 = \lfloor \log_5 (\max_Y \cdot 2^{-k}) \rfloor$;
  print $2^k \cdot 5^{\ell_1}$;
  if $\ell_1 \neq \ell_2$ **then** print $2^k \cdot 5^{\ell_2}$;
end

Algorithm 2: Enumerating reciprocal exactpoints

Furthermore, given an input exponent, the result below provides an explicit formula for the number N of floating-point inputs having this exponent and whose inverse if also a floating-point number:

Theorem 3.17. *For a given exponent e_y, the number N of positive values $y \in \mathbb{F}_{\beta,p}$ such that $1/y \in \mathbb{F}_{\beta,p}$ is given by*

$$N = \begin{cases} q, & \text{if } \beta = 2^q,\ q \in \mathbb{N}_{>0}; \\ 2\lfloor p \log_5(10) \rfloor + 1, & \text{if } \beta = 10. \end{cases}$$

Proof. When $\beta = 2^q$, Theorem 3.16 says that each exact point corresponds to an integer k such that $2^{q(p-1)} \leq 2^k < 2^{qp}$ and $0 \leq k \leq q(2p-1)$. The former condition is equivalent to $q(p-1) \leq k < qp$ and thus implies the latter. From this we deduce that the number of possible values of k is q when $\beta = 2^q$.

When $\beta = 10$, Theorem 3.16 says in this case that each exact point corresponds to a pair of integers (k, ℓ) such that

$$10^{p-1} \leq 2^k \cdot 5^\ell < 10^p \qquad \text{and} \qquad 0 \leq k, \ell \leq 2p-1.$$

The value of N is the number of points $(k, \ell) \in \mathbb{Z}^2$ that satisfy those two sets of constraints. Let $\sigma = \log_5(2) = 0.4306765581\ldots$. The first set of constraints is equivalent to

$$(p-1)(1+\sigma) \leq \sigma k + \ell < p(1+\sigma). \tag{3.21}$$

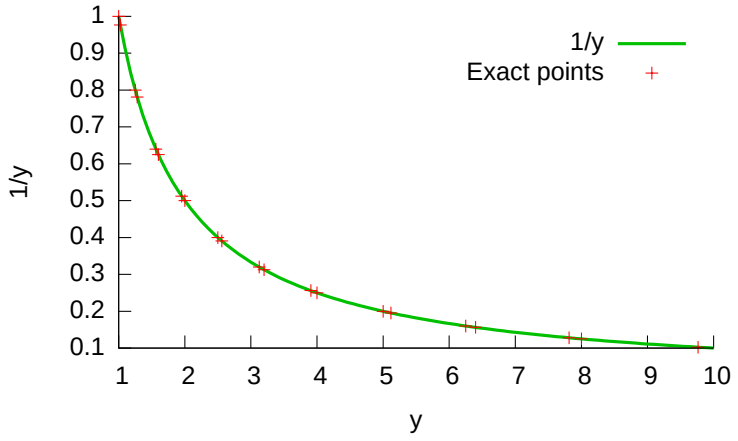


Figure 3.5: The exactpoints of the reciprocal function for the decimal32 format.

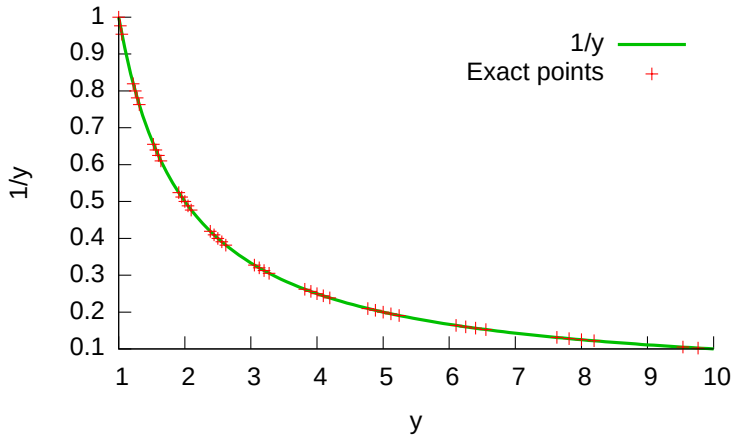


Figure 3.6: The exactpoints of the reciprocal function for the decimal64 format.

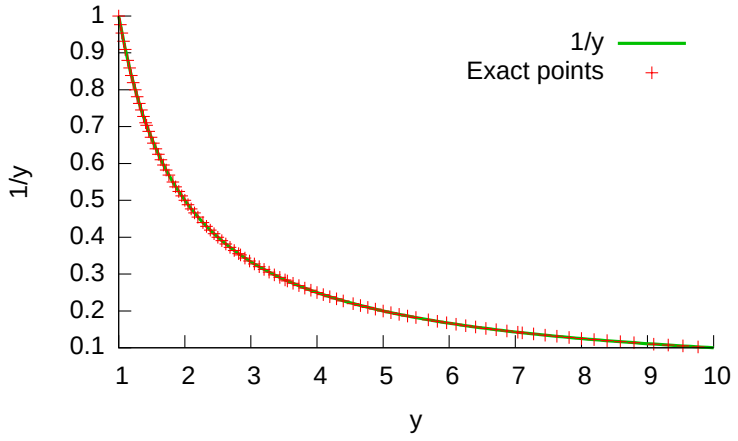


Figure 3.7: The exactpoints of the reciprocal function for the decimal128 format.

Table 3.10: Integral significands Y of $y \in \mathbb{F}_{10,7}$ such that $1/y \in \mathbb{F}_{10,7}$.

Y	$1/Y$
$2^0 \cdot 5^9 = 1953125$	$5.120000 \cdot 10^{-7}$
$2^0 \cdot 5^{10} = 9765625$	$1.024000 \cdot 10^{-7}$
$2^1 \cdot 5^9 = 3906250$	$2.560000 \cdot 10^{-7}$
$2^2 \cdot 5^8 = 1562500$	$6.400000 \cdot 10^{-7}$
$2^2 \cdot 5^9 = 7812500$	$1.280000 \cdot 10^{-7}$
$2^3 \cdot 5^8 = 3125000$	$3.200000 \cdot 10^{-7}$
$2^4 \cdot 5^7 = 1250000$	$8.000000 \cdot 10^{-7}$
$2^4 \cdot 5^8 = 6250000$	$1.600000 \cdot 10^{-7}$
$2^5 \cdot 5^7 = 2500000$	$4.000000 \cdot 10^{-7}$
$2^6 \cdot 5^6 = 1000000$	$1.000000 \cdot 10^{-6}$
$2^6 \cdot 5^7 = 5000000$	$2.000000 \cdot 10^{-7}$
$2^7 \cdot 5^6 = 2000000$	$5.000000 \cdot 10^{-7}$
$2^8 \cdot 5^6 = 4000000$	$2.500000 \cdot 10^{-7}$
$2^9 \cdot 5^5 = 1600000$	$6.250000 \cdot 10^{-7}$
$2^9 \cdot 5^6 = 8000000$	$1.250000 \cdot 10^{-7}$
$2^{10} \cdot 5^5 = 3200000$	$3.125000 \cdot 10^{-7}$
$2^{11} \cdot 5^4 = 1280000$	$7.812500 \cdot 10^{-7}$
$2^{11} \cdot 5^5 = 6400000$	$1.562500 \cdot 10^{-7}$
$2^{12} \cdot 5^4 = 2560000$	$3.906250 \cdot 10^{-7}$
$2^{13} \cdot 5^3 = 1024000$	$9.765625 \cdot 10^{-7}$
$2^{13} \cdot 5^4 = 5120000$	$1.953125 \cdot 10^{-7}$

Table 3.11: Integral significands Y of $y \in \mathbb{F}_{10,16}$ such that $1/y \in \mathbb{F}_{10,16}$.

Y	$1/Y$
$2^0 \cdot 5^{22} = 2384185791015625$	$4.194304000000000 \times 10^{-16}$
$2^1 \cdot 5^{22} = 4768371582031250$	$2.097152000000000 \times 10^{-16}$
$2^2 \cdot 5^{21} = 1907348632812500$	$5.242880000000000 \times 10^{-16}$
$2^2 \cdot 5^{22} = 9536743164062500$	$1.048576000000000 \times 10^{-16}$
$2^3 \cdot 5^{21} = 3814697265625000$	$2.621440000000000 \times 10^{-16}$
$2^4 \cdot 5^{20} = 1525878906250000$	$6.553600000000000 \times 10^{-16}$
$2^4 \cdot 5^{21} = 7629394531250000$	$1.310720000000000 \times 10^{-16}$
$2^5 \cdot 5^{20} = 3051757812500000$	$3.276800000000000 \times 10^{-16}$
$2^6 \cdot 5^{19} = 1220703125000000$	$8.192000000000000 \times 10^{-16}$
$2^6 \cdot 5^{20} = 6103515625000000$	$1.638400000000000 \times 10^{-16}$
$2^7 \cdot 5^{19} = 2441406250000000$	$4.096000000000000 \times 10^{-16}$
$2^8 \cdot 5^{19} = 4882812500000000$	$2.048000000000000 \times 10^{-16}$
$2^9 \cdot 5^{18} = 1953125000000000$	$5.120000000000000 \times 10^{-16}$
$2^9 \cdot 5^{19} = 9765625000000000$	$1.024000000000000 \times 10^{-16}$
$2^{10} \cdot 5^{18} = 3906250000000000$	$2.560000000000000 \times 10^{-16}$
$2^{11} \cdot 5^{17} = 1562500000000000$	$6.400000000000000 \times 10^{-16}$
$2^{11} \cdot 5^{18} = 7812500000000000$	$1.280000000000000 \times 10^{-16}$
$2^{12} \cdot 5^{17} = 3125000000000000$	$3.200000000000000 \times 10^{-16}$
$2^{13} \cdot 5^{16} = 1250000000000000$	$8.000000000000000 \times 10^{-16}$
$2^{13} \cdot 5^{17} = 6250000000000000$	$1.600000000000000 \times 10^{-16}$
$2^{14} \cdot 5^{16} = 2500000000000000$	$4.000000000000000 \times 10^{-16}$
$2^{15} \cdot 5^{15} = 1000000000000000$	$1.000000000000000 \times 10^{-15}$
$2^{15} \cdot 5^{16} = 5000000000000000$	$2.000000000000000 \times 10^{-16}$
$2^{16} \cdot 5^{15} = 2000000000000000$	$5.000000000000000 \times 10^{-16}$
$2^{17} \cdot 5^{15} = 4000000000000000$	$2.500000000000000 \times 10^{-16}$
$2^{18} \cdot 5^{14} = 1600000000000000$	$6.250000000000000 \times 10^{-16}$
$2^{18} \cdot 5^{15} = 8000000000000000$	$1.250000000000000 \times 10^{-16}$
$2^{19} \cdot 5^{14} = 3200000000000000$	$3.125000000000000 \times 10^{-16}$
$2^{20} \cdot 5^{13} = 1280000000000000$	$7.812500000000000 \times 10^{-16}$
$2^{20} \cdot 5^{14} = 6400000000000000$	$1.562500000000000 \times 10^{-16}$
$2^{21} \cdot 5^{13} = 2560000000000000$	$3.906250000000000 \times 10^{-16}$
$2^{22} \cdot 5^{12} = 1024000000000000$	$9.765625000000000 \times 10^{-16}$
$2^{22} \cdot 5^{13} = 5120000000000000$	$1.953125000000000 \times 10^{-16}$
$2^{23} \cdot 5^{12} = 2048000000000000$	$4.882812500000000 \times 10^{-16}$
$2^{24} \cdot 5^{12} = 4096000000000000$	$2.441406250000000 \times 10^{-16}$
$2^{25} \cdot 5^{11} = 1638400000000000$	$6.103515625000000 \times 10^{-16}$
$2^{25} \cdot 5^{12} = 8192000000000000$	$1.220703125000000 \times 10^{-16}$
$2^{26} \cdot 5^{11} = 3276800000000000$	$3.051757812500000 \times 10^{-16}$
$2^{27} \cdot 5^{10} = 1310720000000000$	$7.629394531250000 \times 10^{-16}$
$2^{27} \cdot 5^{11} = 6553600000000000$	$1.525878906250000 \times 10^{-16}$
$2^{28} \cdot 5^{10} = 2621440000000000$	$3.814697265625000 \times 10^{-16}$
$2^{29} \cdot 5^{9} = 1048576000000000$	$9.536743164062500 \times 10^{-16}$
$2^{29} \cdot 5^{10} = 5242880000000000$	$1.907348632812500 \times 10^{-16}$
$2^{30} \cdot 5^{9} = 2097152000000000$	$4.768371582031250 \times 10^{-16}$
$2^{31} \cdot 5^{9} = 4194304000000000$	$2.384185791015625 \times 10^{-16}$

Table 3.12: Integral significands Y of $y \in \mathbb{F}_{10,34}$ such that $1/y \in \mathbb{F}_{10,34}$.

Y	$1/Y$
$2^0 \cdot 5^{48} = 355271367880050092935621337890625$	$2.8147497671065600000000000000000000 \times 10^{-34}$
$2^1 \cdot 5^{47} = 142108547152020037174224853515625 0$	$7.0368744177664000000000000000000000 \times 10^{-34}$
$2^1 \cdot 5^{48} = 710542735760100185871124267578125 0$	$1.4073748835532800000000000000000000 \times 10^{-34}$
$2^2 \cdot 5^{47} = 284217094304040074348449707031250 0$	$3.5184372088832000000000000000000000 \times 10^{-34}$
$2^3 \cdot 5^{46} = 113686837721616029739379882812500 0$	$8.7960930222080000000000000000000000 \times 10^{-34}$
$2^3 \cdot 5^{47} = 568434188608080148696899414062500 0$	$1.7592186044416000000000000000000000 \times 10^{-34}$
$2^4 \cdot 5^{46} = 227373675443232059478759765625000 0$	$4.3980465111040000000000000000000000 \times 10^{-34}$
$2^5 \cdot 5^{46} = 454747350886464118957519531250000 0$	$2.1990232555520000000000000000000000 \times 10^{-34}$
$2^6 \cdot 5^{45} = 181898940354585647583007812500000 0$	$5.4975581388800000000000000000000000 \times 10^{-34}$
$2^6 \cdot 5^{46} = 909494701772928237915039062500000 0$	$1.0995116277760000000000000000000000 \times 10^{-34}$
$2^7 \cdot 5^{45} = 363797880709171295166015625000000 0$	$2.7487790694400000000000000000000000 \times 10^{-34}$
$2^8 \cdot 5^{44} = 145519152283668518066406250000000 0$	$6.8719476736000000000000000000000000 \times 10^{-34}$
$2^8 \cdot 5^{45} = 727595761418342590332031250000000 0$	$1.3743895347200000000000000000000000 \times 10^{-34}$
$2^9 \cdot 5^{44} = 291038304567337036132812500000000 0$	$3.4359738368000000000000000000000000 \times 10^{-34}$
$2^{10} \cdot 5^{43} = 116415321826934814453125000000000 0$	$8.5899345920000000000000000000000000 \times 10^{-34}$
$2^{10} \cdot 5^{44} = 582076609134674072265625000000000 0$	$1.7179869184000000000000000000000000 \times 10^{-34}$
$2^{11} \cdot 5^{43} = 232830643653869628906250000000000 0$	$4.2949672960000000000000000000000000 \times 10^{-34}$
$2^{12} \cdot 5^{43} = 465661287307739257812500000000000 0$	$2.1474836480000000000000000000000000 \times 10^{-34}$
$2^{13} \cdot 5^{42} = 186264514923095703125000000000000 0$	$5.3687091200000000000000000000000000 \times 10^{-34}$
$2^{13} \cdot 5^{43} = 931322574615478515625000000000000 0$	$1.0737418240000000000000000000000000 \times 10^{-34}$
$2^{14} \cdot 5^{42} = 372529029846191406250000000000000 0$	$2.6843545600000000000000000000000000 \times 10^{-34}$
$2^{15} \cdot 5^{41} = 149011611938476562500000000000000 0$	$6.7108864000000000000000000000000000 \times 10^{-34}$
$2^{15} \cdot 5^{42} = 745058059692382812500000000000000 0$	$1.3421772800000000000000000000000000 \times 10^{-34}$
$2^{16} \cdot 5^{41} = 298023223876953125000000000000000 0$	$3.3554432000000000000000000000000000 \times 10^{-34}$
$2^{17} \cdot 5^{40} = 119209289550781250000000000000000 0$	$8.3886080000000000000000000000000000 \times 10^{-34}$
$2^{17} \cdot 5^{41} = 596046447753906250000000000000000 0$	$1.6777216000000000000000000000000000 \times 10^{-34}$
$2^{18} \cdot 5^{40} = 238418579101562500000000000000000 0$	$4.1943040000000000000000000000000000 \times 10^{-34}$
$2^{19} \cdot 5^{40} = 476837158203125000000000000000000 0$	$2.0971520000000000000000000000000000 \times 10^{-34}$
$2^{20} \cdot 5^{39} = 190734863281250000000000000000000 0$	$5.2428800000000000000000000000000000 \times 10^{-34}$
$2^{20} \cdot 5^{40} = 953674316406250000000000000000000 0$	$1.0485760000000000000000000000000000 \times 10^{-34}$
$2^{21} \cdot 5^{39} = 381469726562500000000000000000000 0$	$2.6214400000000000000000000000000000 \times 10^{-34}$
$2^{22} \cdot 5^{38} = 152587890625000000000000000000000 0$	$6.5536000000000000000000000000000000 \times 10^{-34}$
$2^{22} \cdot 5^{39} = 762939453125000000000000000000000 0$	$1.3107200000000000000000000000000000 \times 10^{-34}$
$2^{23} \cdot 5^{38} = 305175781250000000000000000000000 0$	$3.2768000000000000000000000000000000 \times 10^{-34}$
$2^{24} \cdot 5^{37} = 122070312500000000000000000000000 0$	$8.1920000000000000000000000000000000 \times 10^{-34}$
$2^{24} \cdot 5^{38} = 610351562500000000000000000000000 0$	$1.6384000000000000000000000000000000 \times 10^{-34}$
$2^{25} \cdot 5^{37} = 244140625000000000000000000000000 0$	$4.0960000000000000000000000000000000 \times 10^{-34}$
$2^{26} \cdot 5^{37} = 488281250000000000000000000000000 0$	$2.0480000000000000000000000000000000 \times 10^{-34}$
$2^{27} \cdot 5^{36} = 195312500000000000000000000000000 0$	$5.1200000000000000000000000000000000 \times 10^{-34}$
$2^{27} \cdot 5^{37} = 976562500000000000000000000000000 0$	$1.0240000000000000000000000000000000 \times 10^{-34}$
$2^{28} \cdot 5^{36} = 390625000000000000000000000000000 0$	$2.5600000000000000000000000000000000 \times 10^{-34}$
$2^{29} \cdot 5^{35} = 156250000000000000000000000000000 0$	$6.4000000000000000000000000000000000 \times 10^{-34}$
$2^{29} \cdot 5^{36} = 781250000000000000000000000000000 0$	$1.2800000000000000000000000000000000 \times 10^{-34}$
$2^{30} \cdot 5^{35} = 312500000000000000000000000000000 0$	$3.2000000000000000000000000000000000 \times 10^{-34}$
$2^{31} \cdot 5^{34} = 125000000000000000000000000000000 0$	$8.0000000000000000000000000000000000 \times 10^{-34}$
$2^{31} \cdot 5^{35} = 625000000000000000000000000000000 0$	$1.6000000000000000000000000000000000 \times 10^{-34}$
$2^{32} \cdot 5^{34} = 250000000000000000000000000000000 0$	$4.0000000000000000000000000000000000 \times 10^{-34}$
$2^{33} \cdot 5^{33} = 100000000000000000000000000000000 0$	$1.0000000000000000000000000000000000 \times 10^{-33}$

Table 3.13: Integral significands Y of $y \in \mathbb{F}_{10,34}$ such that $1/y \in \mathbb{F}_{10,34}$.

Y	$1/Y$
$2^{33}\cdot5^{34} = 5000000000000000000000000000000000$	$2.000000000000000000000000000000000 \times 10^{-34}$
$2^{34}\cdot5^{33} = 2000000000000000000000000000000000$	$5.000000000000000000000000000000000 \times 10^{-34}$
$2^{35}\cdot5^{33} = 4000000000000000000000000000000000$	$2.500000000000000000000000000000000 \times 10^{-34}$
$2^{36}\cdot5^{32} = 1600000000000000000000000000000000$	$6.250000000000000000000000000000000 \times 10^{-34}$
$2^{36}\cdot5^{33} = 8000000000000000000000000000000000$	$1.250000000000000000000000000000000 \times 10^{-34}$
$2^{37}\cdot5^{32} = 3200000000000000000000000000000000$	$3.125000000000000000000000000000000 \times 10^{-34}$
$2^{38}\cdot5^{31} = 1280000000000000000000000000000000$	$7.812500000000000000000000000000000 \times 10^{-34}$
$2^{38}\cdot5^{32} = 6400000000000000000000000000000000$	$1.562500000000000000000000000000000 \times 10^{-34}$
$2^{39}\cdot5^{31} = 2560000000000000000000000000000000$	$3.906250000000000000000000000000000 \times 10^{-34}$
$2^{40}\cdot5^{30} = 1024000000000000000000000000000000$	$9.765625000000000000000000000000000 \times 10^{-34}$
$2^{40}\cdot5^{31} = 5120000000000000000000000000000000$	$1.953125000000000000000000000000000 \times 10^{-34}$
$2^{41}\cdot5^{30} = 2048000000000000000000000000000000$	$4.882812500000000000000000000000000 \times 10^{-34}$
$2^{42}\cdot5^{30} = 4096000000000000000000000000000000$	$2.441406250000000000000000000000000 \times 10^{-34}$
$2^{43}\cdot5^{29} = 1638400000000000000000000000000000$	$6.103515625000000000000000000000000 \times 10^{-34}$
$2^{43}\cdot5^{30} = 8192000000000000000000000000000000$	$1.220703125000000000000000000000000 \times 10^{-34}$
$2^{44}\cdot5^{29} = 3276800000000000000000000000000000$	$3.051757812500000000000000000000000 \times 10^{-34}$
$2^{45}\cdot5^{28} = 1310720000000000000000000000000000$	$7.629394531250000000000000000000000 \times 10^{-34}$
$2^{45}\cdot5^{29} = 6553600000000000000000000000000000$	$1.525878906250000000000000000000000 \times 10^{-34}$
$2^{46}\cdot5^{28} = 2621440000000000000000000000000000$	$3.814697265625000000000000000000000 \times 10^{-34}$
$2^{47}\cdot5^{27} = 1048576000000000000000000000000000$	$9.536743164062500000000000000000000 \times 10^{-34}$
$2^{47}\cdot5^{28} = 5242880000000000000000000000000000$	$1.907348632812500000000000000000000 \times 10^{-34}$
$2^{48}\cdot5^{27} = 2097152000000000000000000000000000$	$4.768371582031250000000000000000000 \times 10^{-34}$
$2^{49}\cdot5^{27} = 4194304000000000000000000000000000$	$2.384185791015625000000000000000000 \times 10^{-34}$
$2^{50}\cdot5^{26} = 1677721600000000000000000000000000$	$5.960464477539062500000000000000000 \times 10^{-34}$
$2^{50}\cdot5^{27} = 8388608000000000000000000000000000$	$1.192092895507812500000000000000000 \times 10^{-34}$
$2^{51}\cdot5^{26} = 3355443200000000000000000000000000$	$2.980232238769531250000000000000000 \times 10^{-34}$
$2^{52}\cdot5^{25} = 1342177280000000000000000000000000$	$7.450580596923828125000000000000000 \times 10^{-34}$
$2^{52}\cdot5^{26} = 6710886400000000000000000000000000$	$1.490116119384765625000000000000000 \times 10^{-34}$
$2^{53}\cdot5^{25} = 2684354560000000000000000000000000$	$3.725290298461914062500000000000000 \times 10^{-34}$
$2^{54}\cdot5^{24} = 1073741824000000000000000000000000$	$9.313225746154785156250000000000000 \times 10^{-34}$
$2^{54}\cdot5^{25} = 5368709120000000000000000000000000$	$1.862645149230957031250000000000000 \times 10^{-34}$
$2^{55}\cdot5^{24} = 2147483648000000000000000000000000$	$4.656612873077392578125000000000000 \times 10^{-34}$
$2^{56}\cdot5^{24} = 4294967296000000000000000000000000$	$2.328306436538696289062500000000000 \times 10^{-34}$
$2^{57}\cdot5^{23} = 1717986918400000000000000000000000$	$5.820766091346740722656250000000000 \times 10^{-34}$
$2^{57}\cdot5^{24} = 8589934592000000000000000000000000$	$1.164153218269348144531250000000000 \times 10^{-34}$
$2^{58}\cdot5^{23} = 3435973836800000000000000000000000$	$2.910383045673370361328125000000000 \times 10^{-34}$
$2^{59}\cdot5^{22} = 1374389534720000000000000000000000$	$7.275957614183425903320312500000000 \times 10^{-34}$
$2^{59}\cdot5^{23} = 6871947673600000000000000000000000$	$1.455191522836685180664062500000000 \times 10^{-34}$
$2^{60}\cdot5^{22} = 2748779069440000000000000000000000$	$3.637978807091712951660156250000000 \times 10^{-34}$
$2^{61}\cdot5^{21} = 1099511627776000000000000000000000$	$9.094947017729282379150390625000000 \times 10^{-34}$
$2^{61}\cdot5^{22} = 5497558138880000000000000000000000$	$1.818989403545856475830078125000000 \times 10^{-34}$
$2^{62}\cdot5^{21} = 2199023255552000000000000000000000$	$4.547473508864641189575195312500000 \times 10^{-34}$
$2^{63}\cdot5^{21} = 4398046511104000000000000000000000$	$2.273736754432320594787597656250000 \times 10^{-34}$
$2^{64}\cdot5^{20} = 1759218604441600000000000000000000$	$5.684341886080801486968994140625000 \times 10^{-34}$
$2^{64}\cdot5^{21} = 8796093022208000000000000000000000$	$1.136868377216160297393798828125000 \times 10^{-34}$
$2^{65}\cdot5^{20} = 3518437208883200000000000000000000$	$2.842170943040400743484497070312500 \times 10^{-34}$
$2^{66}\cdot5^{19} = 1407374883553280000000000000000000$	$7.105427357601001858711242675781250 \times 10^{-34}$
$2^{66}\cdot5^{20} = 7036874417766400000000000000000000$	$1.421085471520200371742248535156250 \times 10^{-34}$
$2^{67}\cdot5^{19} = 2814749767106560000000000000000000$	$3.552713678800500929355621337890625 \times 10^{-34}$

It implies in particular that $(p-1)(1+\sigma) \leq \ell < p(1+\sigma)$, which is stronger than $0 \leq \ell \leq 2p-1$ for $p \geq 2$, since $1+\sigma \approx 1.43$. Hence $N = \sum_{0 \leq k < 2p} N_k$, where N_k is the number of integers ℓ satisfying (3.21) for a given k.

Recalling that half-open real intervals $[a, b)$ such that $a \leq b$ contain exactly $\lceil b \rceil - \lceil a \rceil$ integers [20, p. 74], we deduce that, for $0 \leq k < 2p$,

$$\begin{aligned} N_k &= \big\lceil p(1+\sigma) - \sigma k \big\rceil - \big\lceil (p-1)(1+\sigma) - \sigma k \big\rceil \\ &= \big\lceil (p-k)\sigma \big\rceil - \big\lceil (p-k-1)\sigma \big\rceil + 1. \end{aligned}$$

Consequently, the sum $\sum_{0 \leq k < 2p} N_k$ telescopes to $2p + \lfloor p\sigma \rfloor + \lceil p\sigma \rceil$. Since the integer p is nonzero and σ is irrational, $p\sigma$ cannot be an integer. Hence $\lceil p\sigma \rceil = \lfloor p\sigma \rfloor + 1$, which leads to $N = 2\big\lfloor p(1+\sigma) \big\rfloor + 1$. □

According to Theorem 3.17, when $\beta = 2^q$, the number N of different integral significands leading to an exact point is simply q. In radix 10, we have $N = \Theta(p)$, which confirms the fact that the midpoints for the reciprocal can be easily enumerated, even when the precision p is large. In particular, this is in contrast with the exact points of square root in radix 10 or 2^q, whose number was seen to be exponential in p in §3.1.2. For the decimal formats of the IEEE 754-2008, the corresponding values of N are listed below:

Format	decimal32	decimal64	decimal128
p	7	16	34
N	21	45	97

3.7 Reciprocal 2D Euclidean norm

Given a d-dimensional vector y with entries in $\mathbb{F}_{2,p}$, we know from Theorem 3.12 that $z = 1/\|y\|_2$ cannot be a midpoint in radix 2. In this section, we focus on the two-dimensional case, studying the midpoints and the exact points of the reciprocal 2D Euclidean norm, in radices 2^q and 10. In radix 10, our study relies on the representation of products of the form $2^r \cdot 5^s$ as sums of two squares $a^2 + b^2$, where $a, b \in \mathbb{N}$. Thus, we first explain in §3.7.1 the method we used for enumerating all the representations of such a product as the sum of two integer squares. Then midpoints and exact points are studied in §§3.7.2 and 3.7.3, respectively.

3.7.1 Decomposing $2^r \cdot 5^s$ into sums of two squares

Decomposing an integer into sums of two squares is a well studied problem in the mathematical literature (see for instance Wagon [56] and the references therein). In our particular case of interest, we deduce the following theorem that allows to compute all decompositions of $2^r \cdot 5^s$ as sums of two squares. The proof of Theorem 3.18 relies on the unicity of the decomposition of a number into prime factors in the ring of Gaussian integers $\mathbb{Z}[i]$ (see for instance Everest and Ward [15, chap. 2] for more details on this topic).

Theorem 3.18. *Let $r, s \in \mathbb{N}$ be given, and assume $k \in \mathbb{N}$. All the unordered pairs $\{a, b\}$ with $a, b \in \mathbb{N}$ and $a^2 + b^2 = 2^r \cdot 5^s$ are given by $a = |\Re(c)|$ and $b = |\Im(c)|$ with*

$$c = 2^{\lfloor r/2 \rfloor} (1+i)^{r \bmod 2} (2+i)^k (2-i)^{s-k}, \qquad 0 \leq k < \lceil (s+1)/2 \rceil.$$

In particular, there are $\lceil (s+1)/2 \rceil$ *different decompositions of* $2^r \cdot 5^s$ *as the sum of two squares.*

Proof. Let us assume $2^r \cdot 5^s = a^2 + b^2$. Since the decomposition of $2^r \cdot 5^s$ into prime factors in $\mathbb{Z}[i]$ is unique apart from multiplications by ± 1 or $\pm i$, and since $2 = (1+i)(1-i)$, $5 = (2+i)(2-i)$, one has

$$2^r \cdot 5^s = (1+i)^r (1-i)^r (2+i)^s (2-i)^s,$$

On the other hand one has $a^2 + b^2 = (a+ib)(a-ib)$, hence by unicity of the decomposition into prime factors it follows that

$$a + ib = \delta_0 (1+i)^{k_1} (1-i)^{k_2} (2+i)^{k_3} (2-i)^{k_4}$$

for some $k_1, k_2, k_3, k_4 \in \mathbb{N}$ and $\delta_0 \in \{\pm 1, \pm i\}$. Then one has

$$a^2 + b^2 = \delta_0 \overline{\delta_0} (1+i)^{k_1+k_2} (1-i)^{k_1+k_2} (2+i)^{k_3+k_4} (2-i)^{k_3+k_4},$$

and from $2^r \cdot 5^s = a^2 + b^2$ we deduce that $k_1 + k_2 = r$ and $k_3 + k_4 = s$, hence

$$a + ib = \delta_0 (1+i)^{k_1} (1-i)^{r-k_1} (2+i)^{k_3} (2-i)^{s-k_3}$$

Moreover, since $k_1 + k_2 = r$, it can be noticed that

$$(1+i)^{k_1} (1-i)^{r-k_1} = \delta_1 \times \begin{cases} 2^{(r-1)/2}(1+i), & \text{if } r \text{ is odd,} \\ 2^{r/2}, & \text{otherwise,} \end{cases}$$

with $\delta_1 \in \{\pm 1, \pm i\}$. This can be rewritten $(1+i)^{k_1}(1-i)^{r-k_1} = \delta_1 \cdot 2^{\lfloor r/2 \rfloor} (1+i)^{r \bmod 2}$. Hence we obtain

$$a + ib = \delta \cdot 2^{\lfloor r/2 \rfloor} (1+i)^{r \bmod 2} (2+i)^k (2-i)^{s-k},$$

for some $\delta \in \{\pm 1, \pm i\}$ and $k \in \mathbb{N}$ such that $0 \le k \le s$.

Since $a, b \ge 0$, we deduce that necessarily $a = |\Re(c)|$ and $b = |\Im(c)|$ with

$$c = 2^{\lfloor r/2 \rfloor} (1+i)^{r \bmod 2} (2+i)^k (2-i)^{s-k}.$$

However, since both c and $\overline{c} = 2^{\lfloor r/2 \rfloor} (1-i)^{r \bmod 2} (2-i)^k (2+i)^{s-k}$ lead to the same unordered pair $\{a, b\}$, there are at most $\lceil (s+1)/2 \rceil$ such unordered pairs $\{a, b\}$. This implies that we only need the assumption $0 \le k < \lceil (s+1)/2 \rceil$ for k.

Conversely, if $a = |\Re(c)|$ and $b = |\Im(c)|$ with $c = 2^{\lfloor r/2 \rfloor} (1+i)^{r \bmod 2} (2+i)^k (2-i)^{s-k}$, then $a + ib = \delta 2^{\lfloor r/2 \rfloor} (1+i)^{r \bmod 2} (2+i)^k (2-i)^{s-k}$ with $\delta \in \{\pm 1, \pm i\}$. Then one can easily check that $a^2 + b^2 = (a+ib)(a-ib) = 2^r \cdot 5^s$.

By unicity of the factorization into primes in $\mathbb{Z}[i]$, it can be shown that if we take $k_1 \neq k_2$ with $0 \le k_1, k_2 < \lceil (s+1)/2 \rceil$, then the corresponding unordered pairs $\{a_1, b_1\}$ and $\{a_2, b_2\}$ are necessarily different. It means that there are exactly $\lceil (s+1)/2 \rceil$ unordered pairs $\{a, b\}$. □

For later use, we also state the following corollary of Theorem 3.18 for the decomposition of 2^r.

Corollary 3.19. *Given* $r \in \mathbb{N}$*, the unique decomposition of* 2^r *as a sum of two integer squares is*

$$2^r = \begin{cases} 0^2 + (2^{r/2})^2, & \textit{if } r \textit{ is even,} \\ (2^{(r-1)/2})^2 + (2^{(r-1)/2})^2, & \textit{if } r \textit{ is odd.} \end{cases}$$

3.7.2 Midpoints for reciprocal 2D norm

Theorem 3.20 below can be used to determine all the midpoints of the reciprocal 2D-norm function with a given exponent e_z.

Theorem 3.20. *Let $x, y \in \mathbb{F}_{\beta,p}$ be such that $(x, y) \neq (0, 0)$, and let $z = 1/\sqrt{x^2 + y^2}$. If $\beta = 2^q$ ($q \in \mathbb{N}_{>0}$) then $z \notin \mathbb{M}_{\beta,p}$. If $\beta = 10$, one has $z \in \mathbb{M}_{\beta,p}$ if and only if z has the form*

$$\left(\frac{5^\ell - 1}{2} + \frac{1}{2}\right) \cdot 10^{e_z - p + 1},$$

with $e_z \in \mathbb{Z}$, and $\ell \in \mathbb{N}$ such that $2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^p$.

Proof of Theorem 3.20. Let $z = 1/\sqrt{x^2 + y^2}$ be a midpoint, with $x, y \in \mathbb{F}_{\beta,p}$. Without loss of generality, we assume that z is in $[1, \beta)$, and since z is a midpoint then one has $1 < z < \beta$. Let us also assume that $x \geq y \geq 0$, which implies

$$\frac{1}{\sqrt{2}x} \leq \frac{1}{\sqrt{x^2 + y^2}} \leq \frac{1}{x}. \tag{3.22}$$

Denoting by e_x and e_y the exponents of x and y respectively, from (3.22) it follows that $\beta^{-e_x-2} < z \leq \beta^{-e_x}$, and since $1 < z < \beta$, necessarily $e_x \in \{-1, -2\}$. Writing $z = (Z + 1/2) \cdot \beta^{-p+1}$, with $Z \in \mathbb{N}$ such that $\beta^{p-1} \leq Z < \beta^p$, from $(x^2 + y^2)z^2 = 1$ we deduce

$$\left(X^2 \cdot \beta^{2e_x - 2e_y} + Y^2\right)(2Z + 1)^2 = 4 \cdot \beta^{4p - 2e_y - 4}. \tag{3.23}$$

Note that $x \geq y$ implies $e_x \geq e_y$, so that the left-hand side of Equation (3.23) is indeed in $\mathbb{N}$. When $\beta = 2^q$, Equation (3.23) has no solution, since the right-hand side of the equality is a power of two while the left-hand side has an odd factor $(2Z + 1)^2$.

When $\beta = 10$, Equation (3.23) becomes

$$\left(X^2 \cdot 10^{2e_x - 2e_y} + Y^2\right)(2Z + 1)^2 = 2^{4p - 2e_y - 2} \cdot 5^{4p - 2e_y - 4}. \tag{3.24}$$

Then one has necessarily $2Z + 1 = 5^\ell$ with $\ell \in \mathbb{N}$. The bounds on 5^ℓ follow from $10^{p-1} \leq Z \leq 10^p - 1$. Conversely, if z has the form given in Theorem 3.20 it is clearly a midpoint. □

For instance, in the decimal32 format of the IEEE 754-2008 ($p = 7$), the function $1/\sqrt{x^2 + y^2}$ has only one midpoint in the decade $[1, 10)$, namely $z = 4.8828125$. This midpoint corresponds to $5^{10} = 9765625$, which is the only power of 5 in the interval $(2 \cdot 10^6, 2 \cdot 10^7)$. All the other midpoints of the function are obtained by multiplying 4.8828125 by an integral power of 10.

Theorem 3.20 can only be used to determine the midpoints of the reciprocal norm function. Given such a midpoint z, let us now show how to find x and y in $\mathbb{F}_{10,p}$ such that $z = 1/\sqrt{x^2 + y^2}$. For this, we shall use the following lemma.

Lemma 3.21. *Let a be in $\mathbb{Q}$. One has $a^2 \in \mathbb{N}$ if and only if $a \in \mathbb{Z}$.*

Proof. Assume that $a = p/q$, with $p, q \in \mathbb{Z}$, $q \neq 0$, and $\gcd(p, q) = 1$. Assume also that $a^2 = n \in \mathbb{N}$. Then $p^2 = nq^2$, hence $q^2 | p^2$, which in turn implies $q | p$. Since by assumption $\gcd(p, q) = 1$, then one has $q = \pm 1$, hence necessarily $a \in \mathbb{Z}$. Conversely, if $a \in \mathbb{Z}$ then obviously $a^2 \in \mathbb{N}$. □

As in the proof of Theorem 3.20, let us assume that $1 < z < 10$ and $x \geq y \geq 0$, which implies $e_x \in \{-1, -2\}$. We denote by X and Y the integral significands of x and y respectively. From Equation (3.24) we can deduce that X and Y must satisfy

$$2^{4p+2} \cdot 5^{4p-2\ell} = \left(X \cdot 10^{e_x+2}\right)^2 + \left(Y \cdot 10^{e_y+2}\right)^2. \tag{3.25}$$

From $2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^p$, one has $5^{4p-2\ell} \in \mathbb{N}$. Since moreover $e_x \in \{-1, -2\}$, necessarily $X \cdot 10^{e_x+2} \in \mathbb{N}$, and $Y^2 \cdot 10^{2(e_y+2)}$ is also in $\mathbb{N}$. Since $Y \cdot 10^{e_y+2}$ is a nonnegative rational number whose square is a natural integer, it follows from Lemma 3.21 that $Y \cdot 10^{e_y+2} \in \mathbb{N}$. Hence we know that $X \cdot 10^{e_x+2}$ and $Y \cdot 10^{e_y+2}$ both necessarily belong to $\mathbb{N}$.

As a consequence, to find all inputs (X, Y) that give a midpoint for the function $1/\sqrt{x^2+y^2}$, we know from Equation (3.25) that we need to find all the decompositions of the at most two integers $2^{4p+2} \cdot 5^{4p-2\ell}$ as the sum of two squares. We use Algorithm 3 to build all values x and y, $x \geq y$, such that $1/\sqrt{x^2+y^2}$ is a midpoint, for the decimal formats of the IEEE 754-2008 standard. For the decimal32 format, all the pairs of floating-point numbers (x, y) for which $1/\sqrt{x^2+y^2}$ is a midpoint can be deduced from the pairs listed in Table 3.14 by either exchanging x and y or by multiplying them by the same power of 10. Tables 3.15 and 3.16 give similar results for the decimal64 and decimal128 formats.

Data: Precision p
Result: Couples $x, y; x \geq y$ such that $1/\sqrt{x^2+y^2}$ is a midpoint in the decade $[10^{-1}; 1)$

num = 0;
$\ell_1 = \lceil \log_5(2 \cdot 10^{p-1} + 1) \rceil$;
$\ell_2 = \lfloor \log_5(2 \cdot 10^p - 1) \rfloor$;
`/* Decompose` $2^{4p+2} \cdot 5^{4p-2\ell_1} = a^2 + b^2$ `into two lists */`
$ta_1, tb_1 = \text{decomp25}(4p+2, 4p-2i\ell_1)$;
if $\ell_1 \neq \ell_2$ **then**
    $ta_2, tb_2 = \text{decomp25}(4p+2, 4p-2\ell_2)$; `/* Decompose` $2^{4p+2} \cdot 5^{4p-2\ell_2} = a^2 + b^2$ `*/`
    $ta_1 = ta_1 @ ta_2$; `/* Concatenation of lists */`
    $tb_1 = tb_1 @ tb_2$;
end
for i **from** 1 **to** SizeOf(ta_1) **do**
    if $ta_1[i]$ *and* $tb_1[i]$ *fit on* p *digits* **then**
        print $(ta_1[i], tb_1[i])$; `/* Non-normalized output */`
    end
end

Algorithm 3: Determining the midpoints for the reciprocal 2Dnorm function

Table 3.17 gives the number N_z of midpoints z in a decade (i.e., with a fixed exponent e_z), with respect to the decimal format considered. The table also gives the number N of pairs of integral significand (X, Y) with $X \geq Y$ that give these midpoints. In decimal64 arithmetic for instance, the function $(x, y) \mapsto 1/\sqrt{x^2+y^2}$ has 2 midpoints $z_1 < z_2$ in the decade $[1, 10)$: the number of pairs (X, Y) that give z_1 is 10, and 9 pairs give z_2.

Table 3.14: Floating-point numbers $x, y \in \mathbb{F}_{10,7}$ with $X \geq Y$ such that $z = 1/\sqrt{x^2+y^2}$ is a midpoint, with $10^{-8} \leq z < 10^{-7}$.

x	y	$z = 1/\sqrt{x^2+y^2}$
1966080	573440.0	4.8828125×10^{-7}
1638400	1228800	
1916928	720896.0	
2048000	0	

Table 3.15: Floating-point numbers $x, y \in \mathbb{F}_{10,16}$ with $X \geq Y$ such that $z = 1/\sqrt{x^2+y^2}$ is a midpoint.

$$z_1 = 1.1920928955078125 \cdot 10^{-16}$$
$$z_2 = 5.9604644775390625 \cdot 10^{-16}$$

x	y	z
8053063680000000	2348810240000000	z_1
6710886400000000	5033164800000000	
7851737088000000	2952790016000000	
7073274265600000	4509715660800000	
6309843828736000	5527622909952000	
8208004625203200	1731301317017600	
7605184490373120	3539761721507840	
7394920071430144	3960290559393792	
8364448808960000	636192030720000	
8388608000000000	0	
1342177280000000	1006632960000000	z_2
1261968765747200	1105524581990400	
1610612736000000	469762048000000	
1570347417600000	590558003200000	
1414654853120000	901943132160000	
1672889761792000	127238406144000	
1641600925040640	346260263403520	
1521036898074624	707952344301568	
1677721600000000	0	

Table 3.16: Floating-point numbers $x, y \in \mathbb{F}_{10,34}$ with $X \geq Y$ such that $z = 1/\sqrt{x^2 + y^2}$ is a midpoint.

$$z_1 = 1.776356839400250464677810668945312 5 \cdot 10^{-34}$$
$$z_2 = 8.881784197001252323389053344726562 5 \cdot 10^{-34}$$

x	y	z
5404319552844595200000000000000000	1576259869579673600000000000000000	
4503599627370496000000000000000000	3377699720527872000000000000000000	
5269211564023480320000000000000000	1981583836043018240000000000000000	
4746794007248502784000000000000000	3026418949592973312000000000000000	
4234464513638835159040000000000000	3709524941072530145280000000000000	
5508298661041325211648000000000000	1161856646267550040064000000000000	
5103752916593590193356800000000000	2375493879610755094937600000000000	
4962646853644758191964160000000000	2657706005508419097722880000000000	
4019983297956610516923187200000000	3940938884308614627563929600000000	
4908896652364120987199864832000000	2755721078431934000528359424000000	z_1
5149914854164019792742606438400000	2273684674832136389442876211200000	
5484142688230497667859810877440000	1271001172632702764091262894080000	
4307286551044460811988896841728000	3624713447004776475833090965504000	
4685765143417034444156658442567680	3120075323841344966276075497717760	
5307519345123296639514855463714816	1876566920428820575559681455423488	
5613286575554586214400000000000000	426941244674723020800000000000000	
5564741086220858012205056000000000	851423307780119637000192000000000	
5580549968950457190076907520000000	740761128672925391897231360000000	
5620657309038434535090972052684800	315398826977144693473134667366400	
5629499534213120000000000000000000	0	
1080863910568919040000000000000000	3152519739159347200000000000000000	
1053842312804696064000000000000000	3963167672086036480000000000000000	
1101659732208265042329600000000000	2323713292535100080128000000000000	
1020750583318718038671360000000000	4750987759221510189875200000000000	
1112948217244171602441011200000000	1702846615560239274000384000000000	
1116109993790091438015381504000000	1481522257345850783794462720000000	
1029982970832803958548521287680000	4547369349664272778885752422400000	
1096828537646099533571962175488000	2542002345265405528182525788160000	
1122657315110917242880000000000000	853882489349446041600000000000000	
1124131461807686907018194410536960	630797653954289386946269334732 80	z_2
900719925474099200000000000000000	675539944105574400000000000000000	
949358801449700556800000000000000	605283789918594662400000000000000	
846892902727767031808000000000000	741904988214506029056000000000000	
992529370728951638392832000000000	531541201101683819544576000000000	
803996659591322103384637440000000	788187776861722925512785920000000	
981779330472824197439972966400000	551144215686386800105671884800000	
861457310208892162397779368345600	724942689400955295166618193100800	
937153028683406888831331688513536	624015064768268993255215099543552	
112589990684262400000000000000000	0	

Table 3.17: Number of midpoints in a decade for reciprocal 2D norm with a fixed exponent

Format	decimal32	decimal64	decimal128
p	7	16	34
N_z	1	2	2
N	4	$10+9$	$20+19$

3.7.3 Exact points for reciprocal 2D norm

Theorem 3.22. *Let $x, y \in \mathbb{F}_{\beta,p}$ be such that $(x,y) \neq (0,0)$. Let X, Y denote the integral significands of x, y, and let also z denote $1/\sqrt{x^2+y^2}$.*
• *For $\beta = 2^q$ ($q \in \mathbb{N}_{>0}$), the real z is also in $\mathbb{F}_{2^q,p}$ if and only if $\{x,y\} = \{0, \pm 2^k\}$ for some $k \in \mathbb{Z}$.*
• *For $\beta = 10$, the number z is in $\mathbb{F}_{10,p}$ if and only if its integral significand Z satisfies $Z = 2^k \cdot 5^\ell$, with $10^{p-1} \leq 2^k \cdot 5^\ell < 10^p$ and $k, \ell \in \mathbb{N}$. In this case one has $2^{8p-2k} \cdot 5^{8p-2\ell} \in \mathbb{N}$, and (X,Y) must satisfy*

$$(X \cdot 10^m)^2 + (Y \cdot 10^n)^2 = 2^{8p-2k} \cdot 5^{8p-2\ell},$$

where $m, n \in \mathbb{Z}$ such that $X \cdot 10^m$ and $Y \cdot 10^n$ are in $\mathbb{N}$.

Proof. Without loss of generality, we assume that $1 \leq z < \beta$ and that $0 \leq y \leq x$. Reasoning as in the proof of Theorem 3.20, one may check that necessarily $e_x \in \{-2, -1, 0\}$. Using as usual the normalized representations of x, y and z, from $(x^2+y^2)z = 1$ we deduce

$$Z^2(X^2 \cdot \beta^{2e_x - 2e_y} + Y^2) = \beta^{4p-4-2e_y}. \tag{3.26}$$

If $\beta = 2^q$ for some $q \in \mathbb{N}_{>0}$, then Equation (3.26) implies that $Z = 2^\ell$ for some $\ell \in \mathbb{Z}$. From Equation (3.26), we then deduce

$$\left(X \cdot 2^{q(e_x+2)}\right)^2 + \left(Y \cdot 2^{q(e_y+2)}\right)^2 = 2^{4qp-2\ell}. \tag{3.27}$$

Since $2^{q(p-1)} \leq Z < 2^{qp}$, we deduce that $q(p-1) \leq \ell < qp$, hence $2^{4qp-2\ell}$ is in $\mathbb{N}$. Since both $2^{4qp-2\ell}$ and $X \cdot 2^{q(e_x+2)}$ are in $\mathbb{N}$, it follows that $(Y \cdot 2^{q(e_y+2)})^2$ is also in $\mathbb{N}$, and from Lemma 3.21 we deduce that $Y \cdot 2^{q(e_y+2)} \in \mathbb{N}$. Then Corollary 3.19 implies that the only possible decomposition of $2^{4qp-2\ell}$ as the sum of two squares is $2^{4qp-2\ell} = 0^2 + \left(2^{2qp-\ell}\right)^2$, so that $\{X,Y\} = \{0, 2^{2qp-\ell}\}$. Conversely, if $\{x,y\} = \{0, \pm 2^k\}$, then $1/\sqrt{x^2+y^2} = 2^{-k}$ is in $\mathbb{F}_{2^q,p}$.

Now let us assume that $\beta = 10$. Then Equation (3.26) becomes

$$Z^2\left(X^2 \cdot 10^{2e_x - 2e_y} + Y^2\right) = 10^{4p-4-2e_y}. \tag{3.28}$$

Since $10^{4p-4-2e_y}$ is a multiple of Z, necessarily $Z = 2^k \cdot 5^\ell$ with $k, \ell \in \mathbb{N}$ such that $10^{p-1} \leq 2^k \cdot 5^\ell < 10^p$, which implies $\ell \leq 2p$ and $k \leq 4p$. Moreover, from Equation (3.28) with $Z = 2^k \cdot 5^\ell$ we have

$$\left(X \cdot 10^{2p+e_x+2}\right)^2 + \left(Y \cdot 10^{2p+e_y+2}\right)^2 = 2^{8p-2k} \cdot 5^{8p-2\ell}. \tag{3.29}$$

Since $\left(X \cdot 10^{2p+e_x+2}\right)^2$ and $2^{8p-2k} \cdot 5^{8p-2\ell}$ are both in $\mathbb{N}$, then necessarily $Y \cdot 10^{2p+e_y+2}$ also belongs to $\mathbb{N}$, which concludes the proof. □

Table 3.18: Number of inputs giving an exactpoint for the reciprocal 2D norm for a fixed exponent

Format	decimal32	decimal64	decimal128
p	7	16	34
N_z	42	93	196
N	160	764	3373

In radix 2^q, the pairs (x, y) such that $1/\sqrt{x^2+y^2}$ is an exactpoint are clearly characterized by Theorem 3.22. In radix 10, we used Algorithm 4 to determine all couples (x, y) that leads to an exactpoint. For each $Z = 2^k \cdot 5^\ell$ with $k, \ell \in \mathbb{N}$ such that $10^{p-1} \leq 2^k \cdot 5^\ell < 10^p$, we are reduced to find all decompositions of $2^{8p-2k} \cdot 5^{8p-2\ell}$ as sums of two squares. This is done exactly as explained in §3.7.1.

The number of input couples that leads to an exact point for the reciprocal 2D norm with respect to the precision is given in Figure 3.8. For each basic decimal format of the IEEE 754-2008 standard, Table 3.18 gives the number N_z of exactpoints with a fixed exponent e_z, together with the number N of pairs of significands (X, Y) with $X \geq Y$ such that $1/\sqrt{x^2+y^2}$ is also in $\mathbb{F}_{10,p}$.

Data: Precision p
Result: Couples $x, y; x \geq y$ such that $1/\sqrt{x^2+y^2}$ is an exactpoint in the decade $[10^{-1}; 1)$

num = 0;
for k **from** 0 **to** $4p$ **do**
  $\ell_1 = \left\lceil \log_5(\frac{10^{p-1}}{2^k}) \right\rceil$;
  $\ell_2 = \left\lfloor \log_5(\frac{10^{p}-1}{2^k}) \right\rfloor$;
  /* Decompose $2^{8p-2k} \cdot 5^{8p-2\ell_1} = a^2 + b^2$ */
  $ta_1, tb_1 = \text{decomp25}(8p-2k, 8p-2\ell_1)$;
  if $\ell_1 \neq \ell_2$ **then**
    /* Decompose $2^{8p-2k} \cdot 5^{8p-2\ell_2} = a^2 + b^2$ */
    $ta_2, tb_2 = \text{decomp25}(8p-2k, 8p-2\ell_2)$;
    $ta_1 = ta_1 @ ta_2$; /* Concatenation of lists */
    $tb_1 = tb_1 @ tb_2$;
  end
  for i **from** 1 **to** SizeOf(ta_1) **do**
    if $ta_1[i]$ *and* $tb_1[i]$ *fit on* p *digits* **then**
      print $(ta_1[i], tb_1[i])$; /* Non-normalized output */
    end
  end
end

Algorithm 4: Determining the exactpoints for the reciprocal 2Dnorm function

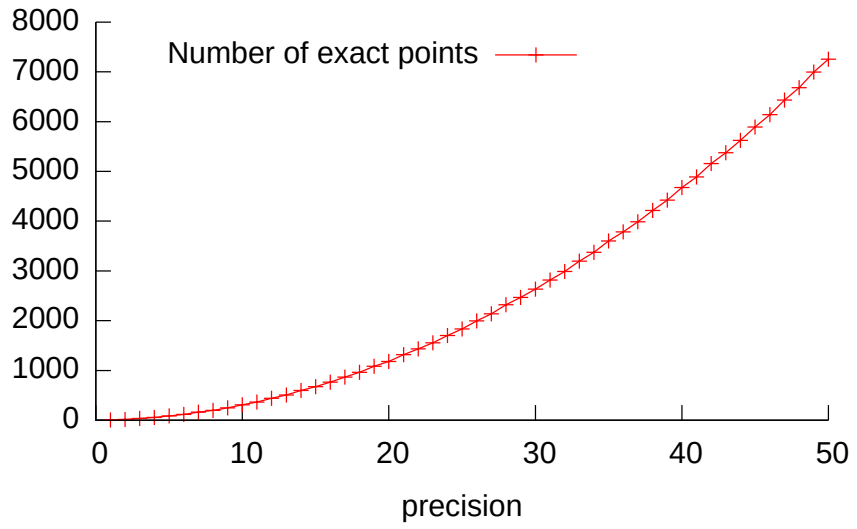


Figure 3.8: Number of inputs leading to exact points for the reciprocal 2D norm in decimal

3.8 Normalization of 2D-vectors

Theorem 3.12 on the function $(x, y) \mapsto x \,/\, \|y\|_2$ implies that $x/\sqrt{x^2+y^2}$, cannot be a midpoint in radix 2. Here we first extend this result to radices 2^q and 10. Then we characterize the exact points of the 2D-normalization function in radix 2^q.

3.8.1 Midpoints for 2D normalization

Theorem 3.23. *Let $x, y \in \mathbb{F}_{\beta,p}$ such that $(x, y) \neq (0, 0)$. If $\beta = 2^q$ ($q \in \mathbb{N}_{>0}$) or $\beta = 10$ then $x/\sqrt{x^2+y^2} \notin \mathbb{M}_{\beta,p}$.*

Proof. Without loss of generality, let us assume $x, y > 0$, and assume that $z = x/\sqrt{x^2+y^2}$ is a midpoint. Hence we write as usual $z = (Z + 1/2) \cdot 10^{e_z - p + 1}$ with $e_z \in \mathbb{Z}$ and $Z \in \mathbb{N}$ such that $\beta^{p-1} \leq Z < \beta^p$. From $x/\sqrt{x^2+y^2} \leq 1$ we deduce that $z \leq 1$, hence $e_z \leq 0$. Using $x^2(1-z^2) = y^2 z^2$ and the normalized representations of x and y gives

$$X^2 \left(4 \cdot \beta^{2p-2-2e_z} - (2Z+1)^2\right) = Y^2 (2Z+1)^2 \cdot \beta^{2e_y - 2e_x}. \tag{3.30}$$

From $e_z \leq 0$, the left-hand side of (3.30) is in $\mathbb{N}$ and thus, using Lemma 3.21, $Y(2Z+1) \cdot \beta^{e_y - e_x} \in \mathbb{N}$. Since $Y^2(2Z+1)^2 \cdot \beta^{2e_y - 2e_z}$ is a multiple of X^2, it follows that $Y(2Z+1) \cdot \beta^{e_y - e_x} = JX$ for some J in $\mathbb{N}_{>0}$. Equation (3.30) then becomes

$$(2 \cdot \beta^{p-1-e_z})^2 = J^2 + (2Z+1)^2, \tag{3.31}$$

which expresses $(2 \cdot \beta^{p-1-e_z})^2$ as a sum of two integer squares.

If $\beta = 2^q$ then $(2 \cdot \beta^{p-1-e_z})^2$ is an even power of two and Corollary 3.19 then implies that it has only one possible decomposition, which is $0^2 + (2^{q(p-1-e_z)+1})^2$. However, this contradicts the fact that both J and $2Z+1$ are positive integers.

With $\beta = 10$, Equation (3.31) becomes

$$2^{2p-2e_z} \cdot 5^{2p-2-2e_z} = J^2 + (2Z+1)^2. \tag{3.32}$$

Since $2p - 2e_z$ is even, according to Theorem 3.18, one has necessarily

$$2^{2p-2e_z} \cdot 5^{2p-2-2e_z} = |\Re(c)|^2 + |\Im(c)|^2,$$

with $c = 2^{p-e_z}(2+i)^k(2-i)^{2p-2-2e_z-k}$ for some $k \in \mathbb{N}$, and one may check that both $|\Re(c)|$ and $|\Im(c)|$ are even. Hence the two squares in the right-hand side of Equation (3.32) must be even, which is a contradiction since $2Z + 1$ is odd. □

3.8.2 Exact points for 2D normalization

The next theorem provides a characterization of the exact points of the 2D-normalization function in radix 2^q.

Theorem 3.24. *Let $q \in \mathbb{N}_{>0}$ and let $x, y \in \mathbb{F}_{2^q,p}$ be such that $(x, y) \neq (0, 0)$. One has $= x/\sqrt{x^2 + y^2} \in \mathbb{F}_{2^q,p}$ if and only if $x = 0$ or $y = 0$.*

Proof. The "if" statement is obvious. Conversely, assume that $z \in \mathbb{F}_{2^q,p}$ and that both x and y are nonzero. We can restrict to $x, y > 0$ with no loss of generality. Let $z = x/\sqrt{x^2 + y^2}$. Since $z \leq 1$, necessarily $e_z \leq 0$. Then, using $x^2(1 - z^2) = y^2z^2$ and the normalized representations of x and y,

$$X^2(\beta^{2p-2e_z-2} - Z^2) = Y^2Z^2 \cdot \beta^{2e_y-2e_x}, \tag{3.33}$$

From $e_z \leq 0$ it follows that the left-hand side of (3.33) is in $\mathbb{N}$ and, due to Lemma 3.21, so is $YZ \cdot \beta^{e_y-e_y}$. Now, since $Z^2Y^2 \cdot \beta^{2e_y-2e_x}$ is a multiple of X^2, we have $ZY \cdot \beta^{e_y-e_x} = JX$ for some $J \in \mathbb{N}_{>0}$. Then we obtain from Equation (3.33)

$$\left(\beta^{e_z-p+1}\right)^2 = J^2 + Z^2. \tag{3.34}$$

When $\beta = 2^q$, Corollary 3.19 implies that either J or Z is zero, a contradiction. Hence, only the cases $x = 0$ or $y = 0$ are obviously exact points. □

In radix 10, we do not have simple results to characterize the exact points of the 2D-normalization function. But they can of course be enumerated using Equation (3.34), at least for some small precisions p. Using Theorem 3.18, we enumerate all the pairs (Z, J) for a fixed e_z such that Equation (3.34) is satisfied. Without loss of generality, we fix $e_x = 0$. The inputs x and y can then be found by searching the points $(X, Y \cdot 10^{e_y})$ on the line $YZ \cdot 10^{e_y} = JX$, with $10^{p-1} \leq X < 10^p$ and $X \in \mathbb{N}$. For some small precisions, the following table gives the number of pairs of inputs (X, Y) such that $x/\sqrt{x^2 + y^2}$ is an exact point:

p	1	2	3	4	5	6	7
$e_z = -1$	4	54	558	5622	56254	562696	5630268
$e_z = -2$	0	0	0	6	60	597	2889

This experiment suggests that the number of (x, y) such that $x/\sqrt{x^2 + y^2}$ is an exact point grows very rapidly with p, and that no useful enumeration can be performed.

3.9 2D Euclidean norm

Let x and y be two numbers in $\mathbb{F}_{\beta,p}$, and assume that the Euclidean norm $z = \sqrt{x^2 + y^2}$ is a midpoint. We use the normalized representations of x, y and we write as usual $z = (Z + 1/2) \cdot \beta^{e_z - p + 1}$. Without loss of generality, we assume that $x \geq y$, which implies $e_z \geq e_x \geq e_y$. Then from $x^2 + y^2 = z^2$ it follows that

$$4(Y^2 + X^2 \cdot \beta^{2e_x - 2e_y}) = (2Z + 1)^2 \cdot \beta^{2e_z - 2e_y}. \tag{3.35}$$

When β is odd, the right-hand side of Equation (3.35) is odd, while the left-hand side is always even. Hence, if the radix β is odd, $\sqrt{x^2 + y^2}$ cannot be a midpoint, and this observation can be generalized to the Euclidean norm in higher dimensions. Nevertheless, this not a very useful result since it does not hold for binary, decimal nor hexadecimal arithmetic.

For even radices, we do not have general results. Equation (3.35) has solutions, and exhaustive enumeration can be performed at least for small precisions. In radices 2 and 10, and for some small precisions p, the following tables display the number N of input pairs (x, y), with $x \geq y$, such that $z = \sqrt{x^2 + y^2}$ is a midpoint in the interval $[1, \beta)$.

Radix 2

p	1	2	3	4	5	6	7	8	9	10
N	0	1	1	3	5	18	30	76	155	348

Radix 10

p	1	2	3	4
N	0	11	177	2428

These experiments suggest that the number of midpoints for the function $(x, y) \mapsto \sqrt{x^2 + y^2}$ grows very rapidly with p.

On the other hand, in one-dimension the Euclidean norm reduces to the absolute value, which suffices to see that it admits all exact points, whatever the parity of β.

3.10 Inputs and/or outputs in the subnormal range

In this chapter, we have assumed that the exponent range is unbounded and all floating-point numbers are normalized. Let us now briefly describe what changes if the input or output variables are in the subnormal range.

A subnormal floating-point number x can be written

$$x = X \cdot \beta^{1 - p + e_{\min}},$$

with $0 < X \leq \beta^{p-1}$, i.e., x is of absolute value less than $\beta^{e_{\min}}$. We also call subnormal midpoint a number z of the form

$$z = \left(Z + \frac{1}{2}\right) \cdot \beta^{1 - p + e_{\min}},$$

with $0 \leq Z < \beta^{p-1}$. In the following, we will note $\mathbb{SF}_{\beta,p,e_{\min}}$ the set of subnormal floating-point number of precision p in radix β, and $\mathbb{SM}_{\beta,p,e_{\min}}$ the set of subnormal midpoints of precision p in radix β.

An interesting fact is that when considering an unlimited exponent range, a subnormal number becomes a normal floating-point number with trailing zeros. For example, the binary32 subnormal number

$$x = 0.00010110101100001101011 \cdot 2^{-126}$$

can be represented as

$$x = 1.01101011000011010110000 \cdot 2^{-130}.$$

Since $\mathbb{F}_{\beta,p}$ is the set of floating-point number assuming an unbounded exponent range, we deduce that $\mathbb{SF}_{\beta,p,e_{\min}}$ is a subset of $\mathbb{F}_{\beta,p}$. Notice however that $\mathbb{SM}_{\beta,p,e_{\min}}$ is not a subset of $\mathbb{M}_{\beta,p}$, but a subset of $\mathbb{F}_{\beta,p}$.

From this, we deduce the following lemmata.

Lemma 3.25. *Assume that function f admits no midpoints in $\mathbb{F}_{\beta,p}$. Then for all $x_1, \ldots, x_n \in \mathbb{F}_{\beta,p} \bigcup \mathbb{SF}_{\beta,p,e_{\min}}$, the value $f(x_1, \ldots, x_n)$ is not in $\mathbb{M}_{\beta,p}$.*

Lemma 3.26. *Assume that function f admits no exact points in $\mathbb{F}_{\beta,p}$. Then for all $x_1, \ldots, x_n \in \mathbb{F}_{\beta,p} \bigcup \mathbb{SF}_{\beta,p,e_{\min}}$, the value $f(x_1, \ldots, x_n)$ is not in $\mathbb{F}_{\beta,p} \bigcup \mathbb{SF}_{\beta,p,e_{\min}}$.*

Lemma 3.27. *Assume that function f admits a set I of inputs $x_1, \ldots, x_n \in \mathbb{F}_{\beta,p}$ such that $f(x_1, \ldots, x_n)$ is an exact points. Then the set of inputs $x_1, \ldots, x_n \in \mathbb{F}_{\beta,p} \bigcup \mathbb{SF}_{\beta,p,e_{\min}}$ such that $f(x_1, \ldots x_n) \in \mathbb{SF}_{\beta,p,e_{\min}}$ and the set of inputs $x_1, \ldots, x_n \in \mathbb{SF}_{\beta,p}$ such that $f(x_1, \ldots x_n) \in \mathbb{F}_{\beta,p,e_{\min}}$ are subsets of I.*

Lemma 3.28. *Assume that function f admits a set I of inputs $x_1, \ldots, x_n \in \mathbb{F}_{\beta,p}$ such that $f(x_1, \ldots, x_n)$ is a midpoint. Then the set of inputs $x_1, \ldots, x_n \in \mathbb{SF}_{\beta,p}$ such that $f(x_1, \ldots x_n) \in \mathbb{M}_{\beta,p,e_{\min}}$ is a subset of I.*

Proof of Lemma 3.25, 3.26, 3.27 and 3.28. Since $\mathbb{SF}_{\beta,p,e_{\min}}$ is a subset of $\mathbb{F}_{\beta,p}$, we know that $\mathbb{F}_{\beta,p} = \mathbb{SF}_{\beta,p,e_{\min}} \bigcup \mathbb{F}_{\beta,p}$, which proves the two first lemmata.

If $I = \{(x_1, \ldots, x_n) \in \mathbb{F}_{\beta,p} \bigcup \mathbb{SF}_{\beta,p,e_{\min}} | f(x_1, \ldots, x_n) \in \mathbb{F}_{\beta,p}\}$, since $\mathbb{SF}_{\beta,p,e_{\min}}$ is a subset of $\mathbb{F}_{\beta,p}$, the set of inputs such that $f(x_1, \ldots, x_n)$ is in $\mathbb{SF}_{\beta,p,e_{\min}}$ is a subset of I. □

Except for the midpoints in the subnormal range, we can use the four lemmata and the proofs of the previous sections to deduce what happens for subnormal floating-points numbers.

Example 3.29. Function $(x, y) \mapsto x \,/\, \|y\|_2$ of Section 3.4 admits no midpoints in radix 2 when the inputs are normalized floating-point numbers with unbounded exponent range. Then, using Lemma 3.25, we know that this function admits no midpoints in radix 2, even if some or all inputs are subnormal floating-point numbers.

Notice however that Lemma 3.25 proves nothing about midpoints in the subnormal range ($\mathbb{SM}_{\beta,p,e_{\min}}$). This is however covered in §3.10.3.

Example 3.30. The reciprocal square-root function admits two midpoints in the decimal64 format, namely $y = 7.036874417766400 \cdot 10^{e_y}$ if e_y is odd, and $y = 2.814749767106560 \cdot 10^{e_y}$ if e_y is even. Then, according to Lemma 3.28 one can check that in the subnormal range, only the following entries will lead to midpoints (in the normalized range):

$$y = 0.070368744177664 \cdot 10^{-383},$$

$$y = 0.281474976710656 \cdot 10^{-383}.$$

If a function f admits no exact points for normal floating-point numbers, it will admit no exact points when some inputs are subnormals. If there are a few exact points of the function f, the formula that characterize those exact points can yield some false positives when some inputs are subnormals, but will detect them all.

In the case of midpoints in the subnormal range, we have to check each function individually. For the same reason as the lemmata 3.25, 3.26, 3.27, and 3.28, we only consider normal floating-point numbers with unbounded exponent range for the inputs. The results will be the same when considering that some inputs might be subnormals.

Characterizing subnormal midpoints, when there is no easy characterization of midpoints in the normalized range, seems rather pointless. Hence, we focus on radices and functions where normalized midpoints are easily characterized or enumerated.

3.10.1 Square root

For the square-root function, notice that if $\beta^{1-p+e_{\min}} < x < \beta^{e_{max}}$, $\sqrt{x}$ is not subnormal. It means that, for all inputs of precision p (even for subnormal numbers), the output cannot be in the range of subnormal numbers of precision p, hence cannot be a midpoint in the subnormal range.

The only case in which we can have midpoints in the subnormal range for square-root is if we are working with a larger input precision (p_i) than the output precision (p_o). In this case, when an FMA operation is available, midpoints in the subnormal range can be detected at run time: if the rounded approximate $\tilde{z} \approx \sqrt{y}$ would lead to a midpoints in the subnormal range of the output format, then one can check by computing $t = \mathrm{RN}_{p_i}(\tilde{z}^2 - y)$ whether this is really a midpoint ($t = 0$), or if we are in a case of double-rounding ($t \neq 0$) (see §1.5.2). In addition, if $t \neq 0$, then the sign of t indicates in which way $\tilde{z}$ should be rounded in the output format: $t > 0$ means $\tilde{z}$ should be rounded down ($\mathrm{RD}_{p_o}(\tilde{z})$), and $t < 0$ means $\tilde{z}$ should be rounded up ($\mathrm{RU}_{p_o}(\tilde{z})$).

3.10.2 Reciprocal square root

Theorem 3.31. *Let $y \in \mathbb{F}_{\beta,p}$. If $\beta = 2$, then $1/\sqrt{y}$ cannot be in $\mathbb{SM}_{\beta,p,e_{\min}}$. If $\beta = 10$, then $1/\sqrt{y}$ is in $\mathbb{SM}_{\beta,p,e_{\min}}$ if and only if the integral significand Y of y is of the form*

$$2^{3p-1-2e_{\min}-e_y} \cdot 5^{3p-3-2k-2e_{\min}-e_y}$$

with $0 \leq k < p - 1 + p\log_5(2)$.

Notice that having Y of the form

$$2^{3p-1-2e_{\min}-e_y} \cdot 5^{3p-3-2k-2e_{\min}-e_y}$$

with $0 \leq k < p - 1 + p\log_5(2)$ is not sufficient in Theorem 3.31: it should also be checked that Y correspond to an integral significand, i.e., $\beta^{p-1} \leq < \beta^p$. This will also be the case in Theorem 3.33 for the reciprocal.

Proof. Let $z = 1/\sqrt{y}$ be a subnormal midpoint. From $z < \beta^{e_{\min}}$, we deduce that $e_y \geq -2e_{\min}$. Let $y = Y \cdot \beta^{e_y-p+1}$ and $z = (Z + 1/2) \cdot \beta^{e_{\min}-p+1}$ be the representations of y and z. From $yz^2 = 1$ we deduce that

$$(2Z + 1)^2 Y \beta^{e_y+2e_{\min}} = 4\beta^{3p-3}. \tag{3.36}$$

Equation (3.36) has no solution for $\beta = 2$. Now, if $\beta = 10$, we deduce from Equation (3.36) that $2Z+1 = 5^k$ for some integer k. Since $Z < \beta^{p-1}$, we know that $0 \leq k < p-1+p\log_5(2)$. From Equation (3.36), we deduce that

$$Y 10^{e_y-p+1} = 2^{2p-2e_{\min}} \cdot 5^{2p-2-2k-2e_{\min}}.$$

Conversely, if the integral significand of y satisfies $Y = 2^{3p-1-2e_{\min}-e_y} \cdot 5^{3p-3-2k-2e_{\min}-e_y}$, we deduce from Equation (3.36) that $2Z + 1 = 5^k$, and from $0 \leq k < p - 1 + p\log_5(2)$, we deduce that z is in $\mathbb{SM}_{\beta,p,e_{\min}}$. □

3.10.3 Division, $x/\|y\|_2$

In radix $\beta = 2$, there are many subnormal midpoints. Indeed, all subnormal midpoints can be obtained, by choosing adequate inputs.

Example 3.32 (Binary32)**.**

$$x = 1.10000000000000000000000 \cdot 2^{-63},$$

$$y = 1.00000000000000000000000 \cdot 2^{63},$$

$$x/y = \underbrace{0.00000000000000000000001}_{p=24}\ 1 \cdot 2^{-126}.$$

That example is easily generalized to all subnormal midpoints. Let z be a subnormal midpoint, with $z = (Z + 1/2) \cdot 2^{e_{\min}-p+1}$ and $0 \leq Z < 2^{p-1}$. If $Z = 0$, then $x = 2^{e_{\min}-p}$ and $y = 1$ gives $x/y = z$. If $Z \neq 0$, then there exists an integer k such that $2^k \leq Z < 2^{k+1}$ and $0 \leq k \leq p - 2$. Taking $x = (2Z + 1) \cdot 2^{k-p+1}$ and $y = 2^{-e_{\min}}$, we can check that $x, y \in \mathbb{F}_{\beta,p}$, and that $x/y \in \mathbb{SM}_{\beta,p,e_{\min}}$.

Since x/y is a sub-case of the function $x/\|y\|_2$, we also know that all subnormal midpoints can be obtained for the latter function.

3.10.4 Reciprocal

Theorem 3.33. *Let $y \in \mathbb{F}_{\beta,p}$. If $\beta = 2$, then $1/y$ cannot be in $\mathbb{SM}_{\beta,p,e_{\min}}$. If $\beta = 10$, then $1/y$ is in $\mathbb{SM}_{\beta,p,e_{\min}}$ if and only if the integral significand Y of y is of the form $2^{2p-1-e_{\min}-e_y} \cdot 5^{2p-2-k-e_{\min}-e_y}$, with $0 \leq k < p - 1 + p\log_5(2)$.*

Proof. Let $z = 1/y$ be a subnormal midpoint. From $1/2\beta^{1-p+e_{\min}} \leq z < \beta^{e_{\min}}$, we deduce that $e_y \leq -e_{\min} - p$. Let $y = Y \cdot \beta^{e_y-p+1}$ and $z = (Z + 1/2) \cdot \beta^{e_{\min}-p+1}$ be the normalized representations of y and z. From $yz = 1$ we deduce that

$$(2Z + 1)Y = 2\beta^{2p-2-e_{\min}-e_y}. \tag{3.37}$$

Equation (3.37) has no solution for $\beta = 2$. Now, if $\beta = 10$, we deduce from Equation (3.37) that $2Z+1 = 5^k$ for some integer k. Since $Z < \beta^{p-1}$, we know that $0 \leq k < p-1+p\log_5(2)$. From Equation (3.37), we deduce that

$$Y = 2^{2p-1-e_{\min}-e_y} \cdot 5^{2p-2-k-e_{\min}-e_y}.$$

Conversely, if $Y = 2^{2p-1-e_{\min}-e_y} \cdot 5^{2p-2-k-e_{\min}-e_y}$, we deduce from Equation (3.37) that $2Z + 1 = 5^k$, and from $k < p - 1 + p\log_5(2)$, we deduce that z is in $\mathbb{SM}_{\beta,p,e_{\min}}$. □

3.10.5 Reciprocal 2D Euclidean norm

Theorem 3.34. *Let $x, y \in \mathbb{F}_{\beta,p}$. If $\beta = 2$, then $1/\sqrt{x^2 + y^2}$ cannot be in $\mathbb{SM}_{\beta,p,e_{\min}}$. If $\beta = 10$, the subnormal midpoints are of the form*

$$z = 2^{r-1} \cdot 5^s \cdot 10^{e_{\min}-p+1},$$

with $0 < 2^r 5^s < 2 \cdot 10^{p-1}$. The inputs x, y are then given by the following decomposition in a sum of two squares:

$$\left(X \cdot 10^{e_x+p-e_{\min}}\right)^2 + \left(Y \cdot 10^{e_y+p-e_{\min}}\right)^2 = 2^{6p-2-4e_{\min}-2r} \cdot 5^{6p-4-4e_{\min}-2s}.$$

Proof. Let $z = 1/\sqrt{x^2 + y^2}$ be a subnormal midpoint. Without loss of generality, we assume that $x \geq y$, which implies $\beta^{e_x-2} < z \leq \beta^{-e_x}$ (as in §3.7.2). From $z < \beta^{e_{\min}}$, we deduce that $-e_y \geq -e_{\min} > 0$.

Let $x = X \cdot \beta^{e_x-p+1}$ and $y = Y \cdot \beta^{e_y-p+1}$ be the normalized representations of x and y, and $z = (Z+1/2) \cdot \beta^{e_{\min}-p+1}$ be the normalized representations of z. From $(x^2+y^2)z^2 = 1$, we deduce

$$(X^2\beta^{2e_x-2e_y} + Y^2)(2Z + 1)^2 = 4\beta^{4p-4-2e_y-2e_{\min}}. \tag{3.38}$$

If $\beta = 2$, Equation (3.38) has no solution. If $\beta = 10$, then $2Z + 1 = 2^r 5^s$ with $0 < 2^r 5^s < 2 \cdot 10^{p-1}$. From Equation (3.38), we deduce

$$\left(X \cdot 10^{e_x+p-e_{\min}}\right)^2 + \left(Y \cdot 10^{e_y+p-e_{\min}}\right)^2 = 2^{6p-2-4e_{\min}-2r} \cdot 5^{6p-4-4e_{\min}-2s}.$$

□

By decomposing the right hand side integer into the sum of two squares, it is then possible to determine every couple of inputs that give a midpoint in the subnormal range. In decimal16, there are 666 couples of inputs that gives midpoints in the subnormal range, and there are 8025 such couples for decimal32.

3.10.6 Normalization of 2D-vectors

Theorem 3.35. *Let $x, y \in \mathbb{F}_{\beta,p}$. If $\beta = 2$ or $\beta = 10$, then $x/\sqrt{x^2+y^2}$ cannot be in $\mathbb{SM}_{\beta,p,e_{\min}}$.*

Proof. Let $z = x/\sqrt{x^2+y^2}$ be a subnormal midpoint, and let $x = X \cdot \beta^{e_x-p+1}$ and $y = Y \cdot \beta^{e_y-p+1}$ be the normalized representations of x and y. Let $z = (Z+1/2) \cdot \beta^{e_{\min}-p+1}$ be the normalized representations of z. From $x^2(1-z^2) = y^2z^2$, we deduce

$$X^2\left(4\beta^{2p-2-2e_{\min}} - (2Z+1)^2\right) = Y^2(2Z+1)^2\beta^{2e_y-2e_x} \tag{3.39}$$

Since all the terms in the left hand side of Equation (3.39) are integers, we deduce from lemma 3.21 that $Y(2Z+1)\beta^{e_y-e_x}$ is an integer multiple of X. From $Y(2Z+1)\beta^{e_y-e_x} = JX$, we then deduce from Equation (3.39) that

$$4\beta^{2p-2-2e_{\min}} = (2Z+1)^2 + J^2,$$

which has no solution (as in §3.8.1). □

3.11 Conclusion

We have shown that for several simple algebraic functions ($\sqrt{y}$, $1/\sqrt{y}$, x^k for $k \in \mathbb{N}_{>0}$, $x/\|y\|_2$, x/y, $1/y$, $1/\sqrt{x^2+y^2}$, $x/\sqrt{x^2+y^2}$), we can obtain useful information on the existence of midpoints and exact points. This information can be used for simplifying or improving the performance of programs that evaluate these functions.

Finding midpoints and exact points would also be of interest for the most common transcendental functions (sine, cosine, exponential, logarithm,...). Providing these functions with correct rounding is a difficult problem, known as the *Table-Maker's Dilemma* [32, 45]. For the most simple transcendental functions, those built from the complex exponential and logarithm, one can deduce the nonexistence of midpoints from the following corollary of Lindemann's theorem (see for example [3, p. 6]):

Theorem 3.36 (Lindemann)**.** *e^z is transcendental for every non-zero algebraic complex number z.*

Since floating-point numbers as well as midpoints are algebraic numbers, Theorem 3.36 allows us to deduce that for any radix and precision, if x is a floating-point number then $\ln(x)$, $\exp(x)$, $\sin(x)$, $\cos(x)$, $\tan(x)$, $\arctan(x)$, $\arcsin(x)$ and $\arccos(x)$ cannot be midpoints. Furthermore, the only exact points are $\ln(1) = 0$, $\exp(0) = 1$, $\sin(0) = 0$, $\cos(0) = 1$, $\tan(0) = 0$, $\arctan(0) = 0$, $\arcsin(0) = 0$, and $\arccos(1) = 0$.

The case of radix-2 and radix-10 exponentials and logarithms have to be treated more carefully. But one can prove that the radix-2 or 10 logarithm of a rational number is either an integer or an irrational number. This gives the following result. Assume that the exponent size is less than the precision (which is true in any reasonable floating-point system), and that x is a floating-point number. Then we have the following:

- $\log_2(x)$ cannot be a midpoint. It can be an exact point only when $x = 2^k$, where k is an integer;

- $\log_{10}(x)$ cannot be a midpoint. It can be an exact point only when $x = 10^k$, where k is an integer.

It is always possible to build ad-hoc transcendental functions for which something can be said about midpoints or exact points. Unfortunately, for the many common non-elementary transcendental functions useful in scientific applications (physics, statistics, etc.), almost nothing is known about their midpoints or exact points in floating point arithmetic.

Consider for instance the Gamma function. We know that if n is a nonnegative integer then $\Gamma(n) = (n-1)!$ is an integer too (which implies the existence of midpoints in some cases, e.g., in radix-2 arithmetic with $p = 3$, the number $6_{10} = 110_2$ is a floating-point number, and $\Gamma(6) = 5! = 120_{10} = 1111000_2$ is a midpoint). Although we have no proof of that, it is extremely unlikely that Gamma of a non-integer floating-point number could be a midpoint or an exact point. To our knowledge (see for example [57]), the only result that can be used to deal with a very few cases is that $\Gamma(x)$ is shown to be irrational if x modulo 1 belongs to $\{1/6, 1/4, 1/3, 1/2, 2/3, 3/4, 5/6\}$.

Chapter 4

Newton-Raphson division using an FMA

To compute the quotient of two numbers, there are essentially two classes of algorithms: the digit recurrence algorithms, such as the SRT division, and the Newton-Raphson division algorithms. The SRT division is based on the ordinary pencil-and-paper division, and its convergence is known to be linear, whereas the Newton-Raphson algorithm converges quadratically.

The most commonly used division algorithm in hardware is the SRT division. Indeed, for the considered floating-point formats, it is more efficient to use these kinds of algorithms in hardware.

However, due to the constant increase of format lengths and the introduction of correctly rounded FMA operations in the IEEE-754-2008 standard [27], it may become preferable to perform division using the Newton-Raphson algorithms [9, 43]. With an FMA unit available in hardware, the quotient can be computed efficiently purely in software, as it was already the case for example on the HP/Intel Itanium architecture [9, 43]: in this case, there is no need for a hardware division unit.

While we start with a small remainder of the SRT algorithm, this chapter focuses on the proof of correct rounding of Newton-Raphson based algorithms, for both binary and decimal floating-point arithmetic.

4.1 SRT division

The most commonly used division algorithm in today's hardware is known as the SRT division, named after its three finders: D.W. Sweeney, J.E. Robertson and K.D. Tocher [52, 55]. This algorithm is roughly like the ordinary paper-and-pencil division, as depicted in figure 4.1.

In the ordinary division, one guesses a next quotient digit $y^{(-n)}$ one at a time, and computes a residual r_n, usually by subtracting $by^{(-n)}$ to the previous residual r_{n-1}. Hence, the ordinary division might be described as follows, where $\mathrm{SEL}(r_{n-1}b) = \left\lfloor \frac{r_{n-1}}{b} \right\rfloor$.

The SRT division uses many tweaks to improve the ordinary division for radix-2 hardware computers. First, instead of computing the remainder r_n, the SRT algorithm computes $w_n = \beta^n r_n$. While w_n is basically the same thing as the remainder (w_n being r_n with

$$
\begin{array}{l|l}
r_0 = \overbrace{149}^{a} & \overbrace{527}^{b} \\
r_1 = \quad 43.6 & y = 0.2827\ldots \\
r_2 = \quad\ 1.44 & \quad = 0.y^{(-1)}y^{(-2)}y^{(-3)}y^{(-4)}\ldots \\
r_3 = \quad\ .386 & \\
r_4 = \quad\ .0171 & \\
r_5 = \quad\ \ldots &
\end{array}
$$

Figure 4.1: Example of an ordinary paper-and-pencil division

Data: a, b fixed point numbers such that $a < b < \beta a$
Result: The quotient y and a remainder r_k such that $a = by + r_k$

$r_0 = a$, $y^{(0)} = 0$;
for n **from** 1 **to** k **do**
    $y^{(-n)} = \mathrm{SEL}(r_{n-1}, b)$;
    $r_n = r_{n-1} - by^{(-n)}\beta^{-n}$;
end
$y = y^{(0)}.y^{(-1)}y^{(-2)}y^{(-3)}\ldots y^{(-k)}$

Algorithm 5: Ordinary paper-and-pencil division in radix β

a shift), the main loop becomes

$$
\begin{aligned}
y^{(-n)} &= \mathrm{SEL}(w_{n-1}, b), \\
w_n &= \beta\left(w_{n-1} - by^{(-n)}\right).
\end{aligned}
\tag{4.1}
$$

In this new iteration, the variable shift previously used to compute r_n has been replaced with a fixed shift, which is preferable in hardware implementation. To also improve division timing, a redundant representation (usually borrow-save) is used for w_n, which improves the time for the subtraction in (4.1).

That redundant representation, however, makes the choice of $y^{(-n)}$ more complex. In practice, due to the representation of w_n, the SEL() function is based on a few most significant bits of w_n only, which imposes the choice of another redundant number system for y.

4.2 Newton-Raphson division

The Newton-Raphson method is a general method used to approximate a root α of a function f. We assume here[1] that the function f is $\mathscr{C}^2$, with $f'(\alpha) \neq 0$. We then know that there is an interval I around α such that all $y \in I$ are such that $f'(y) > 0$. Assuming

[1]Although the Newton-Raphson method works on a bigger set of functions, all functions considered in this chapter are $\mathscr{C}^2$ on the set of positive real numbers, with $f'(\alpha) \neq 0$.

y_n is an approximation to α, with $y_n \in I$, a second order Taylor's expansion of f around y_n on the interval I gives

$$f(\alpha) = f(y_n) + (\alpha - y_n) f'(y_n) + \frac{(\alpha - y_n)^2}{2} f''(\xi_n),$$

with ξ between y_n and α. Since f is $\mathscr{C}^2$ and $y_n \in I$, we have $f'(y_n) \neq 0$, hence

$$\alpha = \underbrace{y_n - \frac{f(y_n)}{f'(y_n)}}_{y_{n+1}} - (\alpha - y_n)^2 \cdot K_n,$$

with $K_n = \frac{f''(\xi_n)}{2f'(y_n)}$. Since f is $\mathscr{C}^2$, we know that for all n, $K_n \leq K = \sup_{y \in I} \frac{1}{2} \left| \frac{f''(y)}{f'(y)} \right|$. Hence

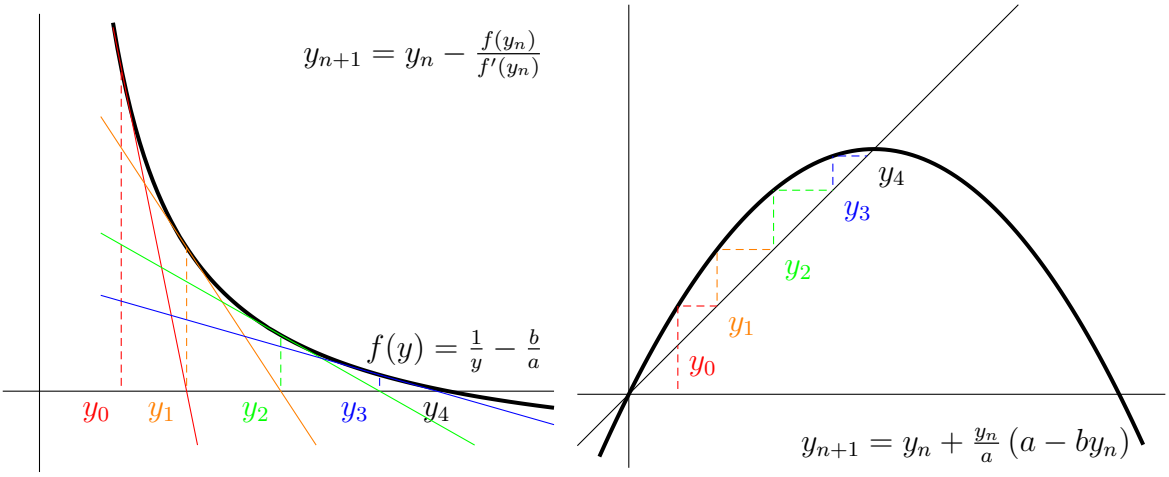
$$|\alpha - y_{n+1}| \leq |\alpha - y_n|^2 \cdot K.$$

This proves that if we take y_0 close enough to a root α of multiplicity 2 of a $\mathscr{C}^2$ function ($|y_0 - \alpha| < 1$), iterating the formula

$$y_{n+1} = y_n - \frac{f(y_n)}{f'(y_n)}, \tag{4.2}$$

will correctly approximate the root α with a quadratic convergence. But how can it be applied to the computation of quotient?

4.2.1 Mathematical iteration



(a) Newton-Raphson's method for the function f

(b) Convergence of Newton-Raphson's iteration

Figure 4.2: Newton-Raphson's iteration on function $f(y) = \frac{1}{y} - \frac{b}{a}$ used to compute $\frac{a}{b}$

To compute the quotient $\frac{a}{b}$, starting from a first approximation y_0, we can use Newton-Raphson's method on the function $f(y) = \frac{1}{y} - \frac{b}{a}$, which gives the iteration

$$y_{n+1} = y_n + \frac{y_n}{a}(a - by_n), \tag{4.3}$$

which is illustrated in Figure 4.2. However, this iteration might seem unusable to compute a quotient, since there is another quotient in the formula. But since y_n is an approximation to $\frac{a}{b}$, we can replace $\frac{y_n}{a}$ in Formula (4.3) by x_m, an approximation to $\frac{1}{b}$.

But now, how do we compute an approximation to $\frac{1}{b}$? This is simply done by using again Newton-Raphson's method on function $f(x) = \frac{1}{x} - b$, which finally gives the iterations

$$x_{m+1} = x_m + x_m(1 - bx_m), \tag{4.4}$$

$$y_{n+1} = y_n + x_m(a - by_n), \tag{4.5}$$

For example, we can compute the quotient $\frac{149}{527}$ starting from an approximation x_0 to $\frac{1}{527}$ as follows.

$$\begin{array}{llll} x_0 & & = 0.0019 & \approx 1/b, \\ x_1 & = x_0 + x_0(1 - bx_0) & = 0.00189753 & \approx 1/b, \\ y_1 & = a \times x_1 & = \quad 0.28273197 & \approx a/b, \\ y_2 & = y_1 + x_1(a - by_1) & = \quad 0.2827324478170293 & \approx a/b. \end{array}$$

$$a/b = 0.28273244781783681214421\ldots$$

There are two ways of using the basic operation Fused-Multiply and Add (FMA, see §1.4.1) to compute a quotient (or a reciprocal) using Newton-Raphson's iteration. We can either perform

$$\text{Markstein} \quad \begin{cases} r_{n+1} = a - by_n \\ y_{n+1} = y_n + r_{n+1}x_m, \end{cases} \tag{4.6}$$

or

$$\text{Goldschmidt} \quad \begin{cases} r_0 = a - by_0 \\ r_{n+2} = r_{n+1}^2 \\ y_{n+1} = y_n + r_{n+1}x_m. \end{cases} \tag{4.7}$$

In this chapter, we consider both iterations. Although they are mathematically equivalent to the Newton-Raphson's iteration (4.5), they will behave differently when implementation is at stake. We can see that in Goldschmidt's iteration, the computations of r_{n+2} and y_{n+1} are independent. Hence, if we have several FMA units, or a pipelined FMA, Goldschmidt's iterations will go faster than Markstein's iteration. However, the Markstein iteration will prove to be less susceptible to rounding errors. Roughly speaking, Goldschmidt's iteration only uses a and b at the first step. Hence, we loose more and more information by roundings at each step, and this loss of information is never corrected by using the initial information a and b in the subsequent iterations.

4.2.2 Floating-point algorithms

To compute a/b, an initial approximation $\hat{x}_0$ to $1/b$ is obtained from a lookup table addressed by the first digits of b [12, 54]. One next refines the approximation to $1/b$ using iteration (4.8) below:

$$\hat{x}_{n+1} = \hat{x}_n + \hat{x}_n\,(1 - b\hat{x}_n)\,. \tag{4.8}$$

Then $\hat{y}_n = a\hat{x}_n$ is taken as an initial approximation to a/b that can be improved using

$$\hat{y}_{n+1} = \hat{y}_n + \hat{x}_m\,(a - b\hat{y}_n)\,. \tag{4.9}$$

There are several ways of using the FMA to perform Newton-Raphson iterations. To compute the reciprocal $1/b$ using Equation (4.8), we have the following two iterations:

$$\text{Markstein} \quad \begin{cases} \tilde{r}_{n+1} = \mathrm{RN}(1 - b\tilde{x}_n) \\ \tilde{x}_{n+1} = \mathrm{RN}(\tilde{x}_n + \tilde{r}_{n+1}\tilde{x}_n) \end{cases} \tag{4.10}$$

$$\text{Goldschmidt} \quad \begin{cases} \tilde{r}_1 = \mathrm{RN}(1 - b\tilde{x}_0) \\ \tilde{r}_{n+2} = \mathrm{RN}(\tilde{r}_{n+1}^2) \\ \tilde{x}_{n+1} = \mathrm{RN}(\tilde{x}_n + \tilde{r}_{n+1}\tilde{x}_n) \end{cases} \tag{4.11}$$

The Markstein iteration [42, 43] immediately derives from Equation (4.8). The Goldschmidt iteration [19] is obtained from the Markstein iteration (4.10), by substituting r_{n+1} with r_n^2. Even if both iterations are mathematically equivalent, when we use them in floating-point arithmetic, they behave differently, as Example 4.1 shows.

Example 4.1. In binary16 ($p_w = 11, \beta = 2$):

$$b = 1.1001011001$$
$$1/b = 0.\underbrace{10100001010}_{11}100011...$$

Markstein's iteration	**Goldschmidt's iteration**
$\tilde{x}_0 = \ 0.10101010101$	$\tilde{x}_0 = \ 0.10101010101$
$\tilde{r}_1 = -1.1101100010 \cdot 2^{-5}$	$\tilde{r}_1 = -1.11011000100 \cdot 2^{-5}$
$\tilde{x}_1 = \ 0.10100000110$	$\tilde{x}_1 = \ 0.10100000110$ // $\tilde{r}_2$...
$\tilde{r}_2 = \ 1.1100111010 \cdot 2^{-9}$	$\tilde{x}_2 = \ 0.10100001010$ // $\tilde{r}_3$...
$\tilde{x}_2 = \ 0.10100001011$	($\tilde{x}_n$ remains the same)

In the Goldschmidt iteration, $\tilde{r}_{n+2}$ and $\tilde{x}_{n+1}$ can be computed concurrently. Hence, this iteration is faster than Markstein's iterations on architectures providing parallel floating-point units, or pipelined FMAs. However, in this example, only the Markstein iteration yields the correct rounding. A common method [43] is to use Goldschmidt's iterations at the beginning, when accuracy is not an issue, and next to switch to Markstein's iterations if needed on the last iterations to get the correctly rounded result.

Concerning the division, one may consider several iterations derived from Equation (4.9). We only consider here the following ones:

$$\text{Markstein} \quad \begin{cases} \tilde{r}_{n+1} = \mathrm{RN}(a - b\tilde{y}_n) \\ \tilde{y}_{n+1} = \mathrm{RN}(\tilde{y}_n + \tilde{r}_{n+1}\tilde{x}_m) \end{cases} \tag{4.12}$$

$$\text{Goldschmidt} \quad \begin{cases} \tilde{r}_0 = \mathrm{RN}(a - b\tilde{y}_0) \\ \tilde{r}_{n+2} = \mathrm{RN}(\tilde{r}_{n+1}^2) \\ \tilde{y}_{n+1} = \mathrm{RN}(\tilde{y}_n + \tilde{r}_{n+1}\tilde{x}_m) \end{cases} \tag{4.13}$$

In this chapter, we will also consider different precisions:

- The input precision (p_i), which is the precision of the operands ($x, y \in \mathbb{F}_{\beta,p_i}$).
- The ouput precision (p_o), which is the precision we want to obtain for the correctly rounded quotient.
- The working precision (p_w), which is the precision at which the Newton-Raphson iterations are performed. This precision is always greater than p_i and p_o.

4.3 Faithful rounding

In some cases explained in Section 4.4, a faithful rounding is required in order to guarantee correct rounding of the quotient a/b. One may also only need a faithful rounding of the quotient or the reciprocal. This section provides a sufficient condition to ensure a faithful rounding of the quotient. We then remind the "exact residual theorem" (Theorem 4.3), that will be used for proving the correct rounding in Section 4.4.

4.3.1 Ensuring a faithful rounding

To prove that the last iteration yields a correct rounding, we use the fact that a faithful rounding has been computed. To prove that at some point, a computed approximation $\tilde{y}_n$ is a faithful rounding of the exact quotient a/b, we use a theorem similar to the one proposed by Rump in [53], adapted here to the general case of radix β.

Theorem 4.2. *Let $\hat{x} \in \mathbb{R}$ be an approximation to $z \in \mathbb{R}$. Let $\tilde{x} \in \mathbb{F}_{\beta,p}$ be such that $\tilde{x} = \mathrm{RN}(\hat{x})$. If*

$$|\hat{x} - z| < \frac{1}{2\beta}\mathrm{ulp}(z), \tag{4.14}$$

then $\tilde{x}$ is a faithful rounding of z.

The condition of Theorem 4.2 is tight: Assuming β is even, if $z = \beta^k$, then $\hat{x} = z - \frac{1}{2\beta}\mathrm{ulp}(z)$ will round to a value that is not a faithful rounding of z, as illustrated on Figure 4.3.

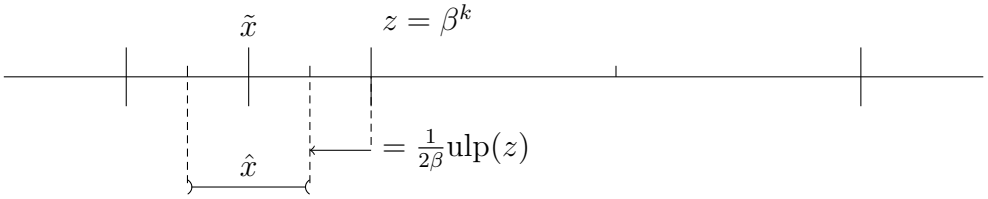


Figure 4.3: Tightness of the condition on $|\hat{x} - z|$

4.3.2 Exact residual theorem

When $\tilde{y}_n$ is a faithful rounding of a/b, the residual $\mathrm{RN}(a - b\tilde{y}_n)$ is computed exactly. The theorem was first stated by Markstein [42] and has been more recently proved by John Harrison [22] and Boldo and Daumas [4] using formal provers.

Theorem 4.3 (Exact residual for the division)**.** *Let a, b be two floating-point numbers in $\mathbb{F}_{\beta,p}$, and assume $\tilde{y}_n$ is a faithful rounding of a/b. For any rounding mode $\circ$, $\tilde{r}_{n+1} = \circ(a - b\tilde{y}_n)$ is computed exactly (without any rounding), provided there is no overflow or underflow.*

4.4 Round-to-nearest

In this section, we present several methods to ensure correct rounding. We first present a general method of exclusion intervals that only applies if the quotient a/b is not a midpoint, and how to extend the exclusion intervals in the case of reciprocal. We then show how to handle the midpoint cases separately.

4.4.1 Exclusion intervals

A common way of proving correct rounding for a given function in floating-point arithmetic is to study its exclusion intervals (see §1.5.1). Given $a, b \in \mathbb{F}_{\beta,p_i}$, either a/b is a midpoint at the output precision p_o, or there is a certain distance between a/b and the closest midpoint. Hence, if we assume that a/b is not a midpoint, then for any midpoint m, there exists a small interval centered at m that cannot contain a/b. Those intervals are called the *exclusion intervals*.

More formally, let us define $\mu_{p_i,p_o} > 0$ as the smallest value such that there exist $a, b \in \mathbb{F}_{\beta,p_i}$ and a midpoint m in precision p_o with $|a/b - m| = \beta^{e_{a/b}+1}\mu_{p_i,p_o}$. If a lower bound on μ_{p_i,p_o} is known, next Theorem 4.4 can be used to ensure correct rounding, as illustrated by Figure 4.4 (see [22] or [46, chap. 12] for a proof).

Theorem 4.4. *Let a, b in $\mathbb{F}_{\beta,p_i}$ be such that a/b is not midpoint in precision p_o for the division, and $\hat{y}$ be in $\mathbb{R}$. If $|\hat{y} - a/b| < \beta^{e_{a/b}+1}\mu_{p_i,p_o}$, then $\mathrm{RN}(\hat{y}) = \mathrm{RN}(a/b)$.*

To bound the radius of the exclusion intervals, we generalize the method used by Harrison [22] and Marius Cornea [10] to the case of radix β.

Theorem 4.5. *Assuming $p_o \geq 2$ and $p_i \geq 1$, a lower bound on μ_{p_i,p_o} is given by*

$$\mu_{p_i,p_o} \geq \frac{1}{2}\beta^{-p_i-p_o}. \tag{4.15}$$

Proof. By definition of μ_{p_i,p_o}, it can be proved that a/b is not a midpoint. Let m be the closest midpoint to a/b, and note $\delta\beta^{e_{a/b}+1}$ the distance between a/b and m:

$$\frac{a}{b} = m + \delta\beta^{e_{a/b}+1}. \tag{4.16}$$

By definition, μ_{p_i,p_o} is the smallest possible value of $|\delta|$. As we excluded the case when $a/b = m$, we have $\delta \neq 0$. We write $a = A\beta^{1-p_i}$, $b = B\beta^{1-p_i}$ and $m = (M + 1/2)\beta^{1-p_o+e_{a/b}}$, with A, B, M integers and $\beta^{p_i-1} \leq A, B, M < \beta^{p_i}$. Equation (4.16) becomes

$$2B\beta^{p_o}\delta = 2A\beta^{p_o-1-e_{a/b}} - 2BM - B. \tag{4.17}$$

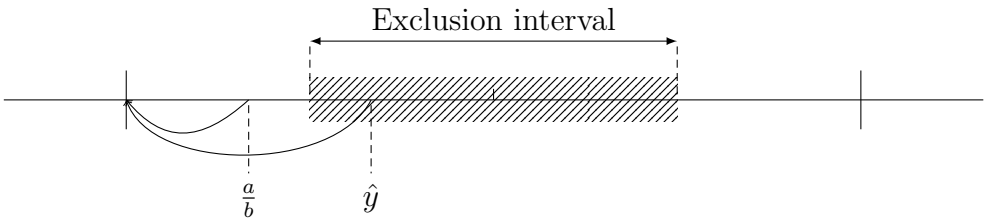


Figure 4.4: Use of exclusion intervals for proving the correct rounding

Since $2A\beta^{p_o-1-e_{a/b}} - 2BM - B$ is an integer and $\delta \neq 0$, we have $|2B\beta^{p_o}\delta| \geq 1$. Since $\beta^{p_i-1} \leq B < \beta^{p_i}$, the conclusion follows. □

In radix 2, the following example illustrates the sharpness of the result of Theorem 4.5.

Example 4.6. In radix 2, for any precision $p_i = p_o$, $b = 1.11\ldots1 = 2 - 2^{1-p_i}$ gives

$$\frac{1}{b} = \frac{1}{2} + 2^{-1-p_i} + \underbrace{\frac{2^{-1-2p_i}}{1-2^{-p_i}}}_{\delta\beta^{e_{a/b}+1}} = 0.\underbrace{100\ldots0}_{p_i \text{ bits}}\underbrace{100\ldots0}_{p_i \text{ bits}}1\ldots$$

From this example, an upper bound on μ_{p_i,p_i} can be deduced, and for any precision $p_i \geq 1$ one has

$$2^{-1-2p_i} \leq \mu_{p_i,p_i} \leq \frac{2^{-1-2p_i}}{1-2^{-p_i}}.$$

The following result can also be seen as a consequence of Theorem 4.4 (see [43, chap. 8] or [46, p.163] for a proof).

Theorem 4.7. *In binary arithmetic, when the working precision is the same as the ouput precision ($p_w = p_o$), if $\tilde{x}$ is a correct rounding of $1/b$ and $\tilde{y}$ is a faithful rounding of a/b, then an extra Markstein's iteration yields* $\mathrm{RN}_{p_o}(a/b)$.

4.4.2 Extending the exclusion intervals

When $p_w = p_o = p_i$, the error bounds of Section 4.5 might remain larger than the bound on the radius of the exclusion intervals of §4.4.1. A way to prove correct rounding is then by extending the exclusion intervals.

In this subsection, we describe a method to determine all the inputs $b \in \mathbb{F}_{\beta,p_i}$ such that the reciprocal $1/b$ is not a midpoint and lies within a distance $\beta^{e_{1/b}+1}\mu$ from the closest midpoint m. Once all such worst cases are determined, correct rounding can be guaranteed as can be seen on Figure 4.5 considering two cases:

- If $1/b$ corresponds to one of the worst cases, we then run the Newton-Raphson algorithm on the input b and check that the result is correct.

- If $(1, b)$ is not one of those worst cases and $\hat{x}$ is an approximation to $1/b$ that satisfies $|\hat{x} - 1/b| < \beta^{e_{1/b}+1}\mu$, then $\mathrm{RN}_{\beta,p_o}(\hat{x}) = \mathrm{RN}_{\beta,p_o}(1/b)$.

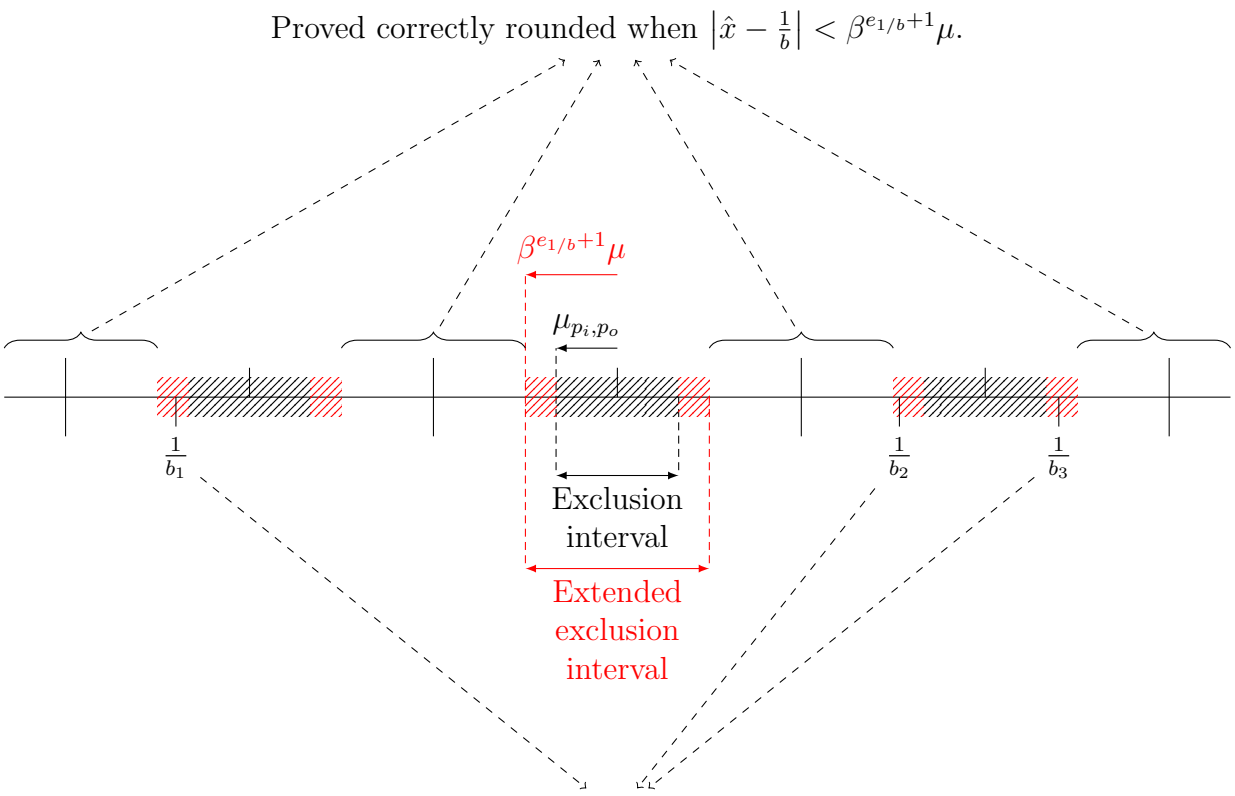


Figure 4.5: Proving the correct rounding using extended exclusion intervals

Unfortunately, there are too many worst cases to do the same for the division. Starting from Equation (4.17) of §4.4.1, one has:

$$2\beta^{p_i+p_o} - \underbrace{2B\beta^{p_o}\delta}_{\Delta} = B(2M+1). \tag{4.18}$$

Factorizing $2\beta^{p_i+p_o} - \Delta$, with $|\Delta| \in \{1, 2, \dots\}$ into $B(2M+1)$ with respect to the range of these integral significands isolates the worst cases. After finding all the worst cases such that $|\Delta| < n$, the extended radius is such that $\mu \geq \beta^{-p_i-p_o}n/2$. Table 4.1 shows how many values of b have to be checked to extend the exclusion interval.

n	2	3	4	5	6	7
binary64	2	68	68	86	86	86
decimal128	1	1	3	3	19	22

Table 4.1: Number of b to check separately according to the extended radius of the exclusion interval $\mu \geq \beta^{-p_i-p_o}n/2$

There is a particular worst case that is worth mentioning: When $b = \beta - 1/2\,\mathrm{ulp}(\beta)$, the correctly rounded reciprocal is $1/\beta + \mathrm{ulp}(1/\beta)$, but if the starting point given by the lookup table is not $\mathrm{RN}_{\beta,p_w}(1/b)$, even Markstein's iterations cannot give a correct rounding, as shown in Example 4.8.

Example 4.8. In binary16, ($p_w = p_i = p_o = 11, \beta = 2$):

$$\begin{aligned} b &= 1.1111111111 \\ \tilde{x} &= 0.1 = \text{Table-lookup}(b) \\ \tilde{r} &= 2^{-11} \text{ (exact)} \\ \tilde{x} &= 0.1 \end{aligned}$$

Hence, $\tilde{x}$ will always equals 0.1, which is not the correct rounding of RN($1/b$).

A common method [43] to deal with this case is to tweak the lookup table for this value. If the lookup table is addressed by the first k digits of b, then the output corresponding to the address $\beta - \beta^{1-k}$ should be $1/\beta + \beta^{-p_w}$.

4.4.3 The midpoint case

Theorem 4.4 can only be used to ensure correct rounding when a/b cannot be a midpoint. In this subsection, we summarize our results about the midpoints for division and reciprocal that were already presented in Sections 3.5 and 3.6.

Midpoints in radix 2

In radix 2, a/b cannot be a midpoint in a precision greater or equal to p_i. However, Example 4.9 shows that when $p_i > p_o$, a/b can be a midpoint.

Example 4.9. $p_i = 24, p_0 = 11, \beta = 2$ (Inputs in binary32 and output in binary16)

$$\begin{aligned} a&=1.00000001011001010011111 \\ b&=1.00101100101000000000000 \\ a/b&=0.\underbrace{11011011001}_{p_o=11}1 \end{aligned}$$

When computing reciprocals, we have the following.

Theorem 4.10. *In radix 2, and for any precisions p_i and p_o, the reciprocal of a floating-point number in $\mathbb{F}_{2,p_i}$ cannot be a midpoint in precision p_o.*

Midpoints in radix 10

In decimal floating-point arithmetic, the situation is quite different. As in radix 2, there are cases where a/b is a midpoint in precision p_o, but they can occur even when $p_i = p_o$, as shown in Example 4.11. Contrarily to the binary case, there are also midpoints for the reciprocal function, characterized by Theorem 4.12.

Example 4.11. $p_i = p_o = 7, \beta = 10$ (inputs and output are in decimal32)

$$a = 2.000005, \quad b = 2.000000, \quad a/b = 1.0000025$$

Theorem 4.12. *Let $y \in \mathbb{F}_{10,p}$ be nonzero. One has $1/y \in \mathbb{M}_{10,p}$ if and only if the integral significand Y of y has the form*

$$Y = 2^{2p} \cdot 5^{2p-1-\ell}, \tag{4.19}$$

with $\ell \in \mathbb{N}$ such that $2 \cdot 10^{p-1} < 5^\ell < 2 \cdot 10^p$.

Using Theorem 4.12, we isolate the at most two values of b whose reciprocal is a midpoint. These values are checked separately when proving the correct rounding of the reciprocal. Table 4.2 gives the corresponding b when $p_i = p_o$, for the IEEE 754-2008 decimal formats.

	decimal32	**decimal64**	**decimal128**
p	7	16	34
b_1	2.048000	1.67772160...0	1.12589990684262400...0
b_2	2.048000	8.38860800...0	5.62949953421312000...0

Table 4.2: Decimal floating-point numbers whose reciprocal is a midpoint in the same precision

4.4.4 Correctly handling midpoint cases

Let us recall that the midpoints cases for reciprocal can be handled as explained in §4.4.3. Hence, we only focus here on division.

When p_i, p_o and the radix are such that division admits midpoints, the last Newton-Raphson iteration must be adapted to handle the case where a/b is a midpoint. We propose two methods, depending whether $p_w = p_o$. Both methods rely on the exact residual theorem 4.3 of §4.3.2, so it is necessary to use a Markstein iteration (4.12) for the last iteration.

When $p_w > p_o$

The exclusion interval theorem of §4.4.1 does not apply, since there are several cases where a/b is a midpoint in precision p_o. In that case, we use the following Theorem 4.13 instead of Theorem 4.4.

Theorem 4.13. *We assume β is even and $p_w > p_o$, and we perform a Markstein iteration:*

$$\begin{cases} \tilde{r} = \mathrm{RN}_{\beta,p_w}(a - b\tilde{y}), \\ \tilde{y}' = \mathrm{RN}_{\beta,p_o}(\tilde{y} + \tilde{r}\tilde{x}). \end{cases}$$

If $\tilde{y}$ is a faithful rounding of a/b in precision p_w, and $|\hat{y}' - a/b| < \beta^{e_{a/b}+1}\mu_{p_i,p_o}$, then $\tilde{y}' = \mathrm{RN}_{\beta,p_o}(a/b)$.

Proof of theorem 4.13. If a/b is not a midpoint in precision p_o, Theorem 4.4 proves that $\tilde{y}'$ is the correct rounding of a/b. Now, we assume that a/b is a midpoint in precision p_o. Since β is even and $p_w > p_o$, a/b is a floating-point number in precision p_w. Since $\tilde{y}$ is a faithful rounding of a/b in precision p_w, we have $\tilde{y} = a/b$. Using Theorem 4.3, we know that $\tilde{r} = 0$, which gives $\hat{y}' = a/b$, hence $\tilde{y}' = \mathrm{RN}_{\beta,p_o}(\hat{y}') = \mathrm{RN}_{\beta,p_o}(a/b)$. □

Example 4.14 shows why it is important to round directly in precision p_o in the last iteration.

Example 4.14. $p_i = 24, p_o = 11, \beta = 2$ (inputs are in binary32 and output is in binary16)

$$\begin{aligned}
&a{=}1, b = 1.01010011001111000011011\\
&\tilde{y}{=}0.11000001001011111111111111 \text{ (faithful)}\\
&\tilde{r}{=}1.01010011000111000011011 \cdot 2^{-24} \text{ (exact)}\\
&\tilde{y}'{=}0.\underbrace{11000001001}_{\text{11 bits}}1000000000000 = \mathrm{RN}_{24}(\tilde{y} + \tilde{r}\tilde{y})\\
&\tilde{y}''{=}0.11000001010 = \mathrm{RN}_{11}(\tilde{y}')
\end{aligned}$$

Due to the double rounding, $\tilde{y}''$ is not $\mathrm{RN}_{11}(a/b)$.

When $p_w = p_o$

The quotient a/b cannot be a midpoint in radix 2. For decimal arithmetic, Example 4.15 suggests that it is not possible in this case to round correctly using only Markstein's iterations: We know from Theorem 4.3 that the residuals $\tilde{r}_1$ and $\tilde{r}_2$ are computed exactly. However, the Markstein's iteration oscillates between the two faithful roundings, which means that the Newton-Raphson method does not converge to the correct rounding.

Example 4.15. In decimal32 ($p_i = p_o = p_w = 7, \beta = 10$):

$$\begin{gathered}
a = 6.000015,\ b = 6.000000\\
\tilde{x} = \mathrm{RN}(1/b) = 0.1666667,\ a/b = 1.000002\,5
\end{gathered}$$

$$\begin{aligned}
\tilde{y}_0 &= \mathrm{RN}(a\tilde{x}) &&= \ 1.000003\\
\tilde{r}_1 &= \mathrm{RN}(a - b\tilde{y}_0) &&= -0.000003\\
\tilde{y}_1 &= \mathrm{RN}(\tilde{y}_0 + \tilde{r}_1\tilde{x}) &&= \ 1.000002\\
\tilde{r}_2 &= \mathrm{RN}(a - b\tilde{y}_1) &&= \ 0.000003\\
\tilde{y}_2 &= \mathrm{RN}(\tilde{y}_1 + \tilde{r}_2\tilde{x}) &&= \ 1.000003
\end{aligned}$$

Algorithm 6 can be used in this case to determine the correct rounding of a/b from a faithfully rounded approximation.

Theorem 4.16. *Let us assume that $\beta = 10$ and $p_w = p_o$ and that $\tilde{y}$ is a faithful rounding of a/b. Then, Algorithm 6 yields the correct rounding of a/b.*

Proof. By assumption, $\tilde{y}$ is a faithful rounding of a/b. Thus, there exists ϵ such that $-\mathrm{ulp}(a/b) < \epsilon < \mathrm{ulp}(a/b)$ and $\tilde{y} = a/b + \epsilon$. Also, according to Theorem 4.3, $\tilde{r} = -b\epsilon$. Six cases, depending on the signs of $\tilde{r}$ and c, have to be considered for the whole proof, as depicted in Figure 4.6. We only present here two cases, the others being similar.

• Case $\tilde{r} \geq 0$ and $2\tilde{r} - b\,\mathrm{ulp}(a/b) < 0$: Since $\tilde{r}$ is positive, $-\epsilon \leq 0$. Moreover, since $2\tilde{r} - b\,\mathrm{ulp}(a/b) < 0$ we have $-1/2\,\mathrm{ulp}(a/b) < \epsilon < 0$. Hence, the correct rounding of a/b is $\tilde{y}$.

• Case $\tilde{r} < 0$ and $2\tilde{r} + b\,\mathrm{ulp}(a/b) = 0$: From $2\tilde{r} + b\,\mathrm{ulp}(a/b) = 0$, we deduce that a/b is a midpoint and $\mathrm{RN}(a/b) = \mathrm{RN}(\tilde{y} - 1/2\,\mathrm{ulp}(a/b))$. □

$b_s = b\,\mathrm{ulp}(a/b)$; /* $b_s \in \mathbb{F}_{\beta,p_w}$ */
/* Assume $\tilde{y}$ faithful */
$\tilde{r} = a - b\tilde{y}$; /* $\tilde{r}$ exactly computed. */
if $\tilde{r} > 0$ **then**
  $c = \mathrm{RN}(2\tilde{r} - b_s)$;
  if $c = 0$ **then return** $\mathrm{RN}(\tilde{y} + \frac{1}{2}\,\mathrm{ulp}(a/b))$;
  if $c < 0$ **then return** $\tilde{y}$;
  if $c > 0$ **then return** $\tilde{y} + \mathrm{ulp}(a/b)$;
else /* $\tilde{r} \leq 0$ */
  $c = \mathrm{RN}(2\tilde{r} + b_s)$;
  if $c = 0$ **then return** $\mathrm{RN}(\tilde{y} - \frac{1}{2}\,\mathrm{ulp}(a/b))$;
  if $c < 0$ **then return** $\tilde{y} - \mathrm{ulp}(a/b)$;
  if $c > 0$ **then return** $\tilde{y}$;
end

Algorithm 6: Returning the correct rounding in decimal arithmetic when $p_w = p_o$.

4.5 Error bounds

In this section, we present the techniques we used to bound the error in the approximation to the reciprocal $1/b$ or to the quotient a/b obtained after a series of Newton-Raphson iterations. As our aim is to analyze any reasonable sequence combining both Markstein's or Goldschmidt's iterations, we only give the basic results needed to analyze one step of these iterations. The analysis of a whole sequence of iterations can be obtained by combining the induction relations proposed here: This is a kind of *running error analysis* (see [46, chap. 6]) that can be used together with the results of §§4.4.1 and 4.4.2 to ensure correct rounding.

All the arithmetic operations are assumed to be performed at precision p_w, which is the precision used for intermediate computations. Let us denote by ϵ the unit roundoff: In round-to-nearest rounding mode, one has $\epsilon = \frac{1}{2}\beta^{1-p_w}$. In the following, we note

$$\begin{array}{ll} \hat{\phi}_n := |\hat{x}_n - 1/b|, & \tilde{\phi}_n := |\tilde{x}_n - 1/b|, \\ \hat{\psi}_n := |\hat{y}_n - a/b|, & \tilde{\psi}_n := |\tilde{y}_n - a/b|, \\ \tilde{\rho}_n := |\tilde{r}_n - (1 - b\tilde{x}_{n-1})|, & \tilde{\sigma}_n := |\tilde{r}_n - (a - b\tilde{y}_{n-1})|. \end{array}$$

4.5.1 Reciprocal iterations

Both for Markstein's iteration (4.10) and for Goldschmidt's iteration (4.11), the absolute error $\hat{\phi}_n$ in the approximation $\hat{x}_n$ is bounded as

$$\hat{\phi}_{n+1} \leq (\tilde{\phi}_n + |1/b|)\tilde{\rho}_{n+1} + |b|\tilde{\phi}_n^2, \tag{4.20}$$

$$\tilde{\phi}_{n+1} \leq (1+\epsilon)\hat{\phi}_{n+1} + |\epsilon/b|. \tag{4.21}$$

Hence it just remains to obtain induction inequalities for bounding $\tilde{\rho}_{n+1}$.

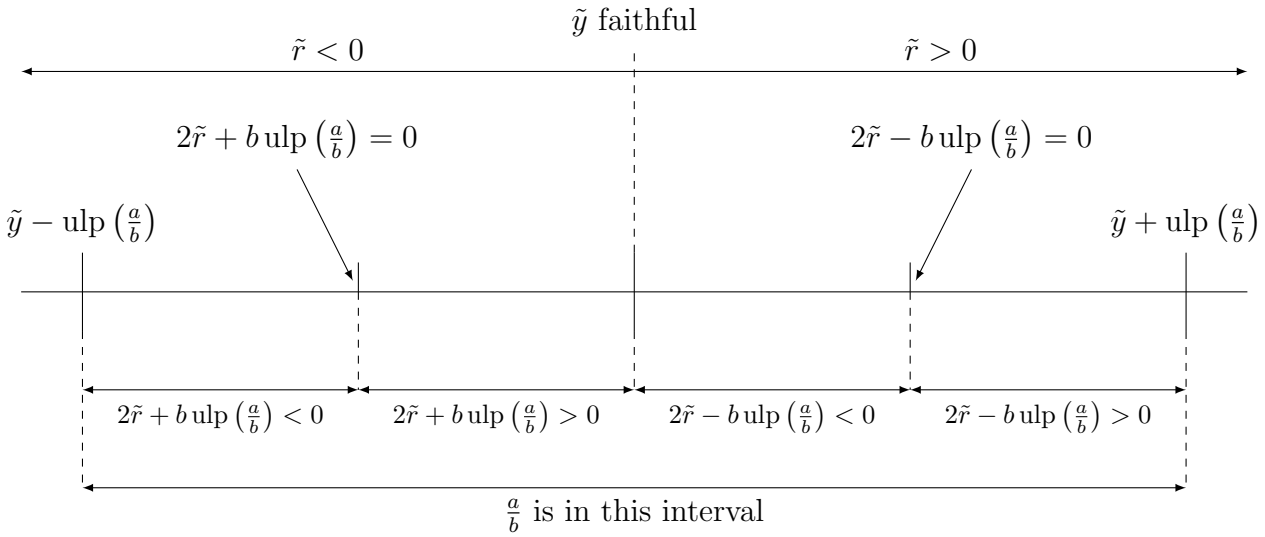


Figure 4.6: The different cases for Algorithm 6, when $\tilde{y}$ is a faithful rounding of $\frac{a}{b}$.

Reciprocal with the Markstein iteration (4.10)

One has $\tilde{r}_{n+1} = \mathrm{RN}(1 - b\tilde{x}_n)$, hence

$$\tilde{\rho}_{n+1} \leq |\epsilon||b|\tilde{\phi}_n. \tag{4.22}$$

The initial value of the recurrence depends on the lookup-table used for the first approximation to $1/b$. Inequality (4.20) together with (4.22) can then be used to ensure either faithful or correct rounding for all values of b in $[1, \beta)$, using Theorems 4.2 or 4.4.

At iteration n, if $\tilde{x}_n$ is a faithful rounding of $1/b$, then Theorem 4.3 implies $\tilde{\rho}_{n+1} = 0$. Hence in this case one has $\hat{\phi}_{n+1} \leq \tilde{\phi}_n^2$, which means that no more accuracy improvement can be expected with Newton-Raphson iterations. Moreover, if we exclude the case $b = 1$, since b belongs to $[1, \beta)$ by hypothesis, it follows that $1/b$ is in $(\beta^{-1}, 1)$. Since $\tilde{x}_n$ is assumed to be a faithful rounding of $1/b$, one has $\mathrm{ulp}(\tilde{x}_n) = \mathrm{ulp}(1/b)$, and we deduce

$$\tilde{\phi}_{n+1} \leq |b|\tilde{\phi}_n^2 + 1/2\,\mathrm{ulp}(\tilde{x}_n), \tag{4.23}$$

which gives a sharper error bound on $\tilde{\phi}_{n+1}$ than (4.20) when $\tilde{x}_n$ is a faithful rounding of $1/b$.

Reciprocal with the Goldschmidt iteration (4.11)

For the Goldschmidt iteration, one has

$$\tilde{\rho}_{n+1} \leq (1 + \epsilon)\left(\tilde{\rho}_n + |b|\tilde{\phi}_{n-1} + 1\right)\tilde{\rho}_n + \epsilon. \tag{4.24}$$

Combining (4.24) into (4.21), one can easily deduce a bound on the error $\hat{\phi}_{n+1}$.

4.5.2 Division iterations

Both for Markstein's iteration (4.12) and for Goldschmidt's iteration (4.13), one may check that

$$\hat{\psi}_{n+1} \leq |b|\tilde{\psi}_n\tilde{\phi}_m + (\tilde{\phi}_m + |1/b|)\tilde{\sigma}_{n+1}, \tag{4.25}$$

$$\tilde{\psi}_{n+1} \leq (1+\epsilon)\hat{\psi}_{n+1} + \epsilon|a/b|. \tag{4.26}$$

Now let us bound $\tilde{\sigma}_{n+1}$.

Division with the Markstein iteration (4.12)

In this case, one has

$$\tilde{\sigma}_{n+1} \leq \epsilon|b|\tilde{\psi}_n. \tag{4.27}$$

Again, if $\tilde{y}_n$ is a faithful rounding of a/b, due to the exact residual theorem 4.3, one has $\hat{\psi}_{n+1} \leq |b|\tilde{\psi}_n\tilde{\phi}_m$. This corresponds exactly to the error bound due to the mathematical method of Newton-Raphson iterations. This means that it is the best accuracy improvement that can be expected from one Newton-Raphson iteration in floating-point arithmetic.

Division with the Goldschmidt iteration (4.12)

Using the same method as in §4.5.1, we now bound $\tilde{\sigma}_{n+1}$:

$$\tilde{\sigma}_{n+1} \leq (1+\epsilon)(\tilde{\sigma}_n + |b|\tilde{\psi}_{n-1})(|b|\tilde{\psi}_{n-1} + |b|\tilde{\phi}_m + \tilde{\sigma}_n) + (1+\epsilon)\tilde{\sigma}_n + \epsilon|a|.$$

Then, from (4.25), a bound on $\hat{\psi}_{n+1}$ can be obtained.

4.6 Experiments

Using the induction relations of Section 4.5, one can bound the error on the approximations to a/b for a given series of Newton-Raphson iterations, and use it with the sufficient conditions presented in Section 4.4 to ensure correct rounding. In this section, we will denote by MR and GR the Markstein (4.10) and Goldschmidt (4.11) iterations for computing the reciprocal, and by MD and GD the Markstein (4.12) and Goldschmidt (4.13) iterations for the division.

Let us consider three examples : Algorithms 7, 8 and 9 below. The certified error on $\hat{x}$ and $\hat{y}$ for those algorithms is displayed on Figure 4.7.

Algorithm 7 computes the quotient of two binary128 ($p_i = 113$) numbers, the output being correctly rounded to binary64 ($p_o = 53$). The internal format used for the computations is also binary128 ($p_w = 113$). Since $p_i > p_o$, there are midpoints for division, as stated in §4.4.3. After the MD_1 iteration, we know from Theorem 4.2 that $\tilde{y}$ is a faithful rounding of a/b, as shown in Figure 4.7(a). An extra Markstein's iteration gives an error on $\hat{y}$ that is smaller than the radius of the exclusion interval $\beta^{e_{a/b}+1}\mu_{113,53}$, as illustrated by Figure 4.7(a). Hence, Theorem 4.13 of §4.4.4 applies and guarantees that Algorithm 7 yields a correct rounding of the division, even for the midpoint cases.

Algorithm 8 computes the quotient of two binary64 numbers, with $p_i = p_w = p_o = 53$. Since binary arithmetic is used and $p_w = p_o$, there are no midpoints for division. After the MR_4 iteration, $\tilde{x}$ is less than $2 \cdot \beta^{e_{1/b}+1}\mu_{53,53}$. Hence, by excluding two worst cases as

$$
\begin{array}{lll}
\tilde{x} = \text{Table-lookup}(b); & \{\text{Error less than } 2^{-8}\} & \\
\tilde{r} = \mathrm{RN}_{113}(1 - b\tilde{x}); & & \\
\tilde{x} = \mathrm{RN}_{113}(\tilde{x} + \tilde{r}\tilde{x}); & \{\mathrm{MR}_1\} \;||\; & \tilde{r} = \mathrm{RN}_{113}(\tilde{r}^2); \\
\tilde{x} = \mathrm{RN}_{113}(\tilde{x} + \tilde{r}\tilde{x}); & \{\mathrm{GR}_2\} \;||\; & \tilde{r} = \mathrm{RN}_{113}(\tilde{r}^2); \\
\tilde{x} = \mathrm{RN}_{113}(\tilde{x} + \tilde{r}\tilde{x}); & \{\mathrm{GR}_3\} & \\
\tilde{y} = \mathrm{RN}_{113}(a\tilde{x}); & \{y_0\} & \\
\tilde{r} = \mathrm{RN}_{113}(a - b\tilde{y}); & & \\
\tilde{y} = \mathrm{RN}_{113}(\tilde{y} + \tilde{r}\tilde{x}); & \{\mathrm{MD}_1\} & \\
\tilde{r} = \mathrm{RN}_{113}(a - b\tilde{y}); & & \\
\tilde{y} = \mathrm{RN}_{53}(\tilde{y} + \tilde{r}\tilde{x}); & \{\mathrm{MD}_2\} &
\end{array}
$$

Algorithm 7: Computing the quotient of two binary128 numbers, output in binary64.

$$
\begin{array}{lll}
\tilde{x} = \text{Table-lookup}(b); & \{\text{Error less than } 2^{-8}\} & \\
\tilde{r} = \mathrm{RN}_{53}(1 - b\tilde{x}); & & \\
\tilde{x} = \mathrm{RN}_{53}(\tilde{x} + \tilde{r}\tilde{x}); & \{\mathrm{MR}_1\} \;||\; & \tilde{r} = \mathrm{RN}_{53}(\tilde{r}^2); \\
\tilde{x} = \mathrm{RN}_{53}(\tilde{x} + \tilde{r}\tilde{x}); & \{\mathrm{GR}_2\} & \\
\tilde{r} = \mathrm{RN}_{53}(1 - b\tilde{x}); & & \\
\tilde{x} = \mathrm{RN}_{53}(\tilde{x} + \tilde{r}\tilde{x}); & \{\mathrm{MR}_3\} & \\
\tilde{r} = \mathrm{RN}_{53}(1 - b\tilde{x}); & & \\
\tilde{x} = \mathrm{RN}_{53}(\tilde{x} + \tilde{r}\tilde{x}); & \{\mathrm{MR}_4\} & \\
\tilde{y} = \mathrm{RN}_{53}(a\tilde{x}); & \{y_0\} & \\
\tilde{r} = \mathrm{RN}_{53}(a - b\tilde{y}); & & \\
\tilde{y} = \mathrm{RN}_{53}(\tilde{y} + \tilde{r}\tilde{x}); & \{\mathrm{MD}_1\} & \\
\tilde{r} = \mathrm{RN}_{53}(a - b\tilde{y}); & & \\
\tilde{y} = \mathrm{RN}_{53}(\tilde{y} + \tilde{r}\tilde{x}); & \{\mathrm{MD}_2\} &
\end{array}
$$

Algorithm 8: Computing the quotient of two binary64 numbers, output in binary64.

explained in §4.4.2, and checking those cases, we ensure a correct rounding of the reciprocal using Theorem 4.4. Since a faithful rounding of a/b at iteration MD_1 is ensured by the error bounds of Section 4.5, Theorem 4.7 proves that the next Markstein's iteration outputs a correct rounding.

Algorithm 9 computes the quotient of two decimal128 numbers, with $p_i = p_w = p_o = 34$. The starting error given by the lookup table is less than $5 \cdot 10^{-5}$. Since $p_w = p_o$, Algorithm 6 is needed to ensure the correct rounding of the division. Notice that to improve the latency, b_s in Algorithm 6 can be computed concurrently with the first Newton-Raphson iterations. As shown in Figure 4.7(c), $\tilde{y}$ is a faithful rounding after the MD_1 iteration. Hence, Theorem 4.16 ensures correct rounding for Algorithm 9.

4.7 Conclusion

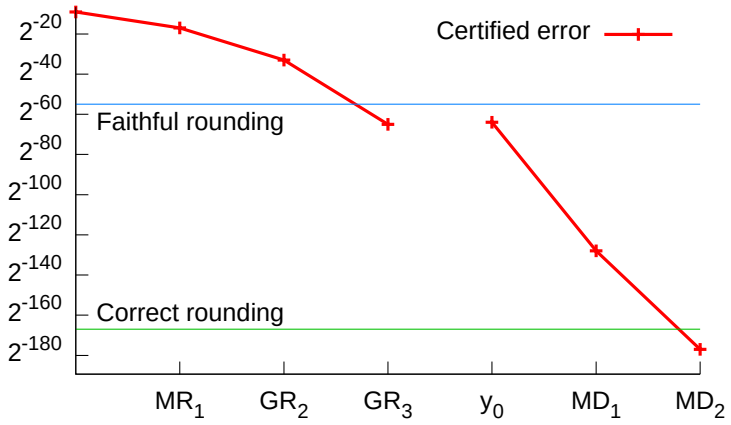
We gave general methods of proving correct rounding for division algorithms based on Newton-Raphson's iterations, for both binary and decimal arithmetic. Performing the division in decimal arithmetic of two floating-point numbers in the working precision seems

$\tilde{x} = \text{Table-lookup}(b)$; $\{\text{Error less than } 5 \cdot 10^{-5}\}$
$\tilde{r} = \text{RN}_{34}(1 - b\tilde{x})$; || $b_s = b\,\text{ulp}(\frac{a}{b})$;
$\tilde{x} = \text{RN}_{34}(\tilde{x} + \tilde{r}\tilde{x})$; $\{\text{MR}_1\}$ || $\tilde{r} = \text{RN}_{34}(\tilde{r}^2)$;
$\tilde{x} = \text{RN}_{34}(\tilde{x} + \tilde{r}\tilde{x})$; $\{\text{GR}_2\}$ || $\tilde{r} = \text{RN}_{34}(\tilde{r}^2)$;
$\tilde{x} = \text{RN}_{34}(\tilde{x} + \tilde{r}\tilde{x})$; $\{\text{GR}_3\}$
$\tilde{y} = \text{RN}_{34}(a\tilde{x})$; $\{y_0\}$
$\tilde{r} = \text{RN}_{34}(a - b\tilde{y})$;
$\tilde{y} = \text{RN}_{34}(\tilde{y} + \tilde{r}\tilde{x})$; $\{\text{MD}_1\}$
$\tilde{r} = \text{RN}_{34}(a - b\tilde{y})$;
Call **Algorithm 6**.

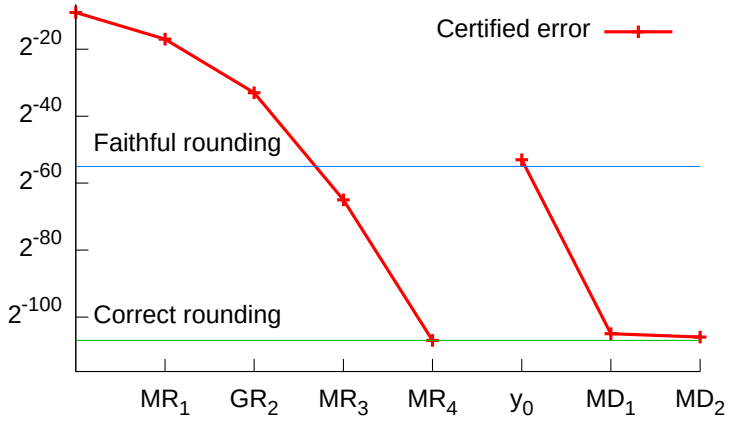
Algorithm 9: Computing the quotient of two decimal128 numbers, output in decimal128.

to be costly, and we recommend to always use a higher internal precision than the precision of inputs.

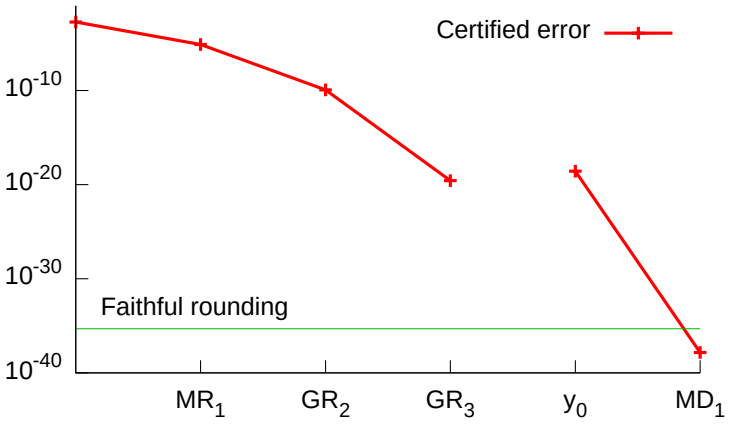
We only considered the round-to-nearest rounding mode. To achieve correct rounding in other rounding modes, only the last iteration of the Newton-Raphson algorithm has to be changed, whereas all the previous computations should be done in the round-to-nearest mode.



(a) Algorithm 7 (binary128)



(b) Algorithm 8 (binary64)



(c) Algorithm 9 (decimal128)

Figure 4.7: Absolute error before rounding for each algorithm considered. (M/G: Markstein/Goldschmidt, R/D: reciprocal/division)

Conclusion

In Chapter 2, we analyzed the RN-codings, a general class of number representations for which truncation of a digit string yields the effect of rounding to nearest, avoiding double-rounding issues. We were able to provide sensible arithmetic based on an efficient encoding of binary RN-coding, for both fixed point and floating-point binary RN-coding.

It is not known yet if it is possible to develop such an arithmetic on other radices RN-coding. For example, it is not known yet if it is possible to have an efficient addition for RN-coded decimal. Also, it may not be possible to have an efficient arithmetic for odd radices.

In Chapter 3, we obtained useful information on the existence of midpoints and exact points for several simple algebraic functions ($\sqrt{y}$, $1/\sqrt{y}$, x^k for $k \in \mathbb{N}_{>0}$, $x/\|y\|_2$, x/y, $1/y$, $1/\sqrt{x^2+y^2}$, $x/\sqrt{x^2+y^2}$). This information can be used for simplifying or improving the performance of programs that evaluate these functions.

The next step for a fast implementation of these functions would be to determine how close can $f(x)$ be to a breakpoint, excluding the cases when $f(x)$ is a breakpoint. Knowing these worst cases would allow one to know in advance at which precision $f(x)$ needs to be approximated in order to ensure a correct rounding.

Such study could also be extended to other functions. For example, for $d \geq 3$, we do not know yet whether the d-dimensional normalization function[2] admits midpoints in decimal, or if it admits exact points in binary. Notice however that the case of midpoints in binary was already covered in Section 3.4, and that the 2-dimensional normalization function admits many exact points in decimal, meaning that this will still hold for higher dimensions.

In Chapter 4, we gave general methods for proving correct rounding for division algorithms based on Newton-Raphson's iterations, both for binary and decimal arithmetic. In decimal arithmetic, performing the division of two floating-point numbers using only the working precision seems to be costly, and we recommend to always use an internal precision higher than the precision of inputs.

To guarantee that a sequence of Newton-Raphson iterations yields a correctly rounded quotient, it would be ideal to prove it using a formal proof checker. This would provide more confidence to processor designers for using Newton-Raphson based algorithms for implementing floating-point division.

The new requirement for a correctly rounded FMA in the IEEE 754-2008 Standard encourages processor manufacturers to implement an FMA in hardware in new architectures. Our work on Newton-Raphson algorithms can be used to exploit these FMAs in a software implementation of correctly rounded division. Speed and throughput of a Newton-Raphson division algorithm vary depending on how many FMA units are available, and

[2]The function that maps a d-dimensional x to $\frac{x_i}{\sqrt{\sum_{i=1}^{d} x_i^2}}$.

their throughput. It would be interesting to implement various Newton-Raphson iterations to see how they compare on a given architecture. It would also be interesting to implement Newton-Raphson algorithms on different architectures, to compare them with SRT implemented in hardware. SRT algorithms having a linear convergence, we think that Newton-Raphson algorithms should become clearly faster than SRT algorithms at higher precisions, such as 128 bits floating-point formats.

The work presented here shows that it is possible to design programs that automatically create a tailored Newton-Raphson algorithm, based on architectural constraints. Such programs may also generate algorithms for special cases: For example, if one needs only to compute $1/x$, it is possible to design a simpler Newton-Raphson algorithm for this special case. Knowing in advance that the inputs are normal floating-point numbers and do not lead to subnormals could also speed up the division algorithm.

Bibliography

[1] A. Avizienis. Signed-digit number representations for fast parallel arithmetic. *IRE Transactions on electronic computers*, 10:389–400, 1961. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 2, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[2] J.-C. Bajard, J. Duprat, S. Kla, and J.-M. Muller. Some operators for on-line radix 2 computations. *Journal of Parallel and Distributed Computing*, 22(2):336–345, August 1994.

[3] G. A. Baker. *Essentials of Padé Approximants*. Academic Press, New York, NY, 1975.

[4] S. Boldo and M. Daumas. Representable correcting terms for possibly underflowing floating point operations. In J.-C. Bajard and M. Schulte, editors, *Proceedings of the 16th Symposium on Computer Arithmetic*, pages 79–86, Santiago de Compostela, Spain, 2003.

[5] S. Boldo and G. Melquiond. Emulation of a FMA and correctly-rounded sums: proved algorithms using rounding to odd. *IEEE Transactions on Computers*, 57(4), April 2008.

[6] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[7] P. E. Ceruzzi. The early computers of Konrad Zuse, 1935 to 1945. *Annals of the History of Computing*, 3(3):241–262, 1981.

[8] M. Cornea, C. Anderson, J. Harrison, P.T.P. Tang, E. Schneider, and C. Tsen. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. In P. Kornerup and J.-M. Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic (ARITH-18)*, pages 29–37. IEEE Computer Society Conference Publishing Services, June 2007.

[9] M. Cornea, J. Harrison, and P.T.P. Tang. *Scientific Computing on Itanium-Based Systems*. Intel Press, 2002.

[10] M. Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide, and remainder algorithms. *Intel Technology Journal*, (Q2):11, 1998. Available at `ftp://download.intel.com/technology/itj/q21998/pdf/ieee.pdf`.

[11] C. Daramy-Loirat, D. Defour, F. de Dinechin, M. Gallet, N. Gast, C. Q. Lauter, and J.-M. Muller. CR-LIBM, a library of correctly-rounded elementary functions in double-precision. Technical report, LIP Laboratory, Arénaire team, Available at `https://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf`, December 2006.

[12] D. DaSarma and D. W. Matula. Measuring the accuracy of ROM reciprocal tables. *IEEE Transactions on Computers*, 43(8):932–940, August 1994.

[13] M. D. Ercegovac and T. Lang. *Division and Square Root: Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, MA, 1994.

[14] G. Even, P.-M. Seidel, and W. E. Ferguson. A parametric error analysis of Goldschmidt's division algorithm. *Journal of Computer and System Sciences*, 70(1):118–139, 2005.

[15] G. Everest and T. Ward. *An Introduction to Number Theory*. Graduate Texts in Mathematics. Springer-Verlag, London, 2005.

[16] P. M. Farmwald. *On the design of high performance digital arithmetic units*. PhD thesis, Stanford, CA, USA, 1981. AAI8201985.

[17] S. A. Figueroa. When is double rounding innocuous? *ACM SIGNUM Newsletter*, 30(3), July 1995.

[18] S. A. Figueroa. *A Rigorous Framework for Fully Supporting the IEEE Standard for Floating-Point Arithmetic in High-Level Programming Languages*. PhD thesis, Department of Computer Science, New York University, 2000.

[19] R. E. Goldschmidt. Applications of division by convergence. Master's thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA, USA, June 1964.

[20] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, second edition, 1994.

[21] A. Guyot, Y. Herreros, and J.-M. Muller. JANUS, an on-line multiplier/divider for manipulating large numbers. In *Proceedings of the 9th IEEE Symposium on Computer Arithmetic*, pages 106–111. IEEE Computer Society Press, Los Alamitos, CA, 1989.

[22] J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 234–251. Springer-Verlag, 2000.

[23] J. Harrison. Decimal transcendentals via binary. In J. D. Bruguera, M. Cornea, D. DasSarma, and J. Harrison, editors, *Proceedings of the 19th IEEE Symposium on Computer Arithmetic*, pages 187–194, Portland, OR, 2009. IEEE Computer Society.

[24] N. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition*. SIAM, Philadelphia, PA, 2002.

[25] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[26] IEEE Standard for Radix Independent Floating-Point Arithmetic. *ANSI/IEEE Standard, Std 854-1987,* New York, 1987.

[27] IEEE Standard for Floating-Point Arithmetic. *IEEE Standard, Std 754-2008,* New York, 2008.

[28] C. Iordache and D. W. Matula. On infinitely precise rounding for division, square root, reciprocal and square root reciprocal. In I. Koren and P. Kornerup, editors, *Proceedings of the 14th IEEE Symposium on Computer Arithmetic (Adelaide, Australia)*, pages 233–240. IEEE Computer Society Press, Los Alamitos, CA, April 1999.

[29] C.-P. Jeannerod, N. Louvet, J.-M. Muller, and A. Panhaleux. Midpoints and exact points of some algebraic functions in floating-point arithmetic. *IEEE Transactions on Computers*, 60(2):228–241, February 2011.

[30] W. Kahan. Why do we need a floating-point standard? Technical report, Computer Science, UC Berkeley, 1981. Available at `http://www.cs.berkeley.edu/~wkahan/ieee754status/why-ieee.pdf`.

[31] W. Kahan. Lecture notes on the status of IEEE-754. Available at `http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF`, 1996.

[32] W. Kahan. A logarithm too clever by half. Available at `http://http.cs.berkeley.edu/~wkahan/LOG10HAF.TXT`, 2004.

[33] P. Kornerup. Correcting the normalization shift of redundant binary representations. *IEEE Transactions on Computers*, 58:1435–1439, October 2009.

[34] P. Kornerup and J.-M. Muller. RN-codings of numbers: definition and some properties. Technical Report 2004-44, LIP, ENS-Lyon, 2004.

[35] P. Kornerup, J.-M. Muller, and A. Panhaleux. Performing arithmetic operations on round-to-nearest representations. *IEEE Transactions on Computers*, 60:282–291, 2011.

[36] C. Q. Lauter. *Arrondi Correct de Fonctions Mathématiques.* PhD thesis, École Normale Supérieure de Lyon, Lyon, France, October 2008. In French, available at `http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2008/PhD2008-07.pdf`.

[37] C. Q. Lauter and V. Lefèvre. Exact and mid-point rounding cases of power(x,y). Research Report 2006-46, LIP Laboratory, 2006. Available at `http://prunel.ccsd.cnrs.fr/ensl-00117433/fr/`.

[38] C. Q. Lauter and V. Lefèvre. An efficient rounding boundary test for pow(x, y) in double precision. *IEEE Transactions on Computers*, 58(2):197–207, February 2009.

[39] C. Lee. Multistep gradual rounding. *IEEE Transactions on Computers*, 38:595–600, 1989.

[40] V. Lefèvre. *Moyens Arithmétiques Pour un Calcul Fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[41] N. Louvet, J.-M. Muller, and A. Panhaleux. Newton-Raphson algorithms for floating-point division using an FMA. In F. Charot, F. Hannig, J. Teich, and C. Wolinski, editors, *21st IEEE International Conference on Application-specific Systems Architectures and Processors (ASAP), 2010*, pages 200–207. IEEE, 2010.

[42] P. W. Markstein. Computation of elementary functions on the IBM RISC System/6000 processor. *IBM J. Res. Dev.*, 34(1):111–119, 1990.

[43] P. W. Markstein. *IA-64 and elementary functions: speed and precision*. Hewlett-Packard professional books. 2000.

[44] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(3):12:1–12:41, May 2008.

[45] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser, Boston, MA, 2nd edition, 2006.

[46] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Boston, MA, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[47] M. A. Overton. *Numerical Computing with IEEE Floating-Point Arithmetic*. SIAM, Philadelphia, PA, 2001.

[48] B. Parhami. Carry-free addition of recoded binary signed-digit numbers. *IEEE Transactions on Computers*, 37(11):1470–1476, 1988.

[49] B. Parhami. Generalized signed-digit number systems: A unifying framework for redundant number representations. *IEEE Transactions on Computers*, 39(1):89–98, January 1990.

[50] M. Parks. Number-theoretic test generation for directed rounding. *IEEE Transactions on Computers*, 49(7):651–658, 2000.

[51] M. Parks. Unifying tests for square root. In G. Hanrot and P. Zimmermann, editors, *Proceedings of the 7th Conference on Real Numbers and Computers (RNC 7) LORIA, Nancy, France, July 10–12, 2006*, pages 125–133, 2006.

[52] J. E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7:218–222, 1958. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[53] S. M. Rump, T. Ogita, and S. Oishi. Accurate floating-point summation part I: Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224, 2008.

[54] D. Das Sarma and D. W. Matula. Faithful bipartite ROM reciprocal tables. In S. Knowles and W. H. McAllister, editors, *Proceedings of the 12th IEEE Symposium on Computer Arithmetic (ARITH-12)*, pages 17–28. IEEE Computer Society Press, Los Alamitos, CA, June 1995.

[55] K. D. Tocher. Techniques of multiplication and division for automatic binary divider. *Quarterly journal of mechanics and applied mathematics*, 11(3):364–384, 1958.

[56] S. Wagon. The Euclidean algorithm strikes again. *The American Mathematical Monthly*, 97(2):125–129, February 1990.

[57] M. Waldschmidt. Transcendence of periods: The state of the art. *Pure and Applied Mathematics Quarterly*, 2(2):435–463, 2006. Available at `http://www.intlpress.com/JPAMQ/p/2006/435-463.pdf`.

[58] Wikipedia. Multiply–accumulate operation — Wikipedia, the free encyclopedia, 2012. `http://en.wikipedia.org/w/index.php?title=Multiply-accumulate_operation&oldid=480740046`.

Abstract

Efficient and reliable computer arithmetic is a key requirement to perform fast and reliable numerical computations. The choice of the number system and the choice of the arithmetic algorithms are important. We present a new representation of numbers, the "RN-codings", such that truncating a RN-coded number to some position is equivalent to rounding it to the nearest. We give some arithmetic algorithms for manipulating RN-codings and introduce the concept of "floating-point RN-codings".

When implementing a function f in floating-point arithmetic, if we wish to always return the floating-point number nearest $f(x)$, one must be able to determine if $f(x)$ is above or below the closest "midpoint", where a midpoint is the middle of two consecutive floating-point numbers. This determination is first done with some given precision, and if it does not suffice, we start again with higher precision, and so on. This process may not terminate if $f(x)$ can be a midpoint. Given an algebraic function f, we try either to show that there are no floating-point numbers x such that $f(x)$ is a midpoint, or we try to enumerate or characterize them.

Since the IBM PowerPC, binary division has frequently been implemented using variants of the Newton-Raphson iteration due to Peter Markstein. This iteration is very fast, but much care is needed if we aim at always returning the floating-point number nearest the exact quotient. We investigate a way of efficiently merging Markstein iterations with faster yet less accurate iterations called Goldschmidt iterations. We also investigate whether those iterations can be used for decimal floating-point arithmetic. We provide sure and tight error bounds for these algorithms.

Résumé

Une arithmétique sûre et efficace est un élément clé pour exécuter des calculs rapides et sûrs. Le choix du système numérique et des algorithmes arithmétiques est important. Nous présentons une nouvelle représentation des nombres, les "RN-codes", telle que tronquer un RN-code à une précision donnée est équivalent à l'arrondir au plus près. Nous donnons des algorithmes arithmétiques pour manipuler ces RN-codes et introduisons le concept de "RN-code en virgule flottante."

Lors de l'implantation d'une fonction f en arithmétique flottante, si l'on veut toujours donner le nombre flottant le plus proche de $f(x)$, il faut déterminer si $f(x)$ est au-dessus ou en-dessous du plus proche "midpoint", un "midpoint" étant le milieu de deux nombres flottants consécutifs. Pour ce faire, le calcul est d'abord fait avec une certaine précision, et si cela ne suffit pas, le calcul est recommencé avec une précision de plus en plus grande. Ce processus ne s'arrête pas si $f(x)$ est un midpoint. Étant donné une fonction algébrique f, soit nous montrons qu'il n'y a pas de nombres flottants x tel que $f(x)$ est un midpoint, soit nous les caractérisons ou les énumérons.

Depuis le PowerPC d'IBM, la division en binaire a été fréquemment implantée à l'aide de variantes de l'itération de Newton-Raphson dues à Peter Markstein. Cette itération est très rapide, mais il faut y apporter beaucoup de soin si l'on veut obtenir le nombre flottant le plus proche du quotient exact. Nous étudions comment fusionner efficacement les itérations de Markstein avec les itérations de Goldschmidt, plus rapides mais moins précises. Nous examinons également si ces itérations peuvent être utilisées pour l'arithmétique flottante décimale. Nous fournissons des bornes d'erreurs sûres et précises pour ces algorithmes.