



Optimized Binary64 and Binary128 Arithmetic with GNU MPFR

Vincent Lefèvre, Paul Zimmermann

► **To cite this version:**

Vincent Lefèvre, Paul Zimmermann. Optimized Binary64 and Binary128 Arithmetic with GNU MPFR. 24th IEEE Symposium on Computer Arithmetic (ARITH 24), Jul 2017, London, United Kingdom.

HAL Id: hal-01502326

<https://hal.inria.fr/hal-01502326>

Submitted on 5 Apr 2017

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

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

Optimized Binary64 and Binary128 Arithmetic with GNU MPFR

Vincent Lefèvre and Paul Zimmermann

Abstract—We describe algorithms used to optimize the GNU MPFR library when the operands fit into one or two words. On modern processors, a correctly rounded addition of two quadruple precision numbers is now performed in 22 cycles, a subtraction in 24 cycles, a multiplication in 32 cycles, a division in 64 cycles, and a square root in 69 cycles. We also introduce a new *faithful* rounding mode, which enables even faster computations. Those optimizations will be available in version 4 of MPFR.

Keywords: floating point, correct rounding, faithful rounding, binary64, binary128, GNU MPFR.

I. INTRODUCTION

The IEEE 754-2008 standard [6] defines — among others — the binary floating-point formats binary64 and binary128. They provide respectively a precision of 53 and 113 bits. Those standard formats are used in many applications, therefore it is very important to provide fast arithmetic for them, either in hardware or in software.

GNU MPFR [3] (MPFR for short) is a reference software implementation of the IEEE 754-2008 standard in arbitrary precision. MPFR guarantees *correct rounding* for all its operations, including elementary and special functions [6, Table 9.1]. It provides mixed-precision operations, i.e., the precision of the input operand(s) and the result may differ.

Since 2000, several authors have cited MPFR, either to use it for a particular application, or to compare their own library to MPFR [2], [4], [7], [8]. Most of those comparisons are in the case when all operands have the same precision, which is usually one of the standard binary64 or binary128 formats.

Since its 4.3.0 release, GCC uses MPFR in its middle-end to generate correctly rounded compile-time results regardless of the math library implementation or floating-point precision of the host platform. Several interval arithmetic libraries depend on MPFR: MPFI [12], GNU Octave interval package¹, libieee1788², Moore [10]. The Sage computer algebra system³ uses MPFR with a default precision of 53 bits for its numerical computations.

The contributions of this article are: (i) a description of new algorithms for basic floating-point arithmetic in one- or two-word precision, (ii) an implementation in MPFR of those algorithms with corresponding timings for basic arithmetic and mathematical functions, (iii) a description of a new *faithful* rounding mode, with corresponding implementation and timings in MPFR.

Notations: We use p to denote the precision in bits. A *limb*, following GMP terminology [5], is an unsigned integer that fits in a machine word, and is assumed to have 64 bits here. Some algorithms are also valid when the radix is not 2^{64} , we

then denote β the radix, assumed to be a power of two. We call *left shift* (resp. *right shift*) a shift towards the most (resp. least) significant bits. We use a few shorthands: HIGHBIT denotes the limb 2^{63} ; ONE denotes the limb 1; GMP_NUMB_BITS, which is the number of bits in a limb, is replaced by 64. In the source code, a_0 and e_{\min} are written `a0` and `emin` respectively.

II. BASIC ARITHMETIC

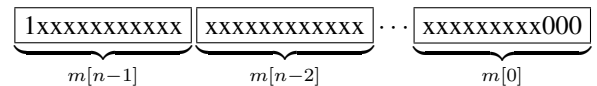
A. The MPFR Internal Format

Internally, a MPFR number is represented by a precision p , a sign s , an exponent e , denoted by `EXP(\cdot)`, and a significand m . The significand is a multiple-precision natural integer represented by an array of $n = \lceil p/64 \rceil$ limbs: we denote by $m[0]$ the least significant limb, and by $m[n-1]$ the most significant one. For a *regular* number — neither NaN, nor $\pm\infty$, nor ± 0 — the most significant bit of $m[n-1]$ must be set, i.e., the number is always *normalized*⁴. The corresponding number is:

$$(-1)^s \cdot (m \cdot 2^{-64n}) \cdot 2^e,$$

i.e., the rational significand $m/2^{64n}$ lies in $[1/2, 1[$.

When the precision p is not an exact multiple of 64, the least $\text{sh} = 64n - p$ bits of $m[0]$ are not used. By convention they must always be zero, like the 3 bits below (drawing the most significant limbs and bits on the left):



In the description of the algorithms below, those sh trailing bits will be represented by 3 zeros.

In this section, we describe basic algorithms used to perform arithmetic on 1 or 2 limbs (addition and subtraction, multiplication, division and square root) for regular inputs, after non-regular inputs have been dealt with. Those algorithms assume the number sh of unused bits is non-zero. As a consequence, on a 64-bit processor, if efficiency is required, it is recommended to use precision $p = 63$ or $p = 127$ instead of $p = 64$ or $p = 128$, whenever possible. In the following, we assume that all input and output arguments have the same precision p . Note that MPFR functions have to take into account that input and output variables can be the same, for example `mpfr_sub(a, b, a, rnd_mode)` will round $b - a$ and store the result in a .

In addition to correct rounding, all MPFR functions return a *ternary value*, giving the sign of the rounding error: 0 indicates that the computed correctly rounded result y exactly equals the infinite precision value $f(x)$ (no rounding error did occur);

¹<https://octave.sourceforge.io/interval/>

²<https://github.com/nehmeier/libieee1788>

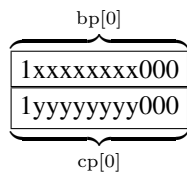
³<http://www.sagemath.org>

⁴MPFR does not have *subnormals*, but provides a function to emulate them for IEEE 754 support.

a positive value indicates that $y > f(x)$, and a negative value indicates that $y < f(x)$. Determining that ternary value is sometimes more expensive than determining the correct rounding: for example if two high-precision numbers x and $1 - x$ are added with a low target precision, the rounding error will usually be less than $\frac{1}{2}$ ulp, thus we can easily decide that the correctly rounded result to nearest is 1, but more work is needed to determine the ternary value.

B. Addition

We describe here the internal `mpfr_add1sp1` function for $0 < p < 64$. Let b and c be the input operands, and a the result. By “addition”, we mean that b and c have same sign. Their significands consist of one limb only: `bp[0]` and `cp[0]`, and their exponents are `bx` and `cx`. Since b and c are regular numbers, we have $2^{63} \leq \text{bp}[0], \text{cp}[0] < 2^{64}$. If $\text{bx} = \text{cx}$, the addition of the significands always produces a carry: $2^{64} \leq \text{bp}[0] + \text{cp}[0] < 2^{65}$:



We thus simply add the two shifted significands — since $p < 64$, we lose no bits in doing this — and increase `bx` by 1 (this will be the result exponent):

```
a0 = (bp[0] >> 1) + (cp[0] >> 1);
bx ++;
```

Since $\text{sh} = 64 - p$ is the number of unused bits of a_0 , the round bit is bit $\text{sh} - 1$ from a_0 , which we set to 0 before storing in memory, and the sticky bit is always 0 in this case:

```
rb = a0 & (ONE << (sh - 1));
ap[0] = a0 ^ rb;
sb = 0;
```

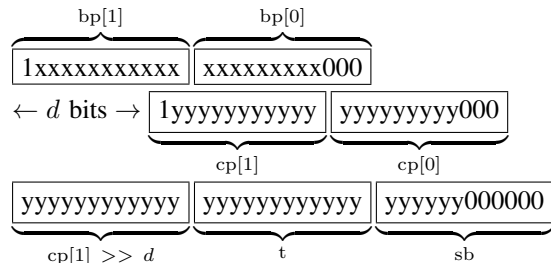
The case $\text{bx} > \text{cx}$ is treated similarly: we have to shift `cp[0]` by $d = \text{bx} - \text{cx}$ bits to the right. We distinguish three cases: $d < \text{sh}$ where the shifted `cp[0]` fully overlaps with `ap[0]`, $\text{sh} \leq d < 64$, where the shifted `cp[0]` partly overlaps with `ap[0]`, and $d \geq 64$, where the shifted `cp[0]` does not overlap with `ap[0]`. In the last case $\text{bx} < \text{cx}$, we simply swap the inputs and reduce to the $\text{bx} > \text{cx}$ case.

Now consider the rounding. All rounding modes are first converted to nearest, toward zero or away from zero. At this point, `ap[0]` contains the current significand, `bx` the exponent, `rb` the round bit and `sb` the sticky bit. No underflow is possible in the addition, since the inputs have same sign: $|b + c| \geq \min(|b|, |c|)$. An overflow occurs when $\text{bx} > \text{emax}$. Otherwise we set the exponent of a to `bx`. If $\text{rb} = \text{sb} = 0$, we return 0 as ternary value, which means the result is exact, whatever the rounding mode. If rounding to nearest, we let `ap[0]` unchanged when either $\text{rb} = 0$, or $\text{rb} \neq 0$ and $\text{sb} = 0$ and bit sh of `ap[0]` is 0 (even rule); otherwise we add one ulp, i.e., 2^{sh} to `ap[0]`. If rounding toward zero, `ap[0]` is unchanged, and we return the opposite of the sign of a as ternary value, which means that the computed value is less than $b + c$ when $b, c > 0$ (remember the exact case $\text{rb} = \text{sb} = 0$ was treated before). If rounding

away from zero, we add one ulp to `ap[0]`; while doing this, we might have a carry, which might in turn give an overflow.

C. Subtraction

We detail here the `mpfr_sub1sp2` function, in the case where the exponent difference $d = \text{bx} - \text{cx}$ satisfies $0 < d < 64$. In that case, the significand of c overlaps with the upper word of the significand of b . We align c on b , giving 3 limbs `cp[1] >> d`, `t` and `sb`:



The goal is to subtract these 3 limbs from b , word by word. While this 3-word subtraction could be done with only 3 instructions on a typical processor, we need to write a portable C code, which is more complex to express the borrow propagation in C (and unfortunately, compilers are not yet able to detect patterns to generate only these 3 instructions):

```
t = (cp[1] << (64 - d)) | (cp[0] >> d);
a0 = bp[0] - t;
a1 = bp[1] - (cp[1] >> d) - (bp[0] < t);
sb = cp[0] << (64 - d);
if (sb) {
    a1 -= (a0 == 0);
    a0 --;
    sb = -sb; /* 2^64 - sb */
}
```

At this point the exact subtraction corresponds to $a_1 + 2^{-64}a_0 + 2^{-128}sb$, where a_1 and a_0 cannot be both zero: if $d \geq 2$ then $a_1 \geq 2^{62}$; if $d = 1$, since bit 0 of `cp[0]` is 0 because $p < 128$, necessarily $\text{sb} = 0$. However we can have $a_1 = 0$, in which case we shift a by one word to the left, and decrease the exponent by 64 (this can only occur for $d = 1$, in which case $\text{sb} = 0$):

```
if (a1 == 0) {
    a1 = a0;
    a0 = 0;
    bx -= 64; }
```

Now $a_1 \neq 0$, and we shift a to the left by the number of leading zeros of a_1 :

```
count_leading_zeros (cnt, a1);
if (cnt) {
    ap[1] = (a1 << cnt) | (a0 >> (64-cnt));
    a0 = (a0 << cnt) | (sb >> (64-cnt));
    sb <<= cnt;
    bx -= cnt; }
else
    ap[1] = a1;
```

We now compute the round and sticky bit, and set to zero the last sh bits of a_0 before storing it in memory, where $\text{mask} = 2^{\text{sh}} - 1$:

```
rb = a0 & (ONE << (sh - 1));
sb |= (a0 & mask) ^ rb;
ap[0] = a0 & ~mask;
```

The rounding is done exactly like for the addition, except in the subtraction we cannot have any overflow since here

all arguments have the same precision, but we can have an underflow.

D. Multiplication

The multiplication is easy to implement with the internal format chosen for MPFR. We detail here the `mpfr_mul_1` function, for $0 < p < 64$. We first add the two exponents and multiply the two input significands `bp[0]` and `cp[0]` using GMP's `umul_ppmm` macro, which multiplies two 64-bit words, and stores the upper and lower words in `a0` and `sb` here:

```
ax = EXP(b) + EXP(c);
umul_ppmm(a0, sb, bp[0], cp[0]);
```

Since $2^{63} \leq bp[0], cp[0] < 2^{64}$, we have $2^{126} \leq bp[0]cp[0] = a_0 \cdot 2^{64} + sb < 2^{128}$. The upper word a_0 therefore satisfies $2^{62} \leq a_0 < 2^{64}$, and in case $2^{62} \leq a_0 < 2^{63}$, we have to shift a_0 and `sb` by one bit to the left and decrease the output exponent:

```
if (a0 < HIGHBIT) {
    ax --;
    a0 = (a0 << 1) | (sb >> 63);
    sb = sb << 1; }
```

The round and sticky bits are computed exactly like in the 2-limb subtraction, and the sign of the result is set:

```
rb = a0 & (ONE << (sh - 1));
sb |= (a0 & mask) ^ rb;
ap[0] = a0 & ~mask;
SIGN(a) = MULT_SIGN(SIGN(b), SIGN(c));
```

For the multiplication, both overflow and underflow can happen. Overflow is easy to handle. For underflow, MPFR signals underflow *after* rounding: this means that underflow occurs when the rounded result — with an unbounded exponent range — has an exponent smaller than e_{\min} . Thus for example, when `ax = $e_{\min} - 1$` , `rb` or `sb` is non-zero, `ap[0] = 111...111000` and we round away from zero, there is no underflow. Apart from this special case, the rounding is the same as for the subtraction. (This special case where `ax = $e_{\min} - 1$` and nevertheless no underflow occurs cannot happen in the subtraction, since b and c are multiples of $2^{e_{\min}-p}$, likewise for $b - c$; thus if the exponent `ax` of the difference is smaller than e_{\min} , the difference $b - c$ is exact, and `rb = sb = 0`.)

In the `mpfr_mul_2` function, which multiplies $b_1 \cdot 2^{64} + b_0$ by $c_1 \cdot 2^{64} + c_0$, the product b_0c_0 of the low limbs contributes to less than 1 to the second most significant limb of the full 4-limb product (before a potential left shift by 1 bit to get a normalized result). Thus we only perform 3 calls to `umul_ppmm`, to compute b_1c_1 , b_1c_0 and b_0c_1 . Moreover, we ignore the lower limbs of b_1c_0 and b_0c_1 , which yields a 127- or 128-bit approximate product with error less than 3.

E. Division

The division code for 1 or 2 limbs first computes an approximation of the quotient, and if this approximation is not sufficient to determine the correct rounding, an exact quotient and remainder are computed, starting from that approximation. The algorithms use GMP's `invert_limb` function (currently re-implemented in MPFR since it is not yet in GMP's public interface). Given a limb v such that $\beta/2 \leq v < \beta$,

`invert_limb` returns the limb $\lfloor (\beta^2 - 1)/v \rfloor - \beta$, which is called the *reciprocal* of v .

All division functions first check whether the dividend significand u is larger or equal to the divisor significand v , where $\beta^n/2 \leq u, v < \beta^n$. If that occurs, v is subtracted from u , which generates an extra leading quotient bit. After that subtraction, we get $u - v < v$ since $u < \beta^n \leq 2v$. Therefore in the following we assume $u < v$.

We detail in Algorithm DivApprox1 how we compute an approximate quotient q for the `mpfr_div_1` function, which divides two one-limb significands. For $\beta = 2^{64}$, the approx-

Algorithm 1 DivApprox1

Input: integers u, v with $0 \leq u < v$ and $\beta/2 \leq v < \beta$

Output: integer q approximating $u\beta/v$

1: compute an approximate reciprocal i of v , satisfying

$$i \leq \lfloor (\beta^2 - 1)/v \rfloor - \beta \leq i + 1 \quad (1)$$

2: $q = \lfloor iu/\beta \rfloor + u$

imate reciprocal i in step 1 is obtained using the variable v_3 from Algorithm 2 (`RECIPROCAL_WORD`) in [11]; indeed Theorem 1 from [11] proves that — with our notation — $0 < \beta^2 - (\beta + v_3)v < 2v$, from which the inequalities (1) follow.

Theorem 1: The approximate quotient returned by Algorithm DivApprox1 satisfies

$$q \leq \lfloor \frac{u\beta}{v} \rfloor \leq q + 2.$$

Proof. Step 2 of DivApprox1 is simply step 1 of Algorithm 1 (`DIV2BY1`) from [11]. If i is the exact reciprocal $i_0 := \lfloor (\beta^2 - 1)/v \rfloor - \beta$, and $q_0 := \lfloor i_0 u/\beta \rfloor + u$ is the corresponding quotient, it is proven in [11] that the corresponding remainder $r_0 = \beta u - q_0 v$ satisfies $0 \leq r_0 < 4v$. However the upper bound $4v$ includes $2v$ coming from a lower dividend term which is zero here, thus we have $r_0 < 2v$.

In the case $i = i_0 - 1$, then $q \leq q_0 \leq q + 1$, thus $r = \beta u - qv$ satisfies $r < 3v$. \square

As a consequence of Theorem 1, we can decide the correct rounding except when the last `sh - 1` bits from q are 000...000, 111...111, or 111...110, which occurs with probability less than 0.15% for the binary64 format. If the rare cases where we cannot decide the correct rounding, we compute $r = \beta u - qv$, subtract v and increment q at most two times until $r < v$, and deduce the round and sticky bits from the updated quotient q and remainder r .

The upper bound $q + 2$ is tight: it is attained for $(u, v) = (10, 12)$ in radix $\beta = 18$, and for $(u, v) = (21, 24)$ in power-of-two radix $\beta = 32$, taking $i = i_0 - 1$ in step 1.

For the 2-limb division, we use a similar algorithm (DivApprox2), which approximates the integer quotient of the significands. Since this algorithm is not specific to radix 2^{64} , we describe it for a general radix β .

Algorithm DivApprox2 first computes a lower approximation q_1 of the upper quotient word, from the upper word u_1 of the dividend and the approximate reciprocal of the upper word

Algorithm 2 DivApprox2**Input:** integers u, v with $0 \leq u < v$ and $\beta^2/2 \leq v < \beta^2$ **Output:** approximate quotient q of $u\beta^2/v$

- 1: write $u = u_1\beta + u_0$ and $v = v_1\beta + v_0$ with $0 \leq u_i, v_i < \beta$
- 2: compute x satisfying

$$x \leq \lfloor \beta^2 / (v_1 + 1) \rfloor \leq x + 1$$

- 3: $q_1 = \lfloor u_1x/\beta \rfloor$
- 4: $r = u - \lceil q_1v/\beta \rceil$ (equals $\lfloor (u\beta - q_1v)/\beta \rfloor$)
- 5: $q_0 = \lfloor rx/\beta \rfloor$
- 6: return $q = q_1\beta + q_0$

v_1 of the divisor. The end of Algorithm DivApprox2 can be seen as an integer version of Karp-Markstein's algorithm for the division [9], or more simply, once the remainder has been approximated by r , we use the reciprocal again to approach the lower quotient word.

In step 2, x can be computed as follows: if $v_1 + 1 = \beta$, take $x = \beta$, otherwise (assuming β is a power of two) $v_1 + 1$ cannot divide β^2 exactly, thus $\lfloor \beta^2 / (v_1 + 1) \rfloor = \lfloor (\beta^2 - 1) / (v_1 + 1) \rfloor$, and we can take the same approximate reciprocal as in Algorithm DivApprox1 (with $v_1 + 1$ instead of v).⁵ Write $r = r_1\beta + r_0$ with $0 \leq r_0 < \beta$. As we will see later, the upper word satisfies $r_1 \leq 4$. Step 6 becomes $q = q_1\beta + r_1x + \lfloor r_0x/\beta \rfloor$, where the computation of r_1x is done using a few additions instead of a multiplication.

Theorem 2: For $\beta \geq 5$, the approximate quotient returned by Algorithm DivApprox2 satisfies

$$q \leq \lfloor \frac{u\beta^2}{v} \rfloor \leq q + 21.$$

Proof. The input condition $u < v$ implies $u_1 \leq v_1$, thus since $x \leq \beta^2 / (v_1 + 1)$, it follows $q_1 \leq u_1\beta / (v_1 + 1) < \beta$, thus q_1 fits in one limb. We have $\beta^2 = (v_1 + 1)x + \kappa$ with $0 \leq \kappa \leq 2v_1 + 1$. Step 3 gives $u_1x = q_1\beta + t$, with $0 \leq t < \beta$. Then

$$\begin{aligned} u\beta - q_1v &= u\beta - q_1(v_1\beta + v_0) \\ &= u\beta - q_1(v_1 + 1)\beta + q_1(\beta - v_0) \\ &= u\beta - (v_1 + 1)(u_1x - t) + q_1(\beta - v_0) \\ &= u\beta - (\beta^2 - \kappa)u_1 + (v_1 + 1)t + q_1(\beta - v_0) \\ &= u_0\beta + \kappa u_1 + (v_1 + 1)t + q_1(\beta - v_0). \end{aligned}$$

Write $v_1 = \alpha\beta$ with $1/2 \leq \alpha < 1$. Thus

$$\begin{aligned} u\beta - q_1v &\leq (\beta - 1)\beta + (2\alpha\beta + 1)(\alpha\beta) \\ &\quad + (\alpha\beta + 1)(\beta - 1) + (\beta - 1)\beta \\ &< \beta^2(2 + \alpha + 2\alpha^2). \end{aligned}$$

This proves that $r < 5\beta$ at step 4.

From $r < \beta(2 + \alpha + 2\alpha^2)$ it follows $q_0 \leq rx/\beta < (2 + \alpha + 2\alpha^2)x$, thus since $x \leq \beta^2 / (v_1 + 1) < \beta/\alpha$, $q_0 < (2/\alpha + 1 +$

⁵In practice, like in Algorithm DivApprox1, $x = \beta + x'$ is split between one implicit upper bit β and a lower part x' , that for simplicity we do not distinguish here. In the real code, step 3 therefore becomes $q_1 = u_1 + \lfloor u_1x'/\beta \rfloor$, and step 5 becomes $q_0 = r + \lfloor rx'/\beta \rfloor$.

$2\alpha)\beta$ (note that q_0 might exceed β). Let $rx = q_0\beta + \delta_q$, and $u\beta - q_1v = \beta r + \delta_r$, with $0 \leq \delta_q, \delta_r < \beta$. Then

$$\begin{aligned} u\beta^2 - qv &= \beta(\beta r + \delta_r) - q_0v \\ &= \beta^2 r + \beta\delta_r - q_0(v_1 + 1)\beta + q_0(\beta - v_0) \\ &= \beta^2 r + \beta\delta_r - (rx - \delta_q)(v_1 + 1) + q_0(\beta - v_0) \\ &= \beta^2 r + \beta\delta_r - r(\beta^2 - \kappa) + \delta_q(v_1 + 1) + q_0(\beta - v_0) \\ &= \beta\delta_r + r\kappa + \delta_q(v_1 + 1) + q_0(\beta - v_0). \end{aligned}$$

Bounding $\beta\delta_r$ by $\beta^2 - \beta$, $r\kappa$ by $\beta(2 + \alpha + 2\alpha^2)(2\alpha\beta + 1)$, $\delta_q(v_1 + 1)$ by $(\beta - 1)(\alpha\beta + 1)$, and $q_0(\beta - v_0)$ by $(2/\alpha + 1 + 2\alpha)\beta^2$, it follows:

$$0 \leq u\beta^2 - qv < \beta^2 \left(\frac{2}{\alpha} + 2 + 7\alpha + 2\alpha^2 + 4\alpha^3 \right) + \beta(2 + 2\alpha^2).$$

Dividing by $v \geq \alpha\beta^2$ we get:

$$0 \leq \frac{u\beta^2}{v} - q < \frac{2}{\alpha^2} + \frac{2}{\alpha} + 7 + 2\alpha + 4\alpha^2 + \frac{1}{\beta} \left(\frac{2}{\alpha} + 2\alpha \right). \quad (2)$$

The right-hand side is bounded by $21 + 5/\beta$ for $1/2 \leq \alpha < 1$, thus for $\beta \geq 5$ we have $q \leq u\beta^2/v < q + 22$. \square

The largest error we could find is 20, for example for $\beta = 4096$, $u = 8298491$, $v = 8474666$.⁶ With $\beta = 2^{64}$, on 10^7 random inputs, the proportion of inputs for which $r_1 = 0$ is 29%, $r_1 = 1$ is 59%, $r_1 = 2$ is 12%, $r_1 = 3$ is 0.2%, and $r_1 = 4$ is 0%; and the average difference between q and $\lfloor u\beta^2/v \rfloor$ is 2.8.

F. Square Root

Like for the division, we first compute an approximate result for the square root, and if we are not able to get the correctly rounded result, we compute an exact square root with remainder from that approximation. We first outline the exact algorithm for 1 limb. In case the input exponent is odd, we shift the input significand u_0 by one bit to the right (this is possible without any bit lost because $p < 64$). Then given the (possibly shifted) limb u_0 , one computes an integer square root:

$$u_0 \cdot 2^{64} = r_0^2 + s, \quad 0 \leq s \leq 2r_0.$$

Since $2^{62} \leq u_0 < 2^{64}$, we have $2^{126} \leq u_0 \cdot 2^{64} < 2^{128}$, thus $2^{63} \leq r_0 < 2^{64}$: r_0 has exactly 64 significant bits. The round and sticky bits are then computed from the $64 - p$ bits of r_0 and from the remainder s .

The expensive part is the computation of the (approximate or exact) integer square root. Since MPFR has to be independent of the machine floating-point hardware, we should use integer operations only. GMP's `mpn_sqrtrem` function implements the algorithm described in [13], which relies on integer division. We can do better by first computing an approximate reciprocal square root.

1) *Approximate Reciprocal Square Root:* Algorithm 3 uses an integer variant of Newton's iteration for the reciprocal square root: v_0 is a 11-bit value such that $x_0 := v_0/2^{10}$ approximates the root of $a_0 := d_{10}/2^{10}$, v_1 is a 22-bit

⁶If $\beta^2 + 1$ has no divisors in $[\beta/2 + 1, \beta]$, as for $\beta = 2^{32}$ and $\beta = 2^{64}$, then we cannot have $\kappa = 2v_1 + 1$, the $1/\beta$ term disappears in Eq. (2), and the bound becomes $u\beta^2/v < q + 21$.

Algorithm 3 RecSqrtApprox1**Input:** integer d with $2^{62} \leq d < 2^{64}$ **Output:** integer v_3 approximating $s = \lfloor 2^{96}/\sqrt{d} \rfloor$

- 1: $d_{10} = \lfloor 2^{-54}d \rfloor + 1$
- 2: $v_0 = \lfloor \sqrt{2^{30}/d_{10}} \rfloor$ (table lookup)
- 3: $d_{37} = \lfloor 2^{-27}d \rfloor + 1$
- 4: $e_0 = 2^{57} - v_0^2 d_{37}$
- 5: $v_1 = 2^{11}v_0 + \lfloor 2^{-47}v_0e_0 \rfloor$
- 6: $e_1 = 2^{79} - v_1^2 d_{37}$
- 7: $v_2 = 2^{10}v_1 + \lfloor 2^{-70}v_1e_1 \rfloor$
- 8: $e_2 = 2^{126} - v_2^2 d$
- 9: $v_3 = 2^{33}v_2 + \lfloor 2^{-94}v_2e_2 \rfloor$

value such that $x_1 := v_1/2^{21}$ approximates the root of $a_1 := d_{37}/2^{37}$, v_2 is a 32-bit value such that $x_2 := v_2/2^{31}$ approximates the root of $a_2 := d_{37}/2^{37}$, and v_3 is a 65-bit value such that $x_3 := v_3/2^{64}$ approximates the root of $a_3 := d/2^{64}$.

Theorem 3: The value v_3 returned by Algorithm 3 differs by at most 8 from the reciprocal square root:

$$v_3 \leq s := \lfloor 2^{96}/\sqrt{d} \rfloor \leq v_3 + 8.$$

Lemma 1: Assume positive real numbers x_0, x_1, \dots, x_n are computed using the following recurrence, with $x_0 \leq a_0^{-1/2}$:

$$x_{k+1} = x_k + \frac{x_k}{2}(1 - a_{k+1}x_k^2), \quad (3)$$

where $a_0 \geq a_1 \geq \dots \geq a_n \geq a > 0$, then:

$$x_1 \leq x_2 \leq \dots \leq x_n \leq a^{-1/2}.$$

Proof. It follows from [1, Lemma 3.14] that $x_{k+1} \leq a_{k+1}^{-1/2}$. Together with $x_0 \leq a_0^{-1/2}$, this gives $1 - a_k x_k^2 \geq 0$ for all k . Since $a_{k+1} \leq a_k$, it follows that $1 - a_{k+1} x_k^2 \geq 1 - a_k x_k^2 \geq 0$. Put into (3) this proves $x_{k+1} \geq x_k$, and thus the lemma since $x_{k+1} \leq a_{k+1}^{-1/2}$ and $a_{k+1} \geq a$ imply $x_{k+1} \leq a^{-1/2}$. \square

This lemma remains true when the correction term $\frac{x_k}{2}(1 - a_{k+1}x_k^2)$ in Eq. (3) — which is thus non-negative — is rounded down towards zero. By applying this result to $x_0 = v_0/2^{10}$, $x_1 = v_1/2^{21}$, $x_2 = v_2/2^{31}$, $x_3 = v_3/2^{64}$, with $a_0 = d_{10}/2^{10} \geq a_1 = d_{37}/2^{37} = a_2 \geq a_3 = d/2^{64}$, it follows $1 \leq v_0/2^{10} \leq v_1/2^{21} \leq v_2/2^{31} \leq v_3/2^{64} \leq 2^{32}/\sqrt{d} \leq 2$, thus $2^{10} \leq v_0 < 2^{11}$, $2^{21} \leq v_1 < 2^{22}$, $2^{31} \leq v_2 < 2^{32}$, $2^{64} \leq v_3 < 2^{65}$ (the right bounds are strict because in the only case where $2^{32}/\sqrt{d} = 2$, i.e., $d = 2^{62}$, we have $v_3 = 2^{65} - 3$).

Proof of Theorem 3. The construction of the lookup table ensures that $v_0 < 2^{11}$ and $e_0 \geq 0$ at step 4 of the algorithm. By exhaustive search on all possible d_{10} values, we get:

$$0 \leq e_0 < 2^{49.263}.$$

Thus at step 4, we can compute $2^{57} - v_0^2 d_{37}$ in 64-bit arithmetic, which will result in a number of at most 50 bits, and step 5 can be done in 64-bit arithmetic, since the product $v_0 e_0$ will have at most 61 bits.

Let δ_1 be the truncation error in step 5, with $0 \leq \delta_1 < 1$:

$$2^{-47}v_0e_0 = \lfloor 2^{-47}v_0e_0 \rfloor + \delta_1.$$

Then:

$$\begin{aligned} e_1 &= 2^{79} - v_1^2 d_{37} = 2^{79} - d_{37}((v_1 + \delta_1) - \delta_1)^2 \\ &= 2^{79} - d_{37}(v_1 + \delta_1)^2 + d_{37}\delta_1(2v_1 + \delta_1). \end{aligned}$$

Since $v_1^2 d_{37} \leq 2^{79}$ — because $e_1 \geq 0$ —, then $v_1 d_{37} \leq 2^{79}/v_1 \leq 2^{58}$ because $2^{21} \leq v_1$. It follows that $\gamma_1 := d_{37}\delta_1(2v_1 + \delta_1)$ satisfies $0 \leq \gamma_1 < 2^{59} + 2^{37}$, and:

$$\begin{aligned} e_1 - \gamma_1 &= 2^{79} - d_{37}(v_1 + \delta_1)^2 \\ &= 2^{79} - d_{37}(2^{11}v_0 + 2^{-47}v_0e_0)^2 \\ &= 2^{79} - d_{37}v_0^2(2^{11} + 2^{-47}e_0)^2. \end{aligned}$$

Now, using $v_0^2 d_{37} = 2^{57} - e_0$:

$$\begin{aligned} e_1 - \gamma_1 &= 2^{79} - (2^{57} - e_0)(2^{11} + 2^{-47}e_0)^2 \\ &= 2^{79} - (2^{57} - e_0)(2^{22} + 2^{-35}e_0 + 2^{-94}e_0^2) \\ &= 2^{79} - (2^{79} - 2^{-35}e_0^2 + 2^{-37}e_0^2 - 2^{-94}e_0^3) \\ &= 2^{-35}e_0^2(3/4 + 2^{-59}e_0). \end{aligned}$$

Since $e_0 < 2^{49.263}$ we deduce $3/4 + 2^{-59}e_0 < 2^{-0.412}$ and

$$0 \leq e_1 < 2^{-35}2^{98.526}2^{-0.412} + 2^{59} + 2^{37} < 2^{63.196}.$$

Therefore $e_1 < 2^{64}$ and e_1 can be computed using integer arithmetic modulo 2^{64} . Since $d_{10} = \lfloor 2^{-27}(d_{37} - 1) \rfloor + 1$, e_1 only depends on d_{37} , so that we can perform an exhaustive search on the $2^{37} - 2^{35}$ possible values of d_{37} . By doing this, we find that the largest value of e_1 is obtained for $d_{37} = 33 \cdot 2^{30} + 1$, which corresponds to $d_{10} = 265$; this gives $e_1 = 10263103231123743388 < 2^{63.155}$.

Let δ_2 be the truncation error in step 7, $0 \leq \delta_2 < 1$:

$$2^{-70}v_1e_1 = \lfloor 2^{-70}v_1e_1 \rfloor + \delta_2.$$

Then:

$$\begin{aligned} e_2 &= 2^{126} - v_2^2 d = 2^{126} - d((v_2 + \delta_2) - \delta_2)^2 \\ &= 2^{126} - d(v_2 + \delta_2)^2 + d\delta_2(2v_2 + \delta_2). \end{aligned}$$

Since $v_2 \geq 2^{31}$ and $v_2^2 d \leq 2^{126}$, $v_2 d \leq 2^{95}$, thus $d\delta_2(2v_2 + \delta_2) < 2^{96} + 2^{64}$.

$$\begin{aligned} e_2 - (2^{96} + 2^{64}) &\leq 2^{126} - d(v_2 + \delta_2)^2 \\ &= 2^{126} - d(2^{10}v_1 + 2^{-70}v_1e_1)^2 \\ &= 2^{126} - dv_1^2(2^{10} + 2^{-70}e_1)^2. \end{aligned}$$

Now writing $d = 2^{27}d_{37} - \rho$ with $0 < \rho \leq 2^{27}$, using $d_{37}v_1^2 = 2^{79} - e_1$, and writing $\varepsilon = 2^{96} + 2^{64} + \rho v_1^2(2^{10} + 2^{-70}e_1)^2$:

$$\begin{aligned} e_2 - \varepsilon &\leq 2^{126} - 2^{27}(2^{79} - e_1)(2^{10} + 2^{-70}e_1)^2 \\ &= 2^{126} - (2^{79} - e_1)(2^{47} + 2^{-32}e_1 + 2^{-113}e_1^2) \\ &= 2^{126} - (2^{126} - 2^{-32}e_1^2 + 2^{-34}e_1^2 - 2^{-113}e_1^3) \\ &= 2^{-32}e_1^2(3/4 + 2^{-81}e_1). \end{aligned}$$

Since $e_1 < 2^{63.155}$ we deduce $3/4 + 2^{-81}e_1 < 2^{-0.415}$, and:

$$\rho v_1^2(2^{10} + 2^{-70}e_1)^2 \leq 2^{71}(2^{20} + 2^5 + 2^{-12}).$$

Thus $\varepsilon \leq 2^{96} + 2^{64} + 2^{91} + 2^{76} + 2^{59}$ and:

$$0 \leq e_2 < 2^{-32}2^{126.31}2^{-0.415} + \varepsilon < 2^{96.338}. \quad (4)$$

Here again, an exhaustive search is possible, since for a given value of d_{37} , e_2 is maximal when $d = 2^{27}(d_{37} - 1)$: the largest value of e_2 is obtained for $d_{37} = 132607222902$, corresponding to $d_{10} = 989$, with $e_2 = 81611919949651931475229016064 < 2^{96.043}$.

The final error is estimated using the truncation error δ_3 in step 9:

$$2^{-94}v_2e_2 = [2^{-94}v_2e_2] + \delta_3$$

$$\begin{aligned} e_3 &= 2^{192} - v_3^2d = 2^{192} - d((v_3 + \delta_3) - \delta_3)^2 \\ &= 2^{192} - d(v_3 + \delta_3)^2 + d\delta_3(2v_3 + \delta_3). \end{aligned}$$

Since $v_3^2d \leq 2^{192}$, and $v_3 \geq 2^{64}$, it follows $v_3d \leq 2^{128}$, thus $\gamma_3 := d\delta_3(2v_3 + \delta_3) < 2^{129} + 2^{64}$.

$$\begin{aligned} e_3 - \gamma_3 &= 2^{192} - d(v_3 + \delta_3)^2 \\ &= 2^{192} - d(2^{33}v_2 + 2^{-94}v_2e_2)^2 \\ &= 2^{192} - dv_2^2(2^{33} + 2^{-94}e_2)^2. \end{aligned}$$

Now since $dv_2^2 = 2^{126} - e_2$:

$$\begin{aligned} e_3 - \gamma_3 &\leq 2^{192} - (2^{126} - e_2)(2^{33} + 2^{-94}e_2)^2 \\ &= 2^{192} - (2^{126} - e_2)(2^{66} + 2^{-60}e_2 + 2^{-188}e_2^2) \\ &= 2^{192} - (2^{192} - 2^{-60}e_2^2 + 2^{-62}e_2^2 - 2^{-188}e_2^3) \\ &= 2^{-60}e_2^2(3/4 + 2^{-128}e_2). \end{aligned}$$

Since $e_2 < 2^{96.043}$ we deduce $3/4 + 2^{-128}e_2 < 2^{-0.415}$:

$$0 \leq e_3 < 2^{-60}2^{192.086}2^{-0.415} + 2^{129} + 2^{64} < 2^{131.882}.$$

Again by exhaustive search, restricted on the values of d_{37} that give a sufficient large value of e_2 , we found the maximal value of e_3 satisfies $e_3 < 2^{131.878}$.

Now, given v_3 returned by Algorithm 3, let $c \geq 0$ be the real number such that $(v_3 + c)^2d = 2^{192}$. Since $2^{192} - v_3^2d < 2^{131.878}$, it follows $2v_3cd < 2^{131.878}$, thus $c < 2^{131.878}/(2v_3d)$. Since $v_3^2d = 2^{192} - e_3$, and $v_3 < 2^{65}$, we have $v_3d \geq (2^{192} - 2^{131.878})/2^{65} > 2^{126.999}$. Therefore $c < 2^{3.879} < 15$.

By doing again this error analysis for a given value of d_{37} , we get a tighter bound involving the δ_i truncation errors. For example we can bound $d_{37}\delta_1(2v_1 + \delta_1)$ by $d_{37}(2v_{1,\max} + 1)$, where $v_{1,\max} = \sqrt{2^{79}/d_{37}}$. This yields a finer bound. By exhaustive search on all possible d_{37} values, we found the maximal error is at most 8. \square

The bound of 8 is optimal, since it is attained for $d = 4755801239923458105$.

Remark 1: one can replace $[2^{-94}v_2e_2]$ in step 9 by $[2^{-29}v_2e'_2]$, where $e'_2 = [2^{-65}e_2]$ has at most 32 bits, like v_2 , and thus the product $v_2e'_2$ can be computed with a low 64-bit product, which is faster than a full product giving both the low and high words. If we write $e_2 = 2^{65}e'_2 + r$ with $0 \leq r < 2^{65}$, then:

$$2^{-94}v_2e_2 - 2^{-29}v_2e'_2 = 2^{-94}v_2r < 2^3.$$

This increases the maximal error from 8 to 15, since for the only value $d_{37} = 35433480442$ that can yield $c \geq 8$ with the original step 9, we have $c < 8.006$ and $v_2 = 4229391409$, thus $c + 2^{-94}v_2r < 15.9$.

2) *The mpfr_sqrt1 function*: The `mpfr_sqrt1` function computes the square root of a 1-limb number n using Algorithm 4 (`SqrtApprox1`). In step 1, it computes a 32-bit approximation of $2^{63}/\sqrt{n}$ using the value v_2 of Algorithm 3 (called with $d = n$), then uses this approximation to deduce the exact integer square root y of n (step 2) and the corresponding remainder z (step 3), and finally uses again x to approximate the correction term t (step 4) to form the approximation s in step 5. Steps 2 and 3 can be implemented as follows. First compute an initial $y = [2^{-32}x[2^{-31}n]]$ and the corresponding remainder $z = n - y^2$. Then as long as $z \geq 2y + 1$, subtract $2y + 1$ to z and increment y by one. Note that since $x^2n < 2^{126}$, $xn/2^{31} < 2^{95}/n \leq 2^{32}$, thus $x[2^{-31}n]$ fits on 64 bits.

Algorithm 4 SqrtApprox1

Input: integer n with $2^{62} \leq n < 2^{64}$

Output: integer s approximating $\sqrt{2^{64}n}$

1: compute an integer x approximating $2^{63}/\sqrt{n}$ with

$$x \leq 2^{63}/\sqrt{n}$$

2: $y = [\sqrt{n}]$ (using the approximation x)

3: $z = n - y^2$

4: $t = [2^{-32}xz]$

5: $s = y \cdot 2^{32} + t$

Theorem 4: If the approximation x in step 1 is the value v_2 of Algorithm 3, then Algorithm `SqrtApprox1` returns s satisfying

$$s \leq [\sqrt{2^{64}n}] \leq s + 7.$$

Proof Write $n = \alpha 2^{64}$ with $1/4 \leq \alpha < 1$, $x = 2^{63}/\sqrt{n} - \delta_x$, $y = \sqrt{n} - \delta_y$ and $t = 2^{-32}xz - \delta_t$ with $0 \leq \delta_y, \delta_t < 1$. Since $x^2n > 2^{126} - 2^{96.338}$ from Eq. (4), $x\sqrt{n} > 2^{63} - 2^{32.339}$, thus $2^{63} - x\sqrt{n} = \delta_x\sqrt{n} < 2^{33.339}$:

$$\begin{aligned} 2^{64}n - s^2 &= 2^{64}n - (y \cdot 2^{32} + t)^2 \\ &= 2^{64}(n - y^2) - 2^{33}yt - t^2 \\ &= 2^{64}z - 2^{33}yt - t^2. \end{aligned} \quad (5)$$

We first bound $2^{64}n - s^2$ by above, assuming it is non-negative:

$$\begin{aligned} 2^{33}yt &= 2xyz - 2^{33}\delta_t y \\ &= (2^{64}/\sqrt{n} - 2\delta_x)(\sqrt{n} - \delta_y)z - 2^{33}\delta_t y \end{aligned} \quad (6)$$

thus since $\delta_x\sqrt{n} < 2^{33.339}$ and $\delta_y, \delta_t < 1$:

$$\begin{aligned} 2^{64}z - 2^{33}yt &\leq (2\delta_x\sqrt{n} + 2^{64}/\sqrt{n})z + 2^{33}y \\ &= 2^{32}z(2^{2.339}\sqrt{\alpha} + 1/\sqrt{\alpha}) + 2^{65}\sqrt{\alpha}, \end{aligned}$$

where we used $y \leq \sqrt{n} = 2^{32}\sqrt{\alpha}$. Also:

$$z = n - (\sqrt{n} - \delta_y)^2 \leq 2\sqrt{n}\delta_y \leq 2^{33}\alpha^{1/2}.$$

Substituting this in the bound for $2^{64}n - s^2$ gives:

$$2^{64}n - s^2 \leq 2^{64}z - 2^{33}yt \leq 2^{65}f(\alpha),$$

with

$$f(\alpha) = 2^{2.339}\alpha + 1 + \alpha^{1/2}.$$

Let $c \geq 0$ such that $2^{64}n = (s+c)^2$. Then $2^{64}n - s^2 = 2sc + c^2$, which implies $2sc < 2^{65}f(\alpha)$, thus

$$c < 2^{64}f(\alpha)/s. \quad (7)$$

Since $s \geq 2^{63}$, and the maximum of $f(\alpha)$ for $1/4 \leq \alpha \leq 1$ is less than 7.06 (attained at $\alpha = 1$), we get $c < 14.12$. Now this gives $s > \sqrt{2^{64}n} - 15 > \sqrt{\alpha}2^{64}/1.01$, therefore $c < 1.01 \cdot f(\alpha)/\sqrt{\alpha}$. Now the function $f(\alpha)/\sqrt{\alpha}$ is bounded by 7.06 for $1/4 \leq \alpha \leq 1$, the maximum still attained at $\alpha = 1$. Therefore $c < 1.01 \cdot 7.06 < 7.14$, which proves the upper bound 7.

Now assume $2^{64}n - s^2 < 0$. Since $2^{64}z - 2^{33}yt \geq 0$ by Eq. (6), we have $2^{64}n - s^2 \geq -t^2$ by Eq. (5). Since $x \leq 2^{63}/\sqrt{n}$ and $z \leq 2^{33}\sqrt{\alpha}$:

$$t \leq 2^{-32}xz < 2^{32}.$$

(The last inequality is strict because we can have $t = 2^{32}$ only when $x = 2^{63}/\sqrt{n}$, which can only happen when $n = 2^{62}$, but in that case $z = 0$.) This proves that the product xz at step 4 can be computed modulo 2^{64} . Now write $2^{64}n = (s - c)^2$ with $c > 0$:

$$\begin{aligned} 2^{64}n - s^2 &\geq -t^2 > -2^{64} \\ (s - c)^2 - s^2 &> -2^{64} \\ 2sc &< 2^{64} + c^2. \end{aligned}$$

This inequality has no solutions for $2^{63} \leq s - c < 2^{64}$. Indeed, since we assumed $2^{64}n - s^2 < 0$, this implies $s > 2^{63}$, because for $s = 2^{63}$ we have $s^2 \leq 2^{64}n$. But then if $s = 2^{63} + u$ with $u \geq 1$ we would need $2^{64}c + 2uc < 2^{64} + c^2$, which we can rewrite as $2u < 2^{64}/c + c - 2^{64}$. For $1 \leq c \leq 2^{64}$, the expression $2^{64}/c + c$ is bounded by $2^{64} + 1$, which yields $2u < 1$, having no integer solutions $u \geq 1$. \square

3) *The `mpfr_sqrt2` function:* The `mpfr_sqrt2` function first computes a 128-bit approximation of the square root using Algorithm 5, where in step 1 we can use Algorithm 3, with the variant of Remark 3, achieving the bound $\delta = 16$. Al-

Algorithm 5 SqrtApprox2

Input: integer $n = n_1 \cdot 2^{64} + n_0$ with $2^{62} \leq n_1 < 2^{64}$ and $0 \leq n_0 < 2^{64}$

Output: integer s approximating $\sqrt{2^{128}n}$

1: compute an integer x approximating $2^{96}/\sqrt{n_1}$ with

$$0 \leq 2^{96}/\sqrt{n_1} - x < \delta := 16$$

2: $y = \lfloor 2^{32}\sqrt{n_1} \rfloor$ (using the approximation x)

3: $z = n - y^2$

4: $t = \lfloor xz/2^{65} \rfloor$

5: $s = y \cdot 2^{64} + t$

gorithm SqrtApprox2 is an integer version of Karp-Markstein's algorithm for the square root [9], which incorporates $n/2^{128}$ in Newton's iteration for the reciprocal square root.

Theorem 5: Algorithm SqrtApprox2 returns s satisfying

$$s - 4 \leq \lfloor \sqrt{2^{128}n} \rfloor \leq s + 26.$$

By construction, we have $x \leq 2^{96}/\sqrt{n_1 + 1} < 2^{128}/\sqrt{n}$, therefore $y \leq n_1x/2^{64} \leq nx/2^{128} < \sqrt{n}$. As a consequence, we have $z > 0$ at step 3.

The proof of Theorem 5 is very similar to that of Theorem 4, and can be found in the appendix.

The largest difference we found is 24, for example with $n = 2^{64} \cdot 18355027010654859995 - 1$, taking $x = \lfloor 2^{96}/\sqrt{n_1} \rfloor - 15$, whereas the actual code might use a larger value. The smallest difference we found is -1 , for example with $n = 2^{64} \cdot 5462010773357419421 - 1$, taking $x = \lfloor 2^{96}/\sqrt{n_1} \rfloor$.

III. FAITHFUL ROUNDING

In addition to the usual rounding modes, faithful rounding (`MPFR_RNDF`) will be supported in MPFR 4 by the most basic operations. We say that an operation is faithfully rounded if its result is correctly rounded toward $-\infty$ or toward $+\infty$. The actual rounding direction is indeterminate: it may depend on the argument, on the platform, etc. (except when the true result is exactly representable, because in this case, this representable value is always returned, whatever the rounding mode). From this definition, the error on the operation is bounded by 1 ulp, like in the directed rounding modes. Moreover, the ternary value is unspecified for `MPFR_RNDF`.

The advantage of faithful rounding is that it can be computed more quickly than the usual roundings: computing a good approximation (say, with an error less than $1/2$ ulp) and rounding it to the nearest is sufficient. So, the main goal of `MPFR_RNDF` is to speed up the rounding process in internal computations. Indeed, we often do not need correct rounding at this level, just a good approximation and an error bound.

In particular, `MPFR_RNDF` allows us to completely avoid the *Table Maker's Dilemma* (TMD), either for the rounding or for the ternary value.

MPFR provides a function `mpfr_can_round` taking in entry: an approximation to some real, unknown value; a corresponding error bound (possibly with some given direction); a target rounding mode; a target precision. The goal of this function is to tell the caller whether rounding this approximation to the given target precision with the target rounding mode necessarily provides the correct rounding of the unknown value. For correctness, it is important that if the function returns true, then correct rounding is guaranteed (no false positives). Rare false negatives could be acceptable, but for reproducibility, it was chosen to never return a false negative with the usual rounding modes. When faithful rounding was introduced, a new semantic had to be chosen for the `MPFR_RNDF` rounding mode argument: true means that rounding the approximation in an adequate rounding mode can guarantee a faithful rounding of the unknown value. Reproducibility was no longer important as in general with faithful rounding. One reason to possibly accept false negatives here was to avoid the equivalent of the TMD for this function. However, a false negative would often mean a useless recomputation in a higher precision. Since it is better to spend a bit more time (at most linear) to avoid a false negative than getting a very costly reiteration in a higher precision, it was chosen to exclude false negatives entirely, like with the other rounding modes.

	MPFR 3.1.5		MPFR 4.0-dev	
	53	113	53	113
precision	53	113	53	113
mpfr_add	42	43	18	22
mpfr_sub	42	43	20	24
mpfr_mul	42	56	22	32
mpfr_sqr	61	66	17	29
mpfr_div	115	127	45	64
mpfr_sqrt	149	236	43	69

TABLE I
AVERAGE NUMBER OF CYCLES FOR BASIC OPERATIONS.

	T-RNDN	F-RNDN	F-RNDF
mpfr_add	17.6	17.7	17.6
mpfr_sub	19.6	20.7	20.7
mpfr_mul	21.7	24.7	15.1
mpfr_sqr	16.9	19.4	13.2
mpfr_div	47.3	47.5	45.4
mpfr_sqrt	50.2	50.5	47.8

TABLE II
AVERAGE NUMBER OF CYCLES FOR BASIC OPERATIONS IN 53 BITS WITH ROUNDING TO NEAREST IN THE TRUNK (T-RNDN), ROUNDING TO NEAREST IN THE FAITHFUL BRANCH (F-RNDN), AND FAITHFUL ROUNDING IN THE FAITHFUL BRANCH (F-RNDF).

IV. EXPERIMENTAL RESULTS

We used a 3.2Ghz Intel Core i5-6500 for our experiments (Skylake microarchitecture), with only one core enabled to decrease turbo-boost effects, and revision 11249 of MPFR (aka MPFR 4.0-dev), and GCC 6.2.1 under Debian GNU/Linux testing (stretch). MPFR was configured with GMP 6.1.2 [5] and `--disable-shared`. The number of cycles were measured with the `tools/mbench` utility distributed with MPFR. To get stable results, we called the `mbench` binary with `numactl --physcpubind=0`, that binds it to `cpu 0`, and we ensured no other processes were running on the machine.

Table I compares the number of cycles for basic arithmetic operations — with rounding to nearest — between MPFR 3 and the upcoming version 4, with 53 and 113 bits of precision, corresponding to the `binary64` and `binary128` formats respectively. The speedup goes from a factor 1.7 (for the 113-bit division) to a factor 3.6 (for the 53-bit squaring). This improvement of basic arithmetic automatically speeds up the computation of mathematical functions, as demonstrated in Table IV (still for rounding to nearest). Indeed, the computation of a mathematical function reduces to basic arithmetic for the argument reduction or reconstruction, for the computation of Taylor series or asymptotic series, ... Here the speedup goes from 15% (for the 113-bit arc-tangent) to a factor 2.2 (for the 53-bit power).

Timings for the `mpfr_sum` function in the faithful branch⁷ are done with a special test (`misc/sum-timings`) because the input type, an array of MPFR numbers, is specific to this function, and we need to test various kinds of inputs. The inputs are chosen randomly with fixed parameters, corresponding to the first 5 columns⁸ of Table III: the size of the array (number of input MPFR numbers), the number of cancellation terms in the test (0 means no probable cancellation, 1 means

⁷`branches/faithful` revision 11121.

⁸These are the arguments of position 2 to 6 of `sum-timings`.

Parameters					RND*	RNDF
10^1	0	10^7	10^1	1	411	399
10^3	0	10^1	10^5	10^8	27216	20366
10^3	1	10^1	10^5	10^8	39639	32898
10^3	2	10^1	10^5	10^8	44025	35276
10^5	0	10^1	10^1	10^8	1656988	1034802
10^5	0	10^1	10^3	10^8	1393447	833711

TABLE III
NUMBER OF CYCLES FOR `MPFR_SUM` WITH THE USUAL ROUNDING MODES AND `MPFR_RNDF`.

precision	MPFR 3.1.5		MPFR 4.0-dev	
	53	113	53	113
mpfr_exp	3996	6787	2352	4253
mpfr_sin	3746	4907	2684	3838
mpfr_cos	2853	3553	1872	2819
mpfr_log	4473	7086	2325	5993
mpfr_atan	13937	20182	7175	11385
mpfr_pow	12915	18393	5704	11222

TABLE IV
NUMBER OF CYCLES FOR MATHEMATICAL FUNCTIONS.

some cancellation, 2 even more cancellation), the precision of the input numbers (here, all of them have the same precision), the precision of the result, and an exponent range e (1 means that the input numbers have the same order of magnitude, and a large value means that the magnitudes are very different: from 1 to 2^{e-1}). A test consists in several timings on the same data, and we consider the average, the minimum and the maximum, and if the difference between the maximum and the minimum is too large, the result is ignored. Due to differences in timings between invocations, 4 tests have been run. On some parameters, `MPFR_RNDF` is faster than the other rounding modes (its maximum timing is less than the minimum timing for the usual rounding modes): for a small array size, small output precision and no cancellation (first line of Table III), there is a small gain; when the input precision is small and the range is large (so that the input values do not overlap in general, meaning that the TMD occurs in the usual rounding modes), we can typically notice a 25% to 40% gain when there is no cancellation (lines 2, 5 and 6), and a bit less when a cancellation occurs (lines 3 and 4). `MPFR_RNDF` is never slower. Note that the kind of inputs have a great influence on a possible gain, which could be much larger than the one observed here (these tests have not been specifically designed to trigger the TMD).

V. CONCLUSION

This article presents algorithms for fast floating-point arithmetic with correct rounding on small precision numbers. The implementation of those algorithms in version 4 of MPFR gives a speedup of up to 3.6 with respect to MPFR 3. As a consequence, the computation of mathematical functions is greatly improved.

ACKNOWLEDGEMENTS The authors thank Patrick Pélissier who designed the `mbench` utility, which was very useful in comparing different algorithms and measuring their relative efficiency.

REFERENCES

- [1] BRENT, R. P., AND ZIMMERMANN, P. *Modern Computer Arithmetic*. No. 18 in Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2010. Electronic version freely available at <http://www.loria.fr/~zimmerma/mca/pub226.html>.
- [2] DE DINECHIN, F., ERSHOV, A. V., AND GAST, N. Towards the post-ultimate libm. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic* (Washington, DC, USA, 2005), ARITH'17, IEEE Computer Society, pp. 288–295.
- [3] FOUSSE, L., HANROT, G., LEFÈVRE, V., PÉLISSIER, P., AND ZIMMERMANN, P. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.* 33, 2 (2007), article 13.
- [4] GRAILLAT, S., AND MÉNISSIER-MORAIN, V. Accurate summation, dot product and polynomial evaluation in complex floating point arithmetic. *Information and Computation* 216 (2012), 57 – 71. Special Issue: 8th Conference on Real Numbers and Computers.
- [5] GRANLUND, T., AND THE GMP DEVELOPMENT TEAM. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.2 ed., 2016. <http://gmplib.org/>.
- [6] IEEE standard for floating-point arithmetic, 2008. Revision of ANSI-IEEE Standard 754-1985, approved June 12, 2008: IEEE Standards Board.
- [7] JOHANSSON, F. Efficient implementation of elementary functions in the medium-precision range. In *Proceedings of the 22nd IEEE Symposium on Computer Arithmetic* (Washington, DC, USA, 2015), ARITH'22, IEEE Computer Society, pp. 83–89.
- [8] JOHANSSON, F. Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic. <https://arxiv.org/abs/1611.02831>, 2016. 12 pages.
- [9] KARP, A. H., AND MARKSTEIN, P. High-precision division and square root. *ACM Trans. Math. Softw.* 23, 4 (Dec. 1997), 561–589.
- [10] MASCARENHAS, W. Moore: Interval arithmetic in modern C++. <https://arxiv.org/pdf/1611.09567.pdf>, 2016. 8 pages.
- [11] MÖLLER, N., AND GRANLUND, T. Improved division by invariant integers. *IEEE Trans. Comput.* 60, 2 (2011), 165–175.
- [12] REVOL, N., AND ROUILLEIR, F. Motivations for an arbitrary precision interval arithmetic and the MPFI library. In *Reliable Computing* (2002), pp. 23–25.
- [13] ZIMMERMANN, P. Karatsuba square root. Research Report 3805, INRIA, 1999. <http://hal.inria.fr/inria-00072854>.

APPENDIX

Proof of Theorem 5. Write $n_1 = \alpha 2^{64}$ with $1/4 \leq \alpha < 1$. Write $x = 2^{96}/\sqrt{n_1} - \delta_x$, $y = 2^{32}\sqrt{n_1} - \delta_y$ and $t = xz/2^{65} - \delta_t$ with $0 \leq \delta_x < \delta$, and $0 \leq \delta_y, \delta_t < 1$.

$$\begin{aligned} 2^{128}n - s^2 &= 2^{128}n - (y \cdot 2^{64} + t)^2 \\ &= 2^{128}(n - y^2) - 2^{65}yt - t^2 \\ &= 2^{128}z - 2^{65}yt - t^2. \end{aligned} \quad (8)$$

We first bound $2^{128}n - s^2$ by above, assuming $2^{128}n - s^2 \geq 0$:

$$\begin{aligned} 2^{65}yt &= xyz - \delta_t 2^{65}y \\ &= \left(\frac{2^{96}}{\sqrt{n_1}} - \delta_x\right)(2^{32}\sqrt{n_1} - \delta_y)z - \delta_t 2^{65}y, \end{aligned} \quad (9)$$

thus

$$\begin{aligned} 2^{128}z - 2^{65}yt &\leq (2^{32}\sqrt{n_1}\delta_x + 2^{96}/\sqrt{n_1}\delta_y)z + 2^{65}y \\ &= 2^{64}z(\sqrt{\alpha}\delta_x + \delta_y/\sqrt{\alpha}) + 2^{65}y \\ &\leq 2^{64}z(\delta\sqrt{\alpha} + 1/\sqrt{\alpha}) + 2^{129}\sqrt{\alpha}, \end{aligned} \quad (10)$$

where we used $y \leq 2^{32}\sqrt{n_1} = \sqrt{\alpha}2^{64}$. Now

$$\begin{aligned} z &= 2^{64}n_1 + n_0 - (2^{32}\sqrt{n_1} - \delta_y)^2 \\ &\leq n_0 + 2^{33}\sqrt{n_1}\delta_y \\ &< 2^{64}(1 + 2\sqrt{\alpha}\delta_y) \\ &\leq 2^{64}(1 + 2\sqrt{\alpha}). \end{aligned} \quad (11)$$

Substituting Eq. (11) in Eq. (10) yields:

$$2^{128}z - 2^{65}yt \leq 2^{129}f(\alpha),$$

with

$$f(\alpha) = \frac{1}{2}(1 + 2\alpha^{1/2})(\delta\alpha^{1/2} + 1/\alpha^{1/2}) + \alpha^{1/2}.$$

Substituting in Eq. (8) gives

$$2^{128}n - s^2 < 2^{129}f(\alpha).$$

Let c be the real number such that $2^{128}n = (s + c)^2$. Then since $2^{128}n - s^2 = (s + c)^2 - s^2 = 2sc + c^2$, which implies $2sc < 2^{129}f(\alpha)$, thus

$$c < 2^{128}f(\alpha)/s. \quad (12)$$

Since $s \geq 2^{127}$, and the maximum of $f(\alpha)$ for $1/4 \leq \alpha \leq 1$ is 26.5 (attained at $\alpha = 1$), we get $c < 53$. Now this gives $s > \sqrt{2^{128}n - 53} > \sqrt{2^{128}n}/1.01 \geq \sqrt{\alpha}2^{128}/1.01$, therefore Eq. (12) becomes $c < 1.01 \cdot f(\alpha)/\sqrt{\alpha}$. Now the function $f(\alpha)/\sqrt{\alpha}$ is bounded by 26.5 for $1/4 \leq \alpha \leq 1$, the maximum still attained at $\alpha = 1$. Therefore $c < 1.01 \cdot 26.5 < 26.8$, which proves the upper bound 26.

Now assume $2^{128}n - s^2 < 0$. It follows from Eq. (9) that $2^{128}z - 2^{65}yt \geq 0$. Eq. (8) therefore yields $2^{128}n - s^2 \geq -t^2$, and we have to bound t^2 . Using $x \leq 2^{96}/\sqrt{n_1}$, Eq. (11) and $n_1 = \alpha 2^{64}$:

$$t^2 \leq \frac{x^2 z^2}{2^{130}} \leq \frac{2^{62} z^2}{n_1} \leq 2^{128} \frac{(1 + 2\sqrt{\alpha})^2}{4\alpha}.$$

Writing $2^{128}n = (s - c)^2$ with $c > 0$ yields:

$$\begin{aligned} (s - c)^2 - s^2 &> -2^{128} \frac{(1 + 2\sqrt{\alpha})^2}{4\alpha}, \\ 2sc - c^2 &< 2^{128} \frac{(1 + 2\sqrt{\alpha})^2}{4\alpha}. \end{aligned}$$

Now $s^2 > 2^{128}n$ implies $s > 2^{64}\sqrt{n} \geq 2^{128}\sqrt{\alpha}$, thus:

$$c < \frac{c^2}{2^{129}\sqrt{\alpha}} + \frac{1}{8} \frac{(1 + 2\sqrt{\alpha})^2}{\alpha^{3/2}}. \quad (13)$$

This implies for all $1/4 \leq \alpha < 1$:

$$c < \frac{c^2}{2^{128}} + 4,$$

which proves $c < 4.01$, thus the lower bound is $s - 4$. \square