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, libibieep17882, 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 $\beta$ 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 $e_{\text{min}}$ are written $a0$ and $\text{emin}$ 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 $\pm 0$ — the most significant bit of $m[n-1]$ must be set, i.e., the number is always normalized4. The corresponding number is:

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

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 sh trailing 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):

$$\begin{array}{cccc}
1xxxxxxx & xxxxxxxx & \cdots & xxxxxxx000 \\
\text{m[n-1]} & \text{m[n-2]} & \cdots & \text{m[0]} \\
\end{array}$$

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 $\text{mpfr_sub}(a, b, a, \text{rnd_mode})$ 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);

1https://octave.sourceforge.io/interval/
2https://github.com/nehmeier/libibieep1788
3http://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_addlspl` 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: \( b[0] \) and \( c[0] \), and their exponents are \( b \) and \( c \). Since \( b \) and \( c \) are regular numbers, we have \( 2^{63} \leq b[0], c[0] < 2^{64} \). If \( b = c \), the addition of the significands always produces a carry: \( 2^{64} \leq b[0] + c[0] < 2^{65} \):

\[
\begin{align*}
bp[0] &= 1yyyyyyyy000 \\
1yyyyyyyy000 &\text{cp[0]}
\end{align*}
\]

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

\[
a_0 = (bp[0] >> 1) + (cp[0] >> 1); \\
b_0 += 1;
\]

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

\[
\begin{align*}
r_b &= a_0 \& (\text{ONE} << (sh - 1)); \\
ap[0] &= a_0 \& r_b; \\
sb &= 0;
\end{align*}
\]

The case \( b \times c \) is treated similarly: we have to shift \( cp[0] \) by \( d = b \times c \) bits to the right. We distinguish three cases:

- \( d < sh \) where the shifted \( cp[0] \) fully overlaps with \( ap[0] \), \( sh \leq d < 64 \), 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 \( bx < cx \), we simply swap the inputs and reduce to the \( bx > cx \) 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 \( bx > emax \). Otherwise we set the exponent of \( a \) to \( bx \). If \( rb = sb = 0 \), 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 \( rb = 0 \), or \( rb \neq 0 \) and \( sb = 0 \) 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 \( rb = sb = 0 \) 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_sublspl` function, in the case where the exponent difference \( d = bx - cx \) 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):

\[
\begin{align*}
t &= (cp[1] << (64 - d)) | (cp[0] >> d); \\
a_0 &= bp[0] - t; \\
a_1 &= (cp[1] >> d) - (bp[0] < t); \\
sb &= cp[0] < (64 - d); \\
if (sb) \{ \\
\quad a1 -= (a0 == 0); \\
\quad a0 --; \\
\quad sb = -sb; /* 2^{64} - sb */ \\
\}
\end{align*}
\]

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 \( sb = 0 \). 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 \( sb = 0 \)):

\[
\begin{align*}
if (a1 == 0) \{ \\
\quad a1 = a0; \\
\quad a0 = 0; \\
\quad bx -= 64; \\
\}
\end{align*}
\]

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

\[
\begin{align*}
\text{count_leading_zeros} (cnt, a1); \\
if (cnt) \{ \\
\quad ap[1] = (a1 << cnt) | (a0 >> (64-cnt)); \\
\quad a0 = (a0 << cnt) | (sb >> (64-cnt)); \\
\quad sb <<= cnt; \\
\quad bx -= cnt; \\
\} \quad \text{else} \quad ap[1] = a1;
\end{align*}
\]

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 \( mask = 2^{sh} - 1 \):

\[
\begin{align*}
rb &= a0 \& (\text{ONE} << (sh - 1)); \\
sl &= (a0 \& mask) \^ rb; \\
ap[0] &= a0 \& -mask;
\end{align*}
\]

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 \(a_0\) and \(sb\) here:

\[
ax = \text{EXP}(b) + \text{EXP}(c);
\]

\[
\text{umul_ppmm}(a_0, \, sb, \, bp[0], \, cp[0]);
\]

Since \(2^{63} \leq \text{bp}[0], \text{cp}[0] < 2^{64}\), we have \(2^{126} \leq \text{bp}[0]\text{cp}[0] = a_0 \cdot 2^4 + sb < 2^{128}\). 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:

\[
\text{if} \ (a_0 < \text{HIGHBIT}) \{
ax --;
ax = (ax < 1) \mid (sb >> 63);
\}
\]

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

\[
rb = a_0 \& (\text{ONE} \ll (sh - 1));
\]

\[
sb \mid = (a_0 \& \text{mask}) \ll rb;
\]

\[
\text{ap}[0] = a_0 \ll \text{mask} - 64;
\]

\[
\text{SIGN}(a) = \text{MUL\_SIGN}(\text{SIGN}(b), \text{SIGN}(c));
\]

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 is an exponent smaller that \(e_{\text{min}}\). Thus for example, when \(ax = e_{\text{min}} - 1\), \(rb\) or \(sb\) is non-zero. \(ap[0] = 111111111100000000\) 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 \(ax = e_{\text{min}} - 1\) and nevertheless no underflow occurs cannot happen in the subtraction, since \(b\) and \(c\) are multiples of \(2^{\text{word}-r}\), likewise for \(b - c\); thus if the exponent \(ax\) of the difference is smaller than \(e_{\text{min}}\), 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 \([(\beta^2 - 1)/v] - \beta\), 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 \(\beta^p/2 \leq u, v < \beta^p\). If that occurs, \(v\) is subtracted from \(u\), which generates an extra leading quotient bit. After that subtraction, we get \(u - v < v\) since \(u < \beta^p \leq 2v\). 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 \(0 \leq u \leq v < \beta\) and \(\beta/2 \leq v < \beta\)

**Output:** integer \(q\) approximating \(u\beta/v\)

1: compute an approximate reciprocal \(i\) of \(v\), satisfying

\[
i \leq \left(\frac{(\beta^2 - 1)}{v} - \beta\right) \leq i + 1
\]

2: \(q = \lceil iu/\beta \rceil + u\)

---

**Proof.** Step 2 of DivApprox1 is simply step 1 of Algorithm 1 (DIV2BY1) from [11]. If \(i\) is the exact reciprocal \(i_0 := \frac{(\beta^2 - 1)}{v} - \beta\), and \(q_0 := \lceil i_0 u/\beta \rceil + u\) 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 \(q_0 \leq q \leq q_1 + 1\), thus \(r = \beta u - q v\) satisfies \(r < 3v\).

As a consequence of Theorem 1, we can decide the correct rounding except when the last \(sh\) 1 bits from \(q\) are \(000\ldots000\), \(111\ldots110\), or \(111\ldots111\), 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 \(r = \beta u - q v\), 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 \(\beta\).

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
This proves that $r < 5\beta$ at step 4. From $r < 2\alpha + 2\alpha^2$ it follows $q_0 \leq r x / \beta < (2 + 2\alpha + 2\alpha^2) x$, thus since $x \leq \beta^2 / (v_1 + 1) < \beta / \alpha$, $q_0 < (2/\alpha + 1 + 2\alpha) \beta$ (note that $q_0$ might exceed $\beta$). Let $r x = q_0 \beta + \delta_r$, and $u \beta - q_1 v = \beta r + \delta_r$, with $0 \leq \delta_r, \delta_r < \beta$. Then

$$u \beta^2 - q v = \beta (\beta r + \delta_r) - q_0 v = \beta^2 r + \beta \delta_r - q_0 (v_1 + 1) + \beta q_0 (\beta - v_0) = \beta^2 r + \beta \delta_r - (r x - \delta_r)(v_1 + 1) + q_0 (\beta - v_0) = \beta^2 r + \beta \delta_r - r (\beta^2 - \kappa) + \delta_r (v_1 + 1) + q_0 (\beta - v_0) = \beta \delta_r + r x + \delta_r (v_1 + 1) + q_0 (\beta - v_0).$$

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

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

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

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

The largest error we could find is 20, for example for $\beta = 4096$, $u = 8298491$, $v = 8474666$. With $\beta = 2^{64}$, on 107 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

In practice, like in Algorithm DivApprox1, $x = \beta + x'$ is split between one implicit upper bit $\beta$ and a lower part $x'$, that for simplicity we do not distinguish here. In the real code, step 3 therefore becomes $q_1 = u_1 + \lfloor u_1 x' / \beta \rfloor$, and step 5 becomes $q_0 = r + \lfloor r x' / \beta \rfloor$.\footnote{If $\beta^2 + 1$ 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 1/2 term disappears in Eq. (2), and the bound becomes $u \beta^2 / v < q + 21$}
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:

\[
v_3 \leq s := \frac{\lceil 2^{66} / \sqrt{d} \rceil}{v_3 + 8}.
\]

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

\[
x_{k+1} = x_k + \frac{x_k}{2}(1 - a_k + x_k^2),
\]

where \( a_0 \geq a_1 \geq \cdots \geq a_n \geq a > 0 \), then:

\[
x_1 \leq x_2 \leq \cdots \leq x_n \leq a^{-1/2}.
\]

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 \( 1 - a_k x_k^2 \geq 0 \). Put into (3) this proves \( x_{k+1} \leq x_k \), and thus the lemma since \( x_{k+1} \leq a_{k+1}^{-1/2} \) and \( x_{k+1} \geq a \) imply \( x_{k+1} \leq a^{-1/2} \).

This lemma remains true when the correction term \( \frac{x_k}{2}(1 - a_k + x_k^2) \) in Eq. (3) — which is thus non-negative — is rounded down towards zero. By applying this result to \( x_0 = 0/2^{10}, x_1 = v_1/2^{21}, x_2 = v_2/2^{31}, x_3 = v_3/2^{64} \), with \( a_0 = d_10/2^{10} \geq a_1 = d_37/2^{37} \geq a_2 \geq a_3 \geq a = d/2^{64} \), it follows that \( 1 \leq v_0/2^{10} \leq v_1/2^{21} \leq v_2/2^{31} \leq v_3/2^{64} \leq 1/2 \). Thus \( 2^{10} \leq v_0 \leq 2^{11}, 2^{21} \leq v_1 < 2^{22}, 2^{31} \leq v_2 < 2^{32}, 2^{64} \leq v_3 < 2^{65} \). 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 \( 2^{57} - v_0^2d_{37} \) 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 \( v_0e_0 \) will have at most 61 bits.

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

\[
2^{-47}v_0e_0 = [2^{-47}v_0e_0] + \delta_1.
\]

Then:

\[
e_1 = 2^{79} - v_0^2d_{37} = 2^{79} - d_{37}(v_0 + \delta_1 - \delta_1)^2 = 2^{79} - d_{37}(v_0 + \delta_1)^2 + d_{37}\delta_1(2v_0 + \delta_1).
\]

Since \( v_0^2d_{37} \leq 2^{79} \) — because \( e_1 \geq 0 \) — then \( v_0d_{37} \leq 2^{79}/v_0 \leq 2^{58} \) because \( 2^{21} \leq v_0 \). It follows that \( e_1 := d_{37}\delta_1(2v_0 + \delta_1) \) satisfies \( 0 \leq e_1 \leq 2^{59} + 2^{47} \), and:

\[
e_1 - e_1 = 2^{79} - d_{37}(v_0 + \delta_1)^2 = 2^{79} - d_{37}(2v_0 + \delta_1)^2 + 2^{79} + 2^{-47}v_0e_0)^2 = 2^{79} - d_{37}v_2^2(2v_0 + \delta_1)^2.
\]

Now, using \( v_0^2d_{37} = 2^{57} - e_0 \):

\[
e_1 - e_1 = [2^{79} - (5^{57} - e_0)(2^{11} + 2^{-47}e_0)^2 = 2^{79} - d_{37}v_2^2(2v_0 + \delta_1)^2.
\]

Since \( e_0 < 2^{49.263} \) we deduce \( 2^{34} + 2^{-49}e_0 < 2^{58.8} \) and:

\[
0 \leq e_1 < 2^{59} - 2^{-49}2^{52}2^{-0.412} + 2^{59} + 2^{47} < 2^{63.196}.
\]

Therefore \( e_1 < 2^{64} \) and \( e_1 \) can be computed using integer arithmetic modulo \( 2^{64} \). Since \( d_{10} = [2^{-27}(d_{37} - 1)] + 1, e_1 \) only depends on \( d_{37} \), so that we can perform an exhaustive search on the \( 2^{37} - 2^{37} \) possible values of \( d_{37} \). By doing this, we find that the largest value of \( e_1 \) is obtained for \( d_{37} = 33 \cdot 32 + 1 \), which corresponds to \( d_{10} = 265 \); this gives \( e_1 = 10263103231127343388 < 2^{63.155} \).

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

\[
2^{-70}v_1e_1 = [2^{-70}v_1e_1] + \delta_2.
\]

Then:

\[
e_2 = 2^{126} - v_2d = 2^{126} - d(2v_2 + \delta_2 - \delta_2)^2 = 2^{126} - d(2v_2 + \delta_2)^2 + d\delta_2(2v_2 + \delta_2).
\]

Since \( v_2 \geq 2^{31} \) and \( v_2d \leq 2^{126}, v_2d \leq 2^{95} \), thus \( d\delta_2(2v_2 + \delta_2) < 2^{99} + 2^{64} \):

\[
e_2 - (2^{96} + 2^{64}) \leq 2^{126} - d(2v_2 + \delta_2)^2 = 2^{126} - d(2v_2 + \delta_2(2v_2 + \delta_2))^2 = 2^{126} - d2^2(2v_2 + \delta_2)^2.
\]

Now writing \( d = 2^{27}d_{37} - \rho \) with \( 0 < \rho < 2^{27} \), using \( d_{37}v_2^2 < 2^{79} - e_1 \), and writing \( e = 2^{96} + 2^{64} + \rho e_1^2(2^{10} + 2^{70}e_1)^2 \):

\[
e_2 - e \leq 2^{126} - 2^{27}(2^{79} - e_1)(2^{10} + 2^{70}e_1)^2 = 2^{126} - (2^{79} - e_1)(2^{27} + 2^{32}e_1 + 2^{11}3e_1)^2 = 2^{126} - (2^{27} - 2^{32}e_1 + 2^{34}e_1^2 + 2^{11}3e_1)^2 = 2^{32}e_1^2(3/4 + 2^{-81}e_1).
\]

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

\[
\rho e_1^2(2^{10} + 2^{70}e_1)^2 \leq 2^{71}(2^{20} + 2^9 + 2^{-12}).
\]

Thus \( \rho e_1^2 < 2^{96} + 2^{64} + 2^{91} + 2^{76} + 2^{59} \) and:

\[
0 < e_2 < 2^{-32}2^{126.31}2^{-0.415} + \varepsilon < 2^{96.338}.
\]
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 $e_2 = 8161191994651931475229010064 < 2^{96}.043$.

The final error is estimated using the truncation error $\delta_3$ in step 9:

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

where $v_2 = 2^{192} - \delta_3 = 2^{192} - d((v_3 + \delta_3) - \delta_3)^2$

Since $v_3^2d \leq 2^{192}$, and $v_3 \geq 2^{64}$, if $v_3d \leq 2^{128}$, then $\gamma_3 := d_3(2v_3 + \delta_3) < 2^{29} + 2^{64}$.

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

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

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

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

$$0 \leq e_3 < 2^{-60}2^{192}.0862^{0.415} + 2^{129} + 2^{64} < 2^{131.882}.$$

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 $2v_3d < 2^{131.878}$, 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 $c < 2^{37.879} < 15$.

By doing again this error analysis for a given value of $d_{37}$, we get a tighter bound involving the $\delta_3$ truncation errors. For example we can bound $d_{37}\delta_3(2v_1 + \delta_1)$ by $d_{37}(2v_1}\max + 1)$, where $v_1\max = 2^{37}/d_{37}$. This yields a finer bound. By exhaustive search on all possible $d_{37}$ values, we found the maximal error is at most 8.

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 $[2^{-94}v_2e_2']$, 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_sqrt1l function: `mpfr_sqrt1l` 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 $2^{64}/\sqrt{n}$ 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 $x^2n < 2^{126}$, $xn/2^{31} < 2^{85}/n < 2^{32}$, thus $x[2^{-31}n]$ fits on 64 bits.

Algorithm 4 SqrtApprox1

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

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

1: compute an integer $x$ approximating $2^{64}/\sqrt{n}$ with

$$x \leq 2^{65}/\sqrt{n}$$

(2: $y = \lfloor\sqrt{n}\rfloor$)

3: $z = n - y^2$

4: $t = [2^{-32}x\bar{z}]$

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

$$s \leq \lfloor\sqrt{2^{64}/n}\rfloor \leq s + 7.$$

**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 $2^{63} - x\sqrt{n} = \delta_x\sqrt{n} < 2^{33.339}$.

$$2^{64}/\sqrt{n} - s^2 = 2^{64}/\sqrt{n} - (y \cdot 2^{32} + t)^2 = 2^{64}/(n - y^2) - 2^{33}yz - t^2 = 2^{64} - 2^{33}yz - t^2.$$ (5)

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

$$2^{33}yz = 2xyz - 2^{33}\delta_t y = 2^{24}/\sqrt{n} - 2^{33}\delta_t y$$

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

$$2^{64} - 2^{33}yz \leq (2^{64}/\sqrt{n} + 2^{64}/\sqrt{n})z + 2^{33}y = 2^{23}z(2^{33.339}\sqrt{\alpha + 1/\sqrt{\alpha}} + 2^{65}/\sqrt{\alpha},$$

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

$$z = n - (\sqrt{n} - \delta_y)^2 \leq 2\sqrt{n}\delta_y < 2^{33}\alpha^{1/2}.$$ Substituting this in the bound for $2^{64}/n - s^2$ gives:

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

with

$$f(\alpha) = 2^{33.339}\alpha + 1 + \alpha^{1/2}.$$
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
\[
c < 2^{64} f(\alpha)/s.
\]
(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 \( 2^{64} z - 2^{63} yt \geq 0 \) by Eq. (6), we have \( 2^{64} n - s^2 \geq -t^2 \) by Eq. (5). Since \( x \leq 2^{63}/\sqrt{n} \) and \( z \leq 2^{33} \sqrt{n} \):
\[
t \leq -32xz < 2^{32}.
\]

(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 \( c > 0 \):
\[
2^{64} n - s^2 \geq -t^2 > -2^{64}.
\]
\[
(s - c)^2 - s^2 > -2^{64}.
\]
\[
2sc < 2^{64} + c^2.
\]

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 \( u \geq 1 \).

3) The \texttt{mpfr_sqrt2} function: The \texttt{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 \( \delta = 16 \). Algorithm 5

**Algorithm 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 \( 2^{64}/\sqrt{n_1} \) with \( 0 \leq 2^{64}/\sqrt{n_1} - x < \delta := 16 \)
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. \( s = y \cdot 2^{64} + t \)

Algorithm 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 \( x \leq 2^{66}/\sqrt{n_1 + 1} < 2^{128}/\sqrt{n} \), therefore \( y \leq n_1 x/2^{64} \leq n x/2^{128} < \sqrt{n} \). 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 \( x = \lfloor 2^{66}/\sqrt{n_1} \rfloor - 15 \), 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 \( x = \lfloor 2^{66}/\sqrt{n_1} \rfloor \).

### III. FAITHFUL Rounding

In addition to the usual rounding modes, faithful rounding (\texttt{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 \texttt{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 \texttt{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, \texttt{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 \texttt{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 \texttt{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.
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 (stretch). MPFR was configured with GMP 6.1.2 and --disable-shared [5] and --numactl --physcpubind=0. 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 branch 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 columns 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 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.
Substituting Eq. (11) in Eq. (10) yields:

\[ 2^{128} z - 2^{65} y t \leq 2^{129} f(\alpha), \]

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

\[ c < 2^{128} f(\alpha)/s. \]

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 \( s > \sqrt{2^{128}n - 53} > \sqrt{2^{128}n}/1.01 \geq \alpha^{2^{128}}/1.01 \), 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 \( 2^{128} z - 2^{65} y t > 0 \). Eq. (8) therefore yields \( 2^{128} n - s^2 \geq -t^2 \), and we have to bound \( t^2 \). Using \( x \leq 2^{96}/\sqrt{\alpha} \), Eq. (11) and \( n_1 = 2^{64} \):

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

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

\[ (s - c)^2 - s^2 > -2^{128} \frac{(1 + 2\sqrt{\alpha})^2}{4\alpha}, \]

\[ 2sc - c^2 < 2^{128} \frac{(1 + 2\sqrt{\alpha})^2}{4\alpha}. \]

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

\[ c < \frac{c^2}{2^{129} \sqrt{\alpha}} + \frac{1}{8} \left(1 + \frac{2\sqrt{\alpha}}{\alpha^{3/2}}\right). \]

This implies for all \( 1/4 \leq \alpha \leq 1 \):

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

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