The mathematical framework for the `float128` math library

Thrackle Research

April 2, 2025

Introduction

The goal of this document is to explain the technical framework in which we built the `float128` library.

In the first section, we will first describe our representation of floats. A float in our framework uses 128 bits for the significant digits (this means that we will work with 38 significant digits base 10 because $10^{38} < 2^{128} < 10^{39}$). In the second section, we will outline our algorithms for the mathematical operations in the library. In the third section, we will discuss how to quantify the propagation of errors as the various operations are chained. Finally, we will conclude with a discussion of some practical implications that could be useful for solidity practitioners space who want to make use of this library.

1 Float Representation

An instance of the class `float128` is defined by a tuple (a, b) of signed integers. This tuple will represent the real number $a \cdot 10^b$. The parameter a is called `digits` or `mantissa` and the parameter b is called `exp` or `exponent`.

For example, in Python this looks like the following class:

Listing 1: The `float128` class

```
class float128:
    def __init__(self, a: int, b: int):
        assert type(a) == int and type(b) == int
        assert abs(a) < 10**38
        assert abs(b) < 10**38
        self.digits = a
        self.exp = b
```

The motivation for only using 128 bit integers is so that we can operate on two of them as an intermediary step using built in solidity integer operations within the native type `int256` (for example, multiplying to 128 bit integers in solidity will not lead to overflow). Note further that optimizations have been made on the solidity side to pack these two quantities (mantissa and exponent) into one int256 type, but these specifics lie outside of the scope of this document and are unnecessary for it.

Finally, as it is common from other float libraries to use the IEEE 754 quadruple-precision binary floating-point numbers, users of these libraries often must convert back and forth between base 2 and base 10 (if not computationally, there may be mental overhead and potential loss of precision). In this way, our library offers users an alternative in which they can store base 10 quantities (such as token amounts) in a more canonical way.

2 Outline of the Algorithms

2.1 Algorithms for Addition, Subtraction, Multiplication, and Division

For simplicity, we will not describe the four basic mathematical operations beyond including the Python code for the algorithms here. Note that subtraction and addition are unified since we allow negative numbers to be used.

Listing 2: Auxiliary functions

```
def number_of_digits(x: int):
    # Computes the number of digits of x (in base 10).
    if x == 0:
        return 0
    else:
        return(len(str(abs(x))))

def length(self):
        return number_of_digits(self.digits)
```

Listing 3: Addition

```
def add128(a: float128, b: float128) -> float128:
    assert a.digits < 10**38
    assert b.digits < 10**38

    # The result will be computed in a variable s, and thus
    # we will compute s_digits and s_exponent.

    if a.digits == 0:
        return b
    if b.digits == 0:
        return a

    order_a = a.length() + a.exp
    order_b = b.length() + b.exp

    order_s = max(order_a, order_b)
    base_exp = order_s - 75

    # We adequate both a and b to the base exponent.
    delta_a = base_exp - a.exp
    # We want that: exp_a = a.exp + delta_a = base_exp
    if delta_a == 0:
        digits_a = a.digits
    elif delta_a > 0:
        digits_a = a.digits // 10**delta_a
    else:
        digits_a = a.digits * 10**(-delta_a)

    delta_b = base_exp - b.exp
    # We want that: exp_b = b.exp + delta_b = base_exp
    if delta_b == 0:
        digits_b = b.digits
    elif delta_b > 0:
```

```
        digits_b = b.digits // 10**delta_b
    else:
        digits_b = b.digits * 10**(-delta_b)

    if digits_a + digits_b == 0:
        return float128(0,0)

    n_digits = number_of_digits(digits_a + digits_b)
    extra_digits = n_digits - 38
    if extra_digits < 0:
        extra_digits = 0
    digits_s = (digits_a + digits_b) // 10**extra_digits
    exp_s = base_exp + extra_digits

    return float128(digits_s, exp_s)
```

Listing 4: Multiplication

```
def mul128(a: float128, b: float128) -> float128:
    assert a.digits < 10**38
    assert b.digits < 10**38

    m_digits_1 = a.digits * b.digits
    m_digits, exp_1 = truncate128(m_digits_1)
    m_exponent = a.exp + b.exp + exp_1

    if m_digits == 0:
        m_exponent = 0

    return float128(m_digits, m_exponent)
```

Listing 5: Division

```
def div128(a: float128, b: float128) -> float128:
    assert a.digits < 10**38
    assert b.digits < 10**38

    extra_zeroes = 76 - a.length()
    q_digits_1 = (a.digits * 10**extra_zeroes) // b.digits
    q_digits, exp_1 = truncate128(q_digits_1)
    q_exponent = a.exp - b.exp + exp_1 - extra_zeroes

    if q_digits == 0:
        q_exponent = 0

    return float128(q_digits, q_exponent)
```

2.2 Algorithm for the Square Root

In this subsection we present an algorithm for computing the square root in the `float128` Python library. Due to gas consumption reasons, the algorithm employed for the Solidity `float128` library is different from this one.

As is rigorously defined below, this algorithm unlike the others requires the notion of relative error.

Definition 1. Let $a, \widetilde{a} \in \mathbb{R}$ such that $a \neq 0$. We define the *relative error* of $\widetilde{a}$ with respect to a as

$$\mathcal{E}(\widetilde{a}, a) = \left| \frac{\widetilde{a}}{a} - 1 \right| \ .$$

Proposition 2. *Let $a, \widetilde{a} \in \mathbb{R}_{>0}$. Then*

$$\mathcal{E}\left(\sqrt{\widetilde{a}}, \sqrt{a}\right) \leq \mathcal{E}(\widetilde{a}, a) \ .$$

If, in addition, $\mathcal{E}(\widetilde{a}, a) \leq 10^{-M}$ for some $M \in \mathbb{N}$, then

$$\mathcal{E}\left(\sqrt{\widetilde{a}}, \sqrt{a}\right) \leq \frac{\mathcal{E}(\widetilde{a}, a)}{2 - 10^{-M}}$$

Proof.

$$\mathcal{E}\left(\sqrt{\widetilde{a}}, \sqrt{a}\right) = \left| \frac{\sqrt{\widetilde{a}}}{\sqrt{a}} - 1 \right| = \frac{\left| \frac{\sqrt{\widetilde{a}}}{\sqrt{a}} - 1 \right| \cdot \left| \frac{\sqrt{\widetilde{a}}}{\sqrt{a}} + 1 \right|}{\left| \frac{\sqrt{\widetilde{a}}}{\sqrt{a}} + 1 \right|} = \frac{\left| \frac{\widetilde{a}}{a} - 1 \right|}{\left| \frac{\sqrt{\widetilde{a}}}{\sqrt{a}} + 1 \right|} = \frac{\mathcal{E}(a, \widetilde{a})}{\frac{\sqrt{\widetilde{a}}}{\sqrt{a}} + 1} \leq \mathcal{E}(\widetilde{a}, a) \ .$$

This proves the first claim.

Moreover, if there exists $M \in \mathbb{N}$ such that $\mathcal{E}(\widetilde{a}, a) < 10^{-M}$, then we obtain that

$$\begin{aligned} \mathcal{E}\left(\sqrt{\widetilde{a}}, \sqrt{a}\right) = \frac{\left| \frac{\widetilde{a}}{a} - 1 \right|}{\left| \frac{\sqrt{\widetilde{a}}}{\sqrt{a}} + 1 \right|} = \frac{\left| \frac{\widetilde{a}}{a} - 1 \right|}{\left| 2 + \frac{\sqrt{\widetilde{a}}}{\sqrt{a}} - 1 \right|} \leq \frac{\left| \frac{\widetilde{a}}{a} - 1 \right|}{2 - \left| \frac{\sqrt{\widetilde{a}}}{\sqrt{a}} - 1 \right|} = \frac{\mathcal{E}(\widetilde{a}, a)}{2 - \mathcal{E}(\sqrt{\widetilde{a}}, \sqrt{a})} \leq \frac{\mathcal{E}(\widetilde{a}, a)}{2 - \mathcal{E}(\widetilde{a}, a)} \leq \\ \leq \frac{\mathcal{E}(\widetilde{a}, a)}{2 - 10^{-M}} \ , \end{aligned}$$

as we wanted to prove. □

Algorithm for computing the square root

We will now present an algorithm to compute the square root of a `float128` number with high precision in few steps. The algorithm is based in the well-known Babylonian method, which is just the Newton's method applied to a quadratic function of the form $x^2 - d$.

Suppose we want to compute the square root of the `float128` number given by $a \cdot 10^b$, with $a \in \mathbb{N}$ and $b \in \mathbb{Z}$. In order to do so, we proceed as follows.

Let $l = \lfloor \log_{10}(a) \rfloor$. Then

$$10^l \leq a < 10^{l+1} \ .$$

Hence,

$$10^{l+b} \leq a \cdot 10^b < 10^{l+b+1} \ .$$

By the algorithm of the division, there exist $q \in \mathbb{Z}$ and $r \in \{0, 1\}$ such that $l + b = 2q + r$. Thus,

$$10^{2q+r} \leq a \cdot 10^b < 10^{2q+r+1} \ .$$

Hence,

$$10^r \leq a \cdot 10^{b-2q} < 10^{r+1} \ .$$

Then,

$$1 \leq a \cdot 10^{b-2q} < 100 \ .$$

For each $k \in \mathbb{Z}$ such that $0 \leq k \leq 17$ we define the interval I_k by

$$I_k = \left[\left(1 + \frac{k}{2}\right)^2, \left(1 + \frac{k+1}{2}\right)^2 \right)$$

It is not difficult to verify that

$$[1, 100) = \bigcup_{k=0}^{17} I_k$$

and that the intervals I_k, $0 \leq k \leq 17$ are pairwise disjoint.

Let $z = a \cdot 10^{b-2q}$ and let k_z be the unique index such that $z \in I_{k_z}$. Let $R_0 \in \mathbb{R}_{>0}$ be defined by

$$R_0 = 1 + \frac{k_z}{2} \ .$$

Note that R_0 is a first approximation of the square root of $z = a \cdot 10^b$. Moreover, since

$$z \in I_{k_z} = \left[\left(1 + \frac{k_z}{2}\right)^2, \left(1 + \frac{k_z+1}{2}\right)^2 \right)$$

we obtain that

$$R_0 = 1 + \frac{k_z}{2} \leq \sqrt{z} < 1 + \frac{k_z+1}{2} = R_0 + \frac{1}{2} \ .$$

Thus,

$$0 \leq \sqrt{z} - R_0 < \frac{1}{2} \ ,$$

and hence,

$$\left|R_0 - \sqrt{z}\right| < \frac{1}{2} \ .$$

This says that the absolute error of our first approximation of $\sqrt{z}$ is less than $\frac{1}{2}$.

Now, we apply Newton's method with the function $f(x) = x^2 - z$, and we define a sequence $(R_n)_{n\in\mathbb{N}}$ of successive approximations of $\sqrt{z}$ by

$$R_n = R_{n-1} - \frac{f(R_{n-1})}{f'(R_{n-1})} = R_{n-1} - \frac{(R_{n-1})^2 - z}{2R_{n-1}} = \frac{1}{2}\left(R_{n-1} + \frac{z}{R_{n-1}}\right) \ .$$

For each $n \in \mathbb{N}_0$, let $\varepsilon_n = \sqrt{z} - R_n$. Let $I = [\sqrt{z} - |\varepsilon_0| \, , \sqrt{z} + |\varepsilon_0|]$ and let

$$M = \frac{1}{2}\left(\sup_{x\in I} \left|f''(x)\right|\right)\left(\sup_{x\in I} \frac{1}{|f'(x)|}\right).$$

Note that

$$|\varepsilon_0| = \left|\sqrt{z} - R_0\right| = \sqrt{z} - R_0 \ .$$

Hence,

$$I = \left[\sqrt{z} - |\varepsilon_0| \, , \sqrt{z} + |\varepsilon_0|\right] = \left[R_0, 2\sqrt{z} - R_0\right] \ .$$

And since $f'(x) = 2x$ and $f''(x) = 2$, we obtain that

$$M = \frac{1}{2}\left(\sup_{x\in I} \left|f''(x)\right|\right)\left(\sup_{x\in I} \frac{1}{|f'(x)|}\right) = \frac{1}{2} \cdot 2 \cdot \left(\sup_{x\in I} \frac{1}{2x}\right) = \frac{1}{2R_0} \leq \frac{1}{2}.$$

Thus, from the arguments given in the Appendix, it follows that

$$|\varepsilon_{n+1}| \leq M \, |\varepsilon_n|^2 \leq \frac{1}{2} \, |\varepsilon_n|^2 \, .$$

Therefore, using induction, we obtain that, for all $n \in \mathbb{N}$,

$$|\varepsilon_n| \leq 2^{-(2^{n+1}-1)} \ .$$

This means that, for example, if $n = 7$ then the absolute error in R_n (with respect to $\sqrt{z}$) is lower than 2^{-255}.

It is important to observe that

$$\mathcal{E}(R_n, \sqrt{z}) = \left|\frac{R_n}{\sqrt{z}} - 1\right| = \left|\frac{R_n - \sqrt{z}}{\sqrt{z}}\right| = \frac{|\varepsilon_n|}{\sqrt{z}} \leq |\varepsilon_n|$$

since $z \geq 1$. Thus, in this case, the relative error $\mathcal{E}(R_n, \sqrt{z})$ is bounded by the absolute error $|\varepsilon_n|$.

In addition, recall that we wanted to compute the square root of the `float128` number given by $a \cdot 10^b$. Since

$$a \cdot 10^b = z \cdot 10^{2q}$$

we obtain that

$$\sqrt{a \cdot 10^b} = \sqrt{z \cdot 10^{2q}} = \sqrt{z} \cdot 10^q \ .$$

Thus, our approximate value for $\sqrt{a \cdot 10^b}$ will be $R_n \cdot 10^q$, and we obtain that

$$\mathcal{E}(R_n \cdot 10^q, \sqrt{a \cdot 10^b}) = \mathcal{E}(R_n \cdot 10^q, \sqrt{z} \cdot 10^q) = \mathcal{E}(R_n, \sqrt{z}) \leq |\varepsilon_n| \, .$$

Therefore, $|\varepsilon_n|$ is an upper bound for the relative error $\mathcal{E}(R_n \cdot 10^q, \sqrt{a \cdot 10^b})$.

It is also interesting to observe that, with more information about R_0, we might be able to ensure that the error is much lower. For example, if $k_z \geq 2$, then $R_0 \geq 2$ and thus $M \leq \frac{1}{4}$, from where we obtain that, for all $n \in \mathbb{N}$,

$$|\varepsilon_n| \leq 2^{-(2^{n+1}+2^n-2)} \ .$$

In particular, if $n = 7$ then the absolute error in R_n (with respect to $\sqrt{z}$) is lower than 2^{-382}.

Also, in this case, if $n = 6$ we obtain that the absolute error in R_n (with respect to $\sqrt{z}$) is lower than 2^{-190}, which is lower than 2^{-128}, which is all we need if we work with 128 bits of significant digits.

We include below the Python code for computing the square root.

Listing 6: Square root

```
def sqrt128(a: float128) -> float128:
    assert type(a) == float128
    if a.digits == 0:
        return float128(0,0)
    # We compute a good initial guess.
    l = a.length() - 1 # Note that l = floor(log10(|a|)
    q = (l + a.exp) // 2
    r = (l + a.exp) % 2
    # It can be proved that
    # 1 <= a / 10 ** (2*q) < 100
    # We define c as below to take into account two
    # decimal places of a / 10 ** (2*q)
    c = a.digits * 100 // 10 ** (l - r)
    approximate_value = 10
    for k in range(18):
        # For Vyper or Solidity, we can hardcode
        # a sequence of if ... elif ... elif ...
```

```
        # instead of this 'for' loop.
        if (10 + 5 * k)**2 <= c < (10 + 5 * (k+1))**2:
            approximate_value += 5 * k
            break
    # This is a good initial guess:
    x = float128(approximate_value, q - 1)
    # With this initial guess 8 steps should suffice
    # to get an aproximation whose
    # error is lower than 10**(-38).
    for _ in range(8):
        x_next = ( x + (a / x) ) / 2
        if x_next.digits == x.digits and x_next.exp == x.exp:
            return x
        x = x_next
    return x
```

2.3 Algorithm for the Natural Logarithm

The first step is to divide the input x into two cases : $0 < x \leq 1$ and $1 < x < \infty$. In the second case, we reduce to the first case by inverting. We should note that we have taken precautions during this inversion process which will make more sense once we have described the algorithm for inputs $0 < x < 1$.

In this case, the simple idea is to find a pre-multiplier of the form

$$H = 2^j(1.1)^k(1.014)^l(1.0013)^m$$

such that $H \cdot x$ is still less than, but very close to the number 1. Since the Taylor series expansion for $\ln(1+z)$ is accurate for z small, we can compute $\ln(H \cdot x)$ as $\ln(1 + (H \cdot x - 1))$ and then use logarithm rules to deduce the value of $\ln(x)$. The peculiarities of the chosen factors in the pre-multiplier are a function of unimportant solidity implementation details. Ultimately, the natural logarithms of these factors are hard coded and used to compute $\ln(x)$.

Finally, as mentioned, in some cases insufficient precision can cause errors during the inversion process. Take for example a case in which x is very slightly larger than 1. In this case, inversion can cause the $z = (1/x - 1)$ to be very close to 0, leading to a potential loss of precision in the value of z, which will then propagate when computing the Taylor expansion. As a result, our algorithm requires for $1/x$ to be calculated with twice as many digits of precision before then becoming truncated to 38 significant digits. More generally, note that this is an issue whenever the pre-multiplier times the inverted quantity is very close to 1 (again, the adding precision when inverting solves this issue as well).

We share the corresponding Python code in the Appendix.

3 Analysis of Error Propagation

In this important section we give upper bounds for the relative error when computing arithmetic operations with truncating approximations.

Definition 3. We define the function $\kappa\colon \mathbb{R} \times \mathbb{N} \to \mathbb{Z}$ by

$$\kappa(a, N) = \begin{cases} 0 & \text{if } a = 0, \\ N - 1 - \lfloor \log_{10}(|a|) \rfloor & \text{if } a \neq 0. \end{cases}$$

The next proposition shows that, for $a \neq 0$, $\kappa(a, N)$ is the integer number such that the integer part of $10^{\kappa(a,N)} \cdot |a|$ has exactly N digits.

Proposition 4. *Let $a \in \mathbb{R} - \{0\}$ and let $N \in \mathbb{N}$. Then*

$$10^{N-1} \leq 10^{\kappa(a,N)}|a| < 10^N \ .$$

Proof. Since $a \neq 0$ we have that

$$\kappa(a,N) = N - 1 - \lfloor \log_{10}(|a|) \rfloor \ .$$

Thus,

$$\lfloor \log_{10}(|a|) \rfloor = N - 1 - \kappa(a,N) \ .$$

Hence,

$$N - 1 - \kappa(a,N) \leq \log_{10}(|a|) < N - \kappa(a,N) \ .$$

Then

$$10^{N-1-\kappa(a,N)} \leq |a| < 10^{N-\kappa(a,N)} \ .$$

The result follows. □

Recall that the *sign function* is the function $\mathbf{sgn}\colon \mathbb{R} \to \mathbb{R}$ defined by

$$\mathbf{sgn}(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0, \\ -1 & \text{if } x < 0. \end{cases}$$

Definition 5. Let $N \in \mathbb{N}$. We define the *truncation of order N* as the function $T_N\colon \mathbb{R} \to \mathbb{R}$ defined by

$$T_N(a) = \begin{cases} 0 & \text{if } a = 0, \\ \mathbf{sgn}(a)\lfloor 10^{\kappa(a,N)}|a| \rfloor 10^{-\kappa(a,N)} & \text{if } a \neq 0. \end{cases}$$

Applying Proposition 4, it is not difficult to prove that, for all $a \in \mathbb{R} - \{0\}$, $T_N(a)$ is the real number obtained by truncating a to its first N significant digits.

Proposition 6. *Let $a \in \mathbb{R} - \{0\}$ and let $N \in \mathbb{N}$. Then*

$$\left| \frac{T_N(a)}{a} - 1 \right| \leq 10^{-(N-1)}$$

Proof. Since $a \neq 0$ we have that

$$\begin{aligned} \left| \frac{T_N(a)}{a} - 1 \right| &= \left| \frac{\mathbf{sgn}(a)\lfloor 10^{\kappa(a,N)}|a| \rfloor 10^{-\kappa(a,N)}}{\mathbf{sgn}(a)|a|} - 1 \right| = \left| \frac{\lfloor 10^{\kappa(a,N)}|a| \rfloor}{10^{\kappa(a,N)}|a|} - 1 \right| = \\ &= \frac{1}{10^{\kappa(a,N)}|a|} \left| \lfloor 10^{\kappa(a,N)}|a| \rfloor - 10^{\kappa(a,N)}|a| \right| \leq \frac{1}{10^{\kappa(a,N)}|a|} \leq \\ &\leq 10^{-(N-1)} \ , \end{aligned}$$

by Proposition 4. □

Remark 7.

- Note that Proposition 6 states that, for all $a \in \mathbb{R} - \{0\}$ and $N \in \mathbb{N}$,

$$\mathcal{E}(T_N(a), a) \leq 10^{-(N-1)} \ .$$

- Let $a, \widetilde{a} \in \mathbb{R}$ such that $a \neq 0$. Then

$$\mathcal{E}(\widetilde{a}, a) = \mathcal{E}(-\widetilde{a}, -a) \ .$$

The next Proposition states that if the relative error of $\widetilde{a}$ with respect to a is at most 10^{-N}, then $\widetilde{a}$ differs from a in at most 1 unit of the N-th significant digit.

Proposition 8. *Let $a \in \mathbb{R} - \{0\}$. Let $N \in \mathbb{N}$ and let $\widetilde{a} \in \mathbb{R}$ such that $\mathcal{E}(\widetilde{a}, a) \leq 10^{-N}$. Then*

$$\left|\widetilde{a} \cdot 10^{\kappa(a,N)} - a \cdot 10^{\kappa(a,N)}\right| \leq 1 \ .$$

Proof. Recall that, by Proposition 4,

$$10^{N-1} \leq 10^{\kappa(a,N)}|a| < 10^{N} \ .$$

Thus,

$$\left|\widetilde{a} \cdot 10^{\kappa(a,N)} - a \cdot 10^{\kappa(a,N)}\right| = |a|\, 10^{\kappa(a,N)} \left|\frac{\widetilde{a}}{a} - 1\right| = |a|\, 10^{\kappa(a,N)} \mathcal{E}(\widetilde{a}, a) \leq 10^{-N}\, |a|\, 10^{\kappa(a,N)} \leq 1 \ ,$$

as we wanted to prove. □

Proposition 9. *Let $a, b \in \mathbb{R}_{>0}$ and let $\widetilde{a}, \widetilde{b} \in \mathbb{R}$. Then*

$$\mathcal{E}(\widetilde{a} + \widetilde{b}, a + b) \leq \mathcal{E}(\widetilde{a}, a) + \mathcal{E}(\widetilde{b}, b) \ .$$

Proof. Note that

$$\begin{aligned}\mathcal{E}(\widetilde{a} + \widetilde{b}, a + b) = \left|\frac{\widetilde{a} + \widetilde{b}}{a + b} - 1\right| = \frac{1}{|a + b|} \cdot \left|\widetilde{a} + \widetilde{b} - (a + b)\right| \leq \frac{1}{|a + b|} \cdot \left(|\widetilde{a} - a| + \left|\widetilde{b} - b\right|\right) = \\ = \frac{|\widetilde{a} - a|}{a + b} + \frac{\left|\widetilde{b} - b\right|}{a + b} \leq \frac{|\widetilde{a} - a|}{a} + \frac{\left|\widetilde{b} - b\right|}{b} = \mathcal{E}(\widetilde{a}, a) + \mathcal{E}(\widetilde{b}, b) \ ,\end{aligned}$$

as we wanted to prove. □

Proposition 10. *Let $a, b \in \mathbb{R}_{>0}$ and let $\widetilde{a}, \widetilde{b} \in \mathbb{R}$. Suppose that $a - b \neq 0$. Let δ be defined by $\delta = \min\left\{\left|1 - \frac{a}{b}\right|, \left|1 - \frac{b}{a}\right|\right\}$. Then*

$$\mathcal{E}(\widetilde{a} - \widetilde{b}, a - b) \leq \frac{1}{\delta}(\mathcal{E}(\widetilde{a}, a) + \mathcal{E}(\widetilde{b}, b)) \ .$$

Proof. Note that $\delta > 0$ and that

$$\frac{|a - b|}{a} = \frac{|a - b|}{|a|} = \left|1 - \frac{b}{a}\right| \geq \delta \ ,$$

and

$$\frac{|a - b|}{b} = \frac{|a - b|}{|b|} = \left|1 - \frac{a}{b}\right| \geq \delta \ .$$

Thus,

$$\begin{aligned}\mathcal{E}(\widetilde{a} - \widetilde{b}, a - b) = \left|\frac{\widetilde{a} - \widetilde{b}}{a - b} - 1\right| = \frac{1}{|a - b|} \cdot \left|\widetilde{a} - \widetilde{b} - (a - b)\right| \leq \frac{1}{|a - b|} \cdot \left(|\widetilde{a} - a| + \left|\widetilde{b} - b\right|\right) = \\ = \frac{a}{|a - b|} \cdot \frac{|\widetilde{a} - a|}{a} + \frac{b}{|a - b|} \cdot \frac{\left|\widetilde{b} - b\right|}{b} = \frac{a}{|a - b|} \cdot \left|\frac{\widetilde{a}}{a} - 1\right| + \frac{b}{|a - b|} \cdot \left|\frac{\widetilde{b}}{b} - 1\right| \leq \\ \leq \frac{1}{\delta}(\mathcal{E}(\widetilde{a}, a) + \mathcal{E}(\widetilde{b}, b)) \ ,\end{aligned}$$

as we wanted to prove. □

Remark 11. With the notations of the previous proposition, observe that, if $a < b$, then $0 < 1 - \frac{a}{b} < 1$, and if $a > b$, then $0 < 1 - \frac{b}{a} < 1$. Thus, $\delta < 1$.

Proposition 12. *Let $a, b \in \mathbb{R} - \{0\}$ and let $\widetilde{a}, \widetilde{b} \in \mathbb{R}$. Then*

$$\mathcal{E}(\widetilde{a} \cdot \widetilde{b}, a \cdot b) \leq \mathcal{E}(\widetilde{a}, a) + \mathcal{E}(\widetilde{b}, b) + \mathcal{E}(\widetilde{a}, a)\mathcal{E}(\widetilde{b}, b) \ .$$

Proof. Note that

$$\begin{aligned}
\mathcal{E}(\widetilde{a} \cdot \widetilde{b}, a \cdot b) &= \left| \frac{\widetilde{a} \cdot \widetilde{b}}{a \cdot b} - 1 \right| = \left| \frac{\widetilde{a} \cdot \widetilde{b}}{a \cdot b} - 1 + \frac{\widetilde{a}}{a} - \frac{\widetilde{a}}{a} + \frac{\widetilde{b}}{b} - \frac{\widetilde{b}}{b} + 1 - 1 \right| = \\
&= \left| \frac{\widetilde{a}}{a} - 1 + \frac{\widetilde{b}}{b} - 1 + \frac{\widetilde{a} \cdot \widetilde{b}}{a \cdot b} - \frac{\widetilde{a}}{a} - \frac{\widetilde{b}}{b} + 1 \right| = \\
&= \left| \left(\frac{\widetilde{a}}{a} - 1 \right) + \left(\frac{\widetilde{b}}{b} - 1 \right) + \left(\frac{\widetilde{a}}{a} - 1 \right) \cdot \left(\frac{\widetilde{b}}{b} - 1 \right) \right| \leq \\
&\leq \mathcal{E}(\widetilde{a}, a) + \mathcal{E}(\widetilde{b}, b) + \mathcal{E}(\widetilde{a}, a)\mathcal{E}(\widetilde{b}, b) \ ,
\end{aligned}$$

as we wanted to prove. □

Proposition 13. *Let $a, \widetilde{a} \in \mathbb{R} - \{0\}$. Then*

$$\mathcal{E}\left(\frac{1}{\widetilde{a}}, \frac{1}{a} \right) = \mathcal{E}(a, \widetilde{a}) \ .$$

If, in addition, $\mathcal{E}(\widetilde{a}, a) < 1$, then

$$\mathcal{E}\left(\frac{1}{\widetilde{a}}, \frac{1}{a} \right) = \mathcal{E}(a, \widetilde{a}) \leq \frac{\mathcal{E}(\widetilde{a}, a)}{1 - \mathcal{E}(\widetilde{a}, a)} \ .$$

Proof. Clearly,

$$\mathcal{E}\left(\frac{1}{\widetilde{a}}, \frac{1}{a} \right) = \left| \frac{\frac{1}{\widetilde{a}}}{\frac{1}{a}} - 1 \right| = \left| \frac{a}{\widetilde{a}} - 1 \right| = \mathcal{E}(a, \widetilde{a}) \ .$$

Now, suppose that $\mathcal{E}(\widetilde{a}, a) < 1$. Let

$$r = \frac{\widetilde{a}}{a} - 1 \ .$$

Note that $|r| = \mathcal{E}(\widetilde{a}, a) < 1$. We have that

$$\frac{\widetilde{a}}{a} = 1 + r \ .$$

Thus,

$$\mathcal{E}(a, \widetilde{a}) = \left| \frac{a}{\widetilde{a}} - 1 \right| = \left| \frac{1}{1 + r} - 1 \right| = \left| \frac{-r}{1 + r} \right| = \frac{|r|}{|1 + r|} \leq \frac{|r|}{1 - |r|} \ .$$

The result follows. □

Proposition 14. *Let $a, a_1, a_2 \in \mathbb{R}$ be such that $a \neq 0$ and $a_2 \neq 0$. Then*

$$\mathcal{E}(a_2, a) \leq \mathcal{E}(a_1, a) + \mathcal{E}(a_2, a_1) + \mathcal{E}(a_1, a)\mathcal{E}(a_2, a_1)$$

Proof. Note that

$$\mathcal{E}(a_2, a) = \left|\frac{a_2}{a} - 1\right| = \left|\frac{a_1 \cdot a_2}{a \cdot a_1} - 1\right| = \mathcal{E}(a_1 \cdot a_2, a \cdot a_1) \ .$$

Thus, applying Proposition 12 we obtain that

$$\mathcal{E}(a_2, a) = \mathcal{E}(a_1 \cdot a_2, a \cdot a_1) \leq \mathcal{E}(a_1, a) + \mathcal{E}(a_2, a_1) + \mathcal{E}(a_1, a)\mathcal{E}(a_2, a_1) \ ,$$

as we wanted to prove. □

Definition 15. Let $a, b \in \mathbb{R}$ and let $*$ be a basic arithmetic operation (either addition, subtraction, multiplication, or division). Let $N \in \mathbb{N}$. A *computation step of order* N is a function $\mathcal{S}(a, b)$ defined by

$$\mathcal{S}(a, b) = T_N(a * b) \ .$$

We will also say that $*$ is the *basic arithmetic operation associated to* $\mathcal{S}$.

Proposition 16. *Let $a, b \in \mathbb{R} - \{0\}$ and let $N \in \mathbb{N}$. Let $\mathcal{S}(a, b)$ be a computation step of order N and let $*$ be its associated basic arithmetic operation. Let $a', b' \in \mathbb{R}$. Suppose that the following conditions hold.*

- *Either the operation $*$ is the addition and $ab > 0$, or the operation $*$ is the subtraction and $ab < 0$.*
- *$\mathcal{E}(a', a) \leq \frac{1}{1000}$ and $\mathcal{E}(b', b) \leq \frac{1}{1000}$.*

Then

$$\mathcal{E}(\mathcal{S}(a', b'), a * b) \leq \mathcal{E}(a', a) + \mathcal{E}(b', b) + \tfrac{1002}{1000} \cdot 10^{-(N-1)}$$

Proof. By Proposition 14,

$$\begin{aligned}\mathcal{E}(\mathcal{S}(a', b'), a * b) \leq \mathcal{E}(a' * b', a * b) + \mathcal{E}(\mathcal{S}(a', b'), a' * b') + \\ + \mathcal{E}(a' * b', a * b) \cdot \mathcal{E}(\mathcal{S}(a', b'), a' * b') \ .\end{aligned}$$

Now note that, by the first condition of the statement of the Proposition and by Remark 7, we might assume, without loss of generality, that $a * b$ is an addition between two positive numbers. Hence, from Proposition 9, we obtain that

$$\mathcal{E}(a' * b', a * b) \leq \mathcal{E}(a', a) + \mathcal{E}(b', b) \ .$$

In particular,

$$\mathcal{E}(a' * b', a * b) \leq \tfrac{2}{1000} \ .$$

On the other hand, by Remark 7, we have that

$$\mathcal{E}(\mathcal{S}(a', b'), a' * b') \leq 10^{-(N-1)}$$

Therefore,

$$\begin{aligned}\mathcal{E}(\mathcal{S}(a', b'), a * b) \leq \mathcal{E}(a', a) + \mathcal{E}(b', b) + 10^{-(N-1)} + \tfrac{2}{1000} \cdot 10^{-(N-1)} \leq \\ = \mathcal{E}(a', a) + \mathcal{E}(b', b) + \tfrac{1002}{1000} \cdot 10^{-(N-1)} \ ,\end{aligned}$$

as we wanted to prove. □

Proposition 17. *Let $a, b \in \mathbb{R} - \{0\}$ and let $N \in \mathbb{N}$. Let $\mathcal{S}(a, b)$ be a computation step of order N and let $*$ be its associated basic arithmetic operation. Let $\widetilde{a}, \widetilde{b} \in \mathbb{R}$. Suppose that the following conditions hold.*

- *Either the operation $*$ is the addition and $ab < 0$, or the operation $*$ is the subtraction and $ab > 0$.*
- $\mathcal{E}(a', a) \leq \frac{1}{10}$ *and* $\mathcal{E}(b', b) \leq \frac{1}{10}$.
- $\min\left\{\left|1 - \frac{|a|}{|b|}\right|, \left|1 - \frac{|b|}{|a|}\right|\right\} \geq \frac{1}{2}$.

Then

$$\mathcal{E}(\mathcal{S}(a', b'), a * b) \leq 2 \cdot \big(\mathcal{E}(a', a) + \mathcal{E}(b', b)\big) + \tfrac{14}{10} \cdot 10^{-(N-1)}$$

Proof. As in the proof of the previous Proposition, applying Proposition 14 we obtain that

$$\mathcal{E}(\mathcal{S}(a', b'), a * b) \leq \mathcal{E}(a' * b', a * b) + \mathcal{E}(\mathcal{S}(a', b'), a' * b') + \mathcal{E}(a' * b', a * b) \cdot \mathcal{E}(\mathcal{S}(a', b'), a' * b') \ .$$

Now note that, by the first condition of the statement of the Proposition, we might assume, without loss of generality, that $a * b$ is a subtraction between two positive numbers. Hence, from Proposition 10, we obtain that

$$\mathcal{E}(a' * b', a * b) \leq 2 \cdot \big(\mathcal{E}(a', a) + \mathcal{E}(b', b)\big) \ .$$

In particular,

$$\mathcal{E}(a' * b', a * b) \leq \tfrac{4}{10} \ .$$

On the other hand, by Remark 7, we have that

$$\mathcal{E}(\mathcal{S}(a', b'), a' * b') \leq 10^{-(N-1)}$$

Therefore,

$$\begin{aligned} \mathcal{E}(\mathcal{S}(a', b'), a * b) &\leq 2 \cdot \big(\mathcal{E}(a', a) + \mathcal{E}(b', b)\big) + 10^{-(N-1)} + \tfrac{4}{10} \cdot 10^{-(N-1)} \leq \\ &\leq 2 \cdot \big(\mathcal{E}(a', a) + \mathcal{E}(b', b)\big) + \tfrac{14}{10} \cdot 10^{-(N-1)} \ , \end{aligned}$$

as we wanted to prove. □

Proposition 18. *Let $a, b \in \mathbb{R} - \{0\}$ and let $N \in \mathbb{N}$ such that $N \geq 4$. Let $\mathcal{S}(a, b)$ be a computation step of order N such that its associated basic arithmetic operation is the multiplication. Let $a', b' \in \mathbb{R} - \{0\}$. Let $M = \lfloor \frac{N}{2} \rfloor + 1$. Suppose that $\mathcal{E}(a', a) \leq 10^{-M}$ and $\mathcal{E}(b', b) \leq 10^{-M}$. Then*

$$\mathcal{E}(\mathcal{S}(a', b'), a \cdot b) \leq \mathcal{E}(a', a) + \mathcal{E}(b', b) + \tfrac{111}{100} \cdot 10^{-(N-1)}$$

Proof. As in the previous proofs, applying Proposition 14 we obtain that

$$\begin{aligned} \mathcal{E}(\mathcal{S}(a', b'), a \cdot b) \leq \mathcal{E}(a' \cdot b', a \cdot b) + \mathcal{E}(\mathcal{S}(a', b'), a' \cdot b') + \\ + \mathcal{E}(a' \cdot b', a \cdot b) \cdot \mathcal{E}(\mathcal{S}(a', b'), a' \cdot b') \ . \end{aligned}$$

Note that, from Proposition 12, we obtain that

$$\mathcal{E}(a' \cdot b', a \cdot b) \leq \mathcal{E}(a', a) + \mathcal{E}(b', b) + \mathcal{E}(a', a) \cdot \mathcal{E}(b', b) \ .$$

In particular,

$$\mathcal{E}(a' \cdot b', a \cdot b) \leq 10^{-M} + 10^{-M} + 10^{-2M} \leq \tfrac{2001}{1000} \cdot 10^{-M} \ ,$$

since $M \geq 3$.

Note also that

$$M = \left\lfloor \frac{N}{2} \right\rfloor + 1 > \frac{N}{2} \ ,$$

and thus $N < 2M$.

On the other hand, by Remark 7, we have that

$$\mathcal{E}(\mathcal{S}(a',b'),a'\cdot b')\leq 10^{-(N-1)}$$

Therefore,

$$\begin{aligned}\mathcal{E}(\mathcal{S}(a',b'),a\cdot b)&\leq \mathcal{E}(a',a)+\mathcal{E}(b',b)+\mathcal{E}(a',a)\cdot\mathcal{E}(b',b)+10^{-(N-1)}+\tfrac{2001}{1000}\cdot 10^{-M}\cdot 10^{-(N-1)}\leq\\ &\leq \mathcal{E}(a',a)+\mathcal{E}(b',b)+10^{-2M}+10^{-(N-1)}+\tfrac{2001}{1000000}\cdot 10^{-(N-1)}\leq\\ &\leq \mathcal{E}(a',a)+\mathcal{E}(b',b)+10^{-N}+\tfrac{1002001}{1000000}\cdot 10^{-(N-1)}\leq\\ &\leq \mathcal{E}(a',a)+\mathcal{E}(b',b)+\tfrac{1102001}{1000000}\cdot 10^{-(N-1)}\leq\\ &\leq \mathcal{E}(a',a)+\mathcal{E}(b',b)+\tfrac{111}{100}\cdot 10^{-(N-1)}\ ,\end{aligned}$$

as we wanted to prove. □

Proposition 19. *Let $a,b\in\mathbb{R}-\{0\}$ and let $N\in\mathbb{N}$ such that $N\geq 4$. Let $\mathcal{S}(a,b)$ be a computation step of order N such that its associated basic arithmetic operation is the division. Let $a',b'\in\mathbb{R}-\{0\}$. Let $M=\lfloor\frac{N}{2}\rfloor+1$. Suppose that the following conditions hold.*

- $\mathcal{E}(a',a)\leq 10^{-M}$ *and* $\mathcal{E}(b',b)\leq 10^{-M}$.

Then

$$\mathcal{E}(\mathcal{S}(a',b'),a:b)\leq \mathcal{E}(a',a)+\tfrac{1002}{1000}\cdot\mathcal{E}(b',b)+\tfrac{111}{100}\cdot 10^{-(N-1)}$$

Proof. As in the previous proofs, applying Proposition 14 we obtain that

$$\begin{aligned}\mathcal{E}(\mathcal{S}(a',b'),a:b)\leq{}&\mathcal{E}(a':b',a:b)+\mathcal{E}(\mathcal{S}(a',b'),a':b')+\\ &+\mathcal{E}(a':b',a:b)\cdot\mathcal{E}(\mathcal{S}(a',b'),a':b')\ .\end{aligned}$$

Now, applying Proposition 12, we obtain that

$$\mathcal{E}(a':b',a:b)=\mathcal{E}\left(a'\cdot\frac{1}{b'},a\cdot\frac{1}{b}\right)\leq\mathcal{E}(a',a)+\mathcal{E}\left(\frac{1}{b'},\frac{1}{b}\right)+\mathcal{E}(a',a)\cdot\mathcal{E}\left(\frac{1}{b'},\frac{1}{b}\right)\ .$$

From Proposition 13 we get

$$\mathcal{E}\left(\frac{1}{b'},\frac{1}{b}\right)=\mathcal{E}(b,b')\leq\frac{\mathcal{E}(b',b)}{1-\mathcal{E}(b',b)}\leq\frac{\mathcal{E}(b',b)}{1-10^{-M}}\leq(1+2\cdot 10^{-M})\cdot\mathcal{E}(b',b)\ ,$$

since

$$\frac{1}{1-10^{-M}}\leq 1+2\cdot 10^{-M}$$

because

$$(1+2\cdot 10^{-M})\cdot(1-10^{-M})=1+10^{-M}-2\cdot 10^{-2M}=1+10^{-M}\cdot(1-2\cdot 10^{-M})\geq 1\ .$$

In particular, since $M\geq 3$, we obtain that

$$\mathcal{E}\left(\frac{1}{b'},\frac{1}{b}\right)\leq(1+2\cdot 10^{-M})\cdot\mathcal{E}(b',b)\leq\tfrac{1002}{1000}\cdot\mathcal{E}(b',b)\leq(1+2\cdot 10^{-M})\cdot 10^{-M}\leq\tfrac{1002}{1000}\cdot 10^{-M}\ ,$$

and

$$\begin{aligned}\mathcal{E}(a':b',a:b)&\leq 10^{-M}+\tfrac{1002}{1000}\cdot 10^{-M}+\tfrac{1002}{1000}\cdot 10^{-2M}\leq\\ &\leq 10^{-M}+\tfrac{1002}{1000}\cdot 10^{-M}+\tfrac{1002}{1000000}\cdot 10^{-M}=\tfrac{2003002}{1000000}\cdot 10^{-M}\ .\end{aligned}$$

Note also that

$$M=\left\lfloor\frac{N}{2}\right\rfloor+1>\frac{N}{2}\ ,$$

and thus $N < 2M$.

On the other hand, by Remark 7, we have that

$$\mathcal{E}(\mathcal{S}(a', b'), a' : b') \leq 10^{-(N-1)}$$

Therefore,

$$\begin{aligned}
\mathcal{E}(\mathcal{S}(a', b'), a : b) &\leq \mathcal{E}(a', a) + \mathcal{E}\left(\frac{1}{b'}, \frac{1}{b}\right) + \mathcal{E}(a', a) \cdot \mathcal{E}\left(\frac{1}{b'}, \frac{1}{b}\right) + 10^{-(N-1)} + \\
&+ \tfrac{2003002}{1000000} \cdot 10^{-M} \cdot 10^{-(N-1)} \leq \\
&\leq \mathcal{E}(a', a) + \tfrac{1002}{1000} \cdot \mathcal{E}(b', b) + \tfrac{1002}{1000} \cdot 10^{-2M} + 10^{-(N-1)} + \\
&+ \tfrac{2003002}{1000000} \cdot 10^{-M} \cdot 10^{-(N-1)} \leq \\
&\leq \mathcal{E}(a', a) + \tfrac{1002}{1000} \cdot \mathcal{E}(b', b) + \tfrac{1002}{1000} \cdot 10^{-N} + 10^{-(N-1)} + \tfrac{2003002}{10^9} \cdot 10^{-(N-1)} \leq \\
&\leq \mathcal{E}(a', a) + \tfrac{1002}{1000} \cdot \mathcal{E}(b', b) + \tfrac{1102203002}{10^9} \cdot 10^{-(N-1)} \leq \\
&\leq \mathcal{E}(a', a) + \tfrac{1002}{1000} \cdot \mathcal{E}(b', b) + \tfrac{111}{100} \cdot 10^{-(N-1)} \ ,
\end{aligned}$$

as we wanted to prove. □

Theorem 20. *Let $N \in \mathbb{N}$ such that $N > 12$. Let $L \in \mathbb{N}$. Consider a tree of L computation steps of order N such that the associated basic arithmetic operation of each step is either the multiplication or the addition. Suppose that $L \leq 1000$. Then, the relative error after the tree of computation steps is performed is lower than*

$$\tfrac{111}{100} \cdot L \cdot 10^{-(N-1)} \ .$$

Proof. We proceed by induction on L. It is not difficult to verify that the result for $L = 1$ follows from either Proposition 16 and Proposition 18, according to which the first operation is.

Now, suppose that k computation steps have been performed (with $k \leq 999$) and that we want to perform another one. Let $\mathcal{S}$ be this next computation step, let $*$ be its associated basic arithmetic operation. Let a and b be the exact results that were obtained from the sequence of previous operations which are involved in this next operation. This means that the next basic arithmetic operation we need to perform is $a * b$. Let a' and b' be the approximate results that were obtained from the sequence of previous operations, such that a' is an approximation for a and b' is an approximation for b.

Let α be the amount of steps used to obtain a', and let β be the amount of steps used to obtain b'. Note that $\alpha + \beta \leq k$.

By the inductive hypothesis, we have that

$$\mathcal{E}(a', a) \leq \tfrac{111}{100} \cdot \alpha \cdot 10^{-(N-1)}$$

and

$$\mathcal{E}(b', b) \leq \tfrac{111}{100} \cdot \beta \cdot 10^{-(N-1)} \ .$$

Let $M = \lfloor \frac{N}{2} \rfloor + 1$. Note that $M \leq \frac{N}{2} + 1$

$$M \leq \frac{N}{2} + 1 < N - 5$$

since $N > 12$.

Note that, since $\alpha \leq k \leq 999$, we have that

$$\mathcal{E}(a', a) \leq \tfrac{111}{100} \cdot \alpha \cdot 10^{-(N-1)} \leq \tfrac{111}{100} \cdot 10^3 \cdot 10^{-(N-1)} < 10^4 \cdot 10^{-(N-1)} = 10^{-(N-5)} < 10^{-M}.$$

Similarly, $\mathcal{E}(b', b) \leq 10^{-M}$.

Therefore, applying either Proposition 16 or Proposition 18 according to which the operation $*$ is, we obtain that

$$\begin{aligned}\mathcal{E}(\mathcal{S}(a',b'),a*b) &\leq \mathcal{E}(a',a)+\mathcal{E}(b',b)+\tfrac{111}{100}\cdot 10^{-(N-1)} \leq \\ &\leq \tfrac{111}{100}\cdot\alpha\cdot 10^{-(N-1)}+\tfrac{111}{100}\cdot\beta\cdot 10^{-(N-1)}+\tfrac{111}{100}\cdot 10^{-(N-1)} \leq \\ &\leq \tfrac{111}{100}\cdot(k+1)\cdot 10^{-(N-1)}\end{aligned}$$

This completes the proof. □

Theorem 21. *Let $N\in\mathbb{N}$ such that $N>12$. Let $L\in\mathbb{N}$. Consider a tree of L computation steps of order N such that the associated basic arithmetic operation of each step is either the multiplication or the addition or the division. Suppose that $L\leq 1000$. Then, the relative error after the tree of computation steps is performed is lower than*

$$\frac{\left(\frac{1002}{1000}\right)^L-1}{\frac{1002}{1000}-1}\cdot\tfrac{111}{100}\cdot 10^{-(N-1)}\ .$$

Proof. As in the previous proof, we proceed by induction on L. It is not difficult to verify that the result for $L=1$ follows from either Proposition 16 or Proposition 18 or Proposition 19, according to which the first operation is.

Now, suppose that k computation steps have been performed (with $k\leq 999$) and that we want to perform another one. Let $\mathcal{S}$ be this next computation step, let $*$ be its associated basic arithmetic operation. Let a and b be the exact results that were obtained from the sequence of previous operations which are involved in this next operation. This means that the next basic arithmetic operation we need to perform is $a*b$. Let a' and b' be the approximate results that were obtained from the sequence of previous operations, such that a' is an approximation for a and b' is an approximation for b.

Let α be the amount of steps used to obtain a', and let β be the amount of steps used to obtain b'. Note that $\alpha+\beta\leq k$. Let $\sigma\colon\mathbb{N}_0\to\mathbb{R}$ be defined by

$$\sigma(n)=\frac{\left(\frac{1002}{1000}\right)^n-1}{\frac{1002}{1000}-1}\ .$$

Note that

$$\sigma(n)=\sum_{j=0}^{n-1}\left(\tfrac{1002}{1000}\right)^j\ .$$

By the inductive hypothesis, we have that

$$\mathcal{E}(a',a)\leq\frac{\left(\frac{1002}{1000}\right)^\alpha-1}{\frac{1002}{1000}-1}\cdot\tfrac{111}{100}\cdot 10^{-(N-1)}=\tfrac{111}{100}\cdot\sigma(\alpha)\cdot 10^{-(N-1)}$$

and

$$\mathcal{E}(b',b)\leq\frac{\left(\frac{1002}{1000}\right)^\beta-1}{\frac{1002}{1000}-1}\cdot\tfrac{111}{100}\cdot 10^{-(N-1)}=\tfrac{111}{100}\cdot\sigma(\beta)\cdot 10^{-(N-1)}\ .$$

Let $M=\lfloor\frac{N}{2}\rfloor+1$. Note that $M\leq\frac{N}{2}+1$

$$M\leq\frac{N}{2}+1<N-5$$

since $N>12$.

Note also that, since $\alpha\leq k\leq 999$, we have that

$$\mathcal{E}(a',a)\leq\tfrac{111}{100}\cdot\sigma(\alpha)\cdot 10^{-(N-1)}<\tfrac{111}{100}\cdot 3600\cdot 10^{-(N-1)}<10^4\cdot 10^{-(N-1)}=10^{-(N-5)}<10^{-M}.$$

Similarly, $\mathcal{E}(b', b) \leq 10^{-M}$.

Therefore, applying either Proposition 16 or Proposition 18 or Proposition 19 according to which the operation $*$ is, we obtain that

$$\begin{aligned}
\mathcal{E}(\mathcal{S}(a', b'), a * b) &\leq \mathcal{E}(a', a) + \tfrac{1002}{1000} \cdot \mathcal{E}(b', b) + \tfrac{111}{100} \cdot 10^{-(N-1)} \leq \\
&\leq \tfrac{111}{100} \cdot \sigma(\alpha) \cdot 10^{-(N-1)} + \tfrac{1002}{1000} \cdot \tfrac{111}{100} \cdot \sigma(\beta) \cdot 10^{-(N-1)} + \tfrac{111}{100} \cdot 10^{-(N-1)} = \\
&= \tfrac{111}{100} \cdot 10^{-(N-1)} \cdot \left(\sigma(\alpha) + \tfrac{1002}{1000} \cdot \sigma(\beta) + 1\right) \leq \\
&= \tfrac{111}{100} \cdot 10^{-(N-1)} \cdot \left(\left(\tfrac{1002}{1000}\right)^{\beta+1} \sigma(\alpha) + \tfrac{1002}{1000} \cdot \sigma(\beta) + 1\right) = \\
&= \tfrac{111}{100} \cdot 10^{-(N-1)} \cdot \sigma(\alpha + \beta + 1) \leq \\
&\leq \tfrac{111}{100} \cdot \sigma(k+1) \cdot 10^{-(N-1)} \ .
\end{aligned}$$

This completes the proof. □

Remark 22. The previous theorem implies that in a tree of at most 100 computation steps of order $N > 12$, under suitable mild assumptions, the relative error is lower than $2 \cdot 10^{-(N-3)}$. Thus, by Proposition 8, in the great majority of cases we will get $N-5$ correct significant digits.

This means that, if we are working with 38 significant digits, we can expect to obtain 33 correct digits after performing such operations. It is important to observe that 33 correct digits are more than enough for almost all practical purposes in DeFi.

Error bound for the implementation of the square root algorithm in `float128`

In this section we will give a bound for the relative error related to the computation of the square root with the proposed algorithm for the `float128` library.

Let $N = 38$. Let $g\colon \mathbb{R} \to \mathbb{R}$ be defined by

$$g(x) = \frac{1}{2}\left(x + \frac{z}{x}\right)$$

and let $\widetilde{g}\colon \mathbb{R} \to \mathbb{R}$ be defined by

$$\widetilde{g}(x) = \frac{1}{2}\left(x + \frac{z}{x}\right)$$

where the operations in the formula of $\widetilde{g}(x)$ are computed using computation steps of order N.

With the notations of the previous subsection, let $R'_0 = R_0$, and, for each $n \in \mathbb{N}$ let $R'_n = \widetilde{g}(R'_{n-1})$. It is not difficult to verify that, for each $n \in \mathbb{N}$, the number of step operations needed to compute R'_n is $3 \cdot (2^n - 1)$.

Since for all $n \leq 8$, $3 \cdot (2^n - 1) \leq 3 \cdot (2^8 - 1) = 765 \leq 1000$, applying Theorem 21 we obtain that, for all $n \leq 8$,

$$\mathcal{E}(R'_n, R_n) \leq \frac{\left(\frac{1002}{1000}\right)^{765} - 1}{\frac{1002}{1000} - 1} \cdot \tfrac{111}{100} \cdot 10^{-(38-1)} \leq 2005 \cdot 10^{-37} = 2.005 \cdot 10^{-34} \ .$$

Note also that

$$\mathcal{E}(R_6, \sqrt{z}) \leq |\varepsilon_6| \leq 2^{-(2^{6+1}-1)} = 2^{-127} \leq 10^{-38} \ .$$

Hence, applying proposition 14 we obtain that

$$\begin{aligned}
\mathcal{E}(R'_6, \sqrt{z}) &\leq \mathcal{E}(R'_6, R_6) + \mathcal{E}(R_6, \sqrt{z}) + \mathcal{E}(R'_6, R_6) \cdot \mathcal{E}(R_6, \sqrt{z}) \leq \\
&\leq 2.005 \cdot 10^{-34} + 10^{-38} + 2.005 \cdot 10^{-34} \cdot 10^{-38} \leq \\
&\leq 3 \cdot 10^{-34} \ .
\end{aligned}$$

Thus,

$$\left|R'_6 - \sqrt{z}\right| = \sqrt{z} \cdot \mathcal{E}(R'_6, \sqrt{z}) \leq 10 \cdot \mathcal{E}(R'_6, \sqrt{z}) \leq 3 \cdot 10^{-33} \ .$$

From the arguments given in the previous subsection it follows that

$$\left|g(R'_6) - \sqrt{z}\right| \leq \frac{1}{2} \cdot \left|R'_6 - \sqrt{z}\right|^2 \leq \tfrac{9}{2} \cdot 10^{-66} \ .$$

Thus,

$$\mathcal{E}(g(R'_6), \sqrt{z}) = \frac{|g(R'_6) - \sqrt{z}|}{\sqrt{z}} \leq \left|g(R'_6) - \sqrt{z}\right| \leq \tfrac{9}{2} \cdot 10^{-66} \ .$$

Since $\widetilde{g}(R'_6)$ is obtained from R'_6 by performing 3 computation steps, applying theorem 21 we obtain that

$$\mathcal{E}(\widetilde{g}(R'_6), g(R'_6)) \leq \frac{\left(\frac{1002}{1000}\right)^3 - 1}{\frac{1002}{1000} - 1} \cdot \tfrac{111}{100} \cdot 10^{-(38-1)} \leq 3.3367 \cdot 10^{-37} \ .$$

Therefore, applying proposition 14 we obtain that

$$\begin{aligned}
\mathcal{E}(\widetilde{g}(R'_6), \sqrt{z}) &\leq \mathcal{E}(\widetilde{g}(R'_6), g(R'_6)) + \mathcal{E}(g(R'_6), \sqrt{z}) + \mathcal{E}(\widetilde{g}(R'_6), g(R'_6)) \cdot \mathcal{E}(g(R'_6), \sqrt{z}) \leq \\
&\leq 3.3367 \cdot 10^{-37} + \tfrac{9}{2} \cdot 10^{-66} + 3.3367 \cdot 10^{-37} \cdot \tfrac{9}{2} \cdot 10^{-66} \leq \\
&\leq 3.337 \cdot 10^{-37} \ .
\end{aligned}$$

This means that, using 7 steps in the algorithm for computing the square root, we can guarantee that the relative error will be no more than $3.337 \cdot 10^{-37}$.

Error bound for the implementation of the algorithm to compute $\ln$ in `float128`

In this subsection we will give an upper bound for the relative error of the algorithm to compute the natural logarithm that we proposed previously.

We will analyze first the case in which the argument of the natural logarithm is a number that is not greater than 1.

Let $x_0 \in (0, 1]$. Clearly, there exists $m_0 \in \mathbb{N}_0$ such that $10^{m_0} x_0 \in (0.1, 1]$. Let $x_1 = 10^{m_0} x_0$. Observe that, in the `float128` class, this can be achieved by just modifying the exponent of the given `float128` instance.

It is easy to prove that there exists $q_1 \in \{0, 1, 2, 3\}$ such that $2^{q_1} x_1 \in (0.5, 1]$. Let $x_2 = 2^{q_1} x_1$. In a similar way, we can prove that there exist $q_2, q_3, q_4 \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ such that $x_3 = (1.1)^{q_2} x_2 \in (0.9, 1]$, $x_4 = (1.014)^{q_3} x_3 \in (0.986, 1]$ and $x_5 = (1.0013)^{q_4} x_4 \in (0.9949, 1]$.

In this way, we have found a multiplier $H = 10^{m_0} 2^{q_1} (1.1)^{q_2} (1.014)^{q_3} (1.0013)^{q_4}$ such that $x_5 = H x_0 \in (0.9949, 1]$. The idea is that

$$\begin{aligned}
\ln(x_0) &= \ln(x_5) - \ln(H) = \\
&= \ln(x_5) - m_0 \ln(10) - q_1 \ln(2) - q_2 \ln(1.1) - q_3 \ln(1.014) - q_4 \ln(1.0013) \ .
\end{aligned}$$

Hence, once we compute $\ln(x_5)$ we will be able to obtain $\ln(x_0)$ since the values of $\ln(10)$, $\ln(2)$, $\ln(1.1)$, $\ln(1.014)$, and $\ln(1.0013)$ will be precomputed with sufficiently high precision. In addition, since x_5 is close to 1 and we will use the Taylor polynomial of the ln centered at 1, we will be able to obtain very good precision with a few terms. It is important to mention, though, that the factors of the multiplier H that we chose are arbitrary. Our choice was made thinking about ease of implementation in Solidity and gas-efficiency.

Observe also that the value of x_5 will be computed from the value of x_1 doing four successive multiplications by the factors 2^{q_1}, $(1.1)^{q_2}$, $(1.014)^{q_3}$ and $(1.0013)^{q_4}$, which will be also precomputed. And since the value of the input x_0 int he Solidity implementation will be given with 72 digits of precision, applying Proposition 18, it follows that the relative error in the computed value $\widehat{x}_5$ of x_5 with respect to its real value will be bounded by $4 \cdot \frac{111}{100} \cdot 10^{-71}$.

Now, we consider the Taylor series of the natural logarithm, which can be written as follows

$$\ln(1+z) = \sum_{n=1}^{\infty} (-1)^{n-1} \frac{z^n}{n}, \quad |z| < 1$$

Making the substitution $z \mapsto -z$ gives

$$\ln(1-z) = -\sum_{n=1}^{\infty} \frac{z^n}{n}, \quad |z| < 1$$

Once we have the values of x_5 and $\widehat{x}_5$ we compute $z = 1 - x_5$

Appendix: Proof of Quadratic Convergence for Newton's Iterative Method

The contents of this appendix are taken verbatim from this Wikipedia article and included here for completeness.

According to Taylor's theorem, any function $f(x)$ which has a continuous second derivative can be represented by an expansion about a point that is close to a root of $f(x)$. Suppose this root is α. Then the expansion of $f(\alpha)$ about x_n is:

$$f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + R_1$$

where the Lagrange form of the Taylor series expansion remainder is

$$R_1 = \frac{1}{2!} f''(\xi_n)(\alpha - x_n)^2$$

where ξ is in between x_n and α. Since α is the root, equation (1) becomes:

$$0 = f(\alpha) = f(x_n) + f'(x_n)(\alpha - x_n) + \frac{1}{2} f''(\xi_n)(\alpha - x_n)^2$$

Dividing equation (2) by $f'(x_n)$ and rearranging gives

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

Remembering that x_{n+1} is defined by

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

one finds that

$$\underbrace{\alpha - x_{n+1}}_{\varepsilon_{n+1}} = \frac{-f''(\xi_n)}{2f'(x_n)} \cdot \underbrace{(\alpha - x_n)}_{\varepsilon_n}^2$$

That is,

$$\varepsilon_{n+1} = \frac{-f''(\xi_n)}{2f'(x_n)} \cdot \varepsilon_n^2$$

Taking the absolute value of both sides gives

$$|\varepsilon_{n+1}| = \frac{|f''(\xi_n)|}{2\,|f'(x_n)|} \cdot |\varepsilon_n|^2$$

Equation (6) shows that the order of convergence is at least quadratic if the following conditions are satisfied:

- $f'(x) \neq 0;$ for all $x \in I,$ where $I = [\alpha - |\varepsilon_0|, \alpha + |\varepsilon_0|]$
- $f''(x)$ is continuous, for all $x \in I$;
- $M\,|\varepsilon_0| < 1,$

where

$$M = \frac{1}{2}\left(\sup_{x \in I} \left|f''(x)\right|\right)\left(\sup_{x \in I} \frac{1}{|f'(x)|}\right).$$

If these conditions hold,

$$|\varepsilon_{n+1}| \leq M\,|\varepsilon_n|^2.$$

Appendix B: Python code for the implementation of the natural logarithm in `float128`

Listing 7: Natural logarithm

```
def ln128_prec(digits: int, exp: int) -> float128:
    """
    We compute the natural logarithm of digits * 10**exp
    """
    assert digits > 0
    assert digits < 10**76
    len_digits = number_of_digits(digits)

    if exp <= 0 and digits == 10 ** -exp:
        # This is the case in which the argument of the logarithm
            is 1.
        return float128(0,0)
    elif len_digits > -exp:
        # This is the case in which the argument of the logarithm
            is greater than 1.
        # We truncate to (exactly) 38 digits to make the division
            1/argument easier.
        if len_digits > 38:
            extra_digits = len_digits - 38
            digits = digits // 10**extra_digits
            exp = exp + extra_digits
        elif len_digits < 38:
            extra_digits = 38 - len_digits
            digits = digits * 10**extra_digits
            exp = exp - extra_digits
        assert 10**37 <= digits and digits < 10**38
        # We compute 1/argument with 76 digits of precision.
        q1 = 10**76 // digits
        r1 = 10**76 % digits
        q2 = (10**38 * r1) // digits
        one_over_argument_in_long_int = q1 * 10**38 + q2
        m10 = number_of_digits(one_over_argument_in_long_int)
        assert 76 <= m10 and m10 <= 77 # This assertion should
           hold. Included for testing purposes.
        # We truncate one_over_argument_in_long_int to 76 digits.
        one_over_argument_76 = one_over_argument_in_long_int
        m76 = m10
        if m76 > 76:
            extra_digits = m76 - 76
            m76 = m76 - extra_digits
            one_over_argument_76 = one_over_argument_in_long_int
               // 10**extra_digits
        exp_one_over_argument = - 38 - 76 - exp
        assert exp_one_over_argument + m10 <= 0 # Shouldn't be
           needed. It has been included for testing purposes.
        return - ln128_prec(one_over_argument_76, -m76) - (
           exp_one_over_argument + m10) * ln10
```

```
if len_digits <= -exp:
    # This is the case in which the argument of the logarithm
       is smaller than 1.
    # We first adjust the exponent so that the argument is
       between 0.1 and 1.
    m10 = len_digits + exp
    exp = exp - m10
    assert digits < 10**76

    # We add 0's to digits to ensure the argument has
    # 76 digits and the exponent is -76.
    # This is done to simplify the process later.
    len_digits = number_of_digits(digits)
    m2 = 76 - len_digits
    digits = digits * 10**m2
    exp = exp - m2
    # Some assertions for testing purposes.
    assert number_of_digits(digits) == 76
    assert exp == -76

    # We find the suitable value of k and the multiplier 2**k
    # so that 0.5 <= 2**k * x <= 1
    if digits > 25 * 10**74:
        if digits > 50 * 10**74:
            k = 0
            multiplier_k = 1
        else: # if digits <= 50 * 10**74:
            k = 1
            multiplier_k = 2
    else: # digits < 25 * 10**74
        if digits > 125 * 10**73:
            k = 2
            multiplier_k = 4
        else: # if digits <= 125 * 10**73:
            k = 3
            multiplier_k = 8

    digits = multiplier_k * digits
    assert digits > 5 * 10**75
    assert digits <= 10**76

    # We find the suitable value of q1 and the multiplier
       (1.1)**q1
    # so that 0.9 <= (1.1)**q1 * updated_x <= 1
    # We use the following intervals:
    # (index -> lower bound of the interval)
    # 0 ->  50000000 * 10**68
    # 1 ->  51200000 * 10**68
    # 2 ->  56400000 * 10**68
    # 3 ->  62000000 * 10**68
    # 4 ->  68300000 * 10**68
```

```
# 5 ->  75000000 * 10**68
# 6 ->  82000000 * 10**68
# 7 ->  90000000 * 10**68

if digits > 68300000 * 10**68:
    if digits > 82000000 * 10**68:
        if digits > 90000000 * 10**68:
            q1 = 0
            # multiplier_q1 = 1
        else: # digits <= 90000000 * 10**68
            q1 = 1
            # multiplier_q1 = 1.1
            digits = digits + digits // 10
    else: # digits <= 82000000 * 10**68
        if digits > 75000000 * 10**68:
            q1 = 2
            # multiplier_q1 = 1.1 ** 2 # = 1.21
            digits = digits + 2 * digits // 10 + digits
               // 100
        else: # digits <= 75000000 * 10**68
            q1 = 3
            # multiplier_q1 = 1.1 ** 3 # = 1.331
            digits = digits + 3 * digits // 10 + 3 *
               digits // 100 + digits // 1000
else: # digits <= 68300000 * 10**68
    if digits > 56400000 * 10**68:
        if digits > 62000000 * 10**68:
            q1 = 4
            # multiplier_q1 = 1.1 ** 4 # 1.4641
            digits = digits + 4 * digits // 10 + 6 *
               digits // 100 + 4 * digits // 1000 +
               digits // 10000
        else: # digits <= 62000000 * 10**68
            q1 = 5
            # multiplier_q1 = 1.1 ** 5 # = 1.61051
            digits = digits + 6 * digits // 10 + 1 *
               digits // 100 + 0 * digits // 1000 + 5 *
               digits // 10000 + 1 * digits // 100000
    else: # digits <= 56400000 * 10**68
        if digits > 51200000 * 10**68:
            q1 = 6
            # multiplier_q1 = 1.1 ** 6 # = 1.771561
            digits = digits + 7 * digits // 10 + 7 *
               digits // 100 + 1 * digits // 1000 + 5 *
               digits // 10000 + 6 * digits // 100000 + 1
                * digits // 1000000
        else: # digits <= 51200000 * 10**68
            q1 = 7
            # multiplier_q1 = 1.1 ** 7 # = 1.9487171
            digits = digits + 9 * digits // 10 + 4 *
               digits // 100 + 8 * digits // 1000 + 7 *
               digits // 10000 + 1 * digits // 100000 + 7
```

```
                * digits // 1000000 + 1 * digits //
               10000000
# Now digits has already been updated
assert digits > 9 * 10**75
assert digits <= 10**76

# We find the suitable value of q2 and the multiplier
   (1.014)**q2
# so that 0.986 <= (1.014)**q2 * updated_x <= 1
# We use the following intervals:
# (index -> lower bound of the interval)
# 0 ->  9 * 10**75
# 1 ->  9072 * 10**72
# 2 ->  9199 * 10**72
# 3 ->  9328 * 10**72
# 4 ->  9459 * 10**72
# 5 ->  9591 * 10**72
# 6 ->  9725 * 10**72
# 7 ->  9860 * 10**72
# partition_1014 = [0.9, 0.9072, 0.9199, 0.9328, 0.9459,
   0.9591, 0.9725, 0.986, 1]

if digits > 9459 * 10**72:
    if digits > 9725 * 10**72:
        if digits > 9860 * 10**72:
            q2 = 0
            # multiplier_q2 = 1
        else: # digits <= 9860 * 10**72
            q2 = 1
            # multiplier_q2 = 1.014
            digits = digits + 0 * digits // 10 + 1 *
               digits // 100 + 4 * digits //1000
    else: # digits <= 9725 * 10**72
        if digits > 9591 * 10**72:
            q2 = 2
            # multiplier_q2 = 1.028196
            digits = digits + 0 * digits // 10 + 2 *
               digits // 100 + 8 * digits //1000 \
                           + 1 * digits // 10000 + 9 *
                              digits // 100000 + 6 *
                              digits //1000000
        else: # digits <= 9591 * 10**72
            q2 = 3
            # multiplier_q2 = 1.042590744
            digits = digits + 0 * digits // 10 + 4 *
               digits // 100 + 2 * digits //1000 \
                           + 5 * digits // 10000 + 9 *
                              digits // 100000 + 0 *
                              digits //1000000 \
                           + 7 * digits // 10000000 + 4
                              * digits // 100000000 + 4
                              * digits //1000000000
```

```
    else: # digits <= 9459 * 10**72
        if digits > 9199 * 10**72:
            if digits > 9328 * 10**72:
                q2 = 4
                # multiplier_q2 = 1.057187014416
                digits = digits + 0 * digits // 10 + 5 *
                   digits // 100 + 7 * digits //1000 \
                                + 1 * digits // 10000 + 8 *
                                   digits // 100000 + 7 *
                                   digits //1000000 \
                                + 0 * digits // 10000000 + 1
                                   * digits // 100000000 + 4
                                   * digits //1000000000 \
                                + 4 * digits // 10000000000 +
                                    1 * digits //
                                   100000000000 + 6 * digits
                                   //1000000000000
            else: # digits <= 9328 * 10**72
                q2 = 5
                # multiplier_q2 = 1.071987632617824
                digits = digits + 0 * digits // 10 + 7 *
                   digits // 100 + 1 * digits //1000 \
                                + 9 * digits // 10000 + 8 *
                                   digits // 100000 + 7 *
                                   digits //1000000 \
                                + 6 * digits // 10000000 + 3
                                   * digits // 100000000 + 2
                                   * digits //1000000000 \
                                + 6 * digits // 10000000000 +
                                    1 * digits //
                                   100000000000 + 7 * digits
                                   //1000000000000 \
                                + 8 * digits //
                                   10000000000000 + 2 *
                                   digits // 100000000000000
                                   + 4 * digits
                                   //1000000000000000
        else: # digits <= 9199 * 10**72
            if digits > 9072 * 10**72:
                q2 = 6
                # multiplier_q2 = 1.086995459474473536
                digits = digits + 0 * digits // 10 + 8 *
                   digits // 100 + 6 * digits //1000 \
                                + 9 * digits // 10000 + 9 *
                                   digits // 100000 + 5 *
                                   digits //1000000 \
                                + 4 * digits // 10000000 + 5
                                   * digits // 100000000 + 9
                                   * digits //1000000000 \
                                + 4 * digits // 10000000000 +
                                    7 * digits //
                                   100000000000 + 4 * digits
```

```
                                    //1000000000000 \
                                + 4 * digits //
                                    10000000000000 + 7 *
                                    digits // 100000000000000
                                    + 3 * digits
                                    //1000000000000000 \
                                + 5 * digits //
                                    10000000000000000 + 3 *
                                    digits //
                                    100000000000000000 + 6 *
                                    digits
                                    //1000000000000000000
            else: # digits <= 9072 * 10**72
                q2 = 7
                # multiplier_q2 = 1.102213395907116165504
                digits = digits + 1 * digits // 10 + 0 *
                   digits // 100 + 2 * digits //1000 \
                                + 2 * digits // 10000 + 1 *
                                    digits // 100000 + 3 *
                                    digits //1000000 \
                                + 3 * digits // 10000000 + 9
                                    * digits // 100000000 + 5
                                    * digits //1000000000 \
                                + 9 * digits // 10000000000 +
                                     0 * digits //
                                    100000000000 + 7 * digits
                                    //1000000000000 \
                                + 1 * digits //
                                    10000000000000 + 1 *
                                    digits // 100000000000000
                                    + 6 * digits
                                    //1000000000000000 \
                                + 1 * digits //
                                    10000000000000000 + 6 *
                                    digits //
                                    100000000000000000 + 5 *
                                    digits
                                    //1000000000000000000 \
                                + 5 * digits //
                                    10000000000000000000 + 0 *
                                     digits //
                                    100000000000000000000 + 4
                                    * digits
                                    //1000000000000000000000
# Now digits has already been updated
assert digits >= 9860 * 10**72
assert digits <= 10**76

# We find the suitable value of q3 and the multiplier
   (1.0013)**q3
# so that 0.9949 <= (1.0013)**q3 * updated_x <= 1
# We use the following intervals:
```

```
# (index -> lower bound of the interval)
# 0 ->  986 * 10**73
# 1 ->  987274190490 * 10**64
# 2 ->  988557646937 * 10**64
# 3 ->  989842771878 * 10**64
# 4 ->  991129567482 * 10**64
# 5 ->  992418035920 * 10**64
# 6 ->  993708179366 * 10**64
# 7 ->  995 * 10**73
# partition_10013 = [0.986, 0.987274190490,
   0.988557646937, 0.989842771878, 0.991129567482,
   0.992418035920, 0.993708179366, 0.995, 1]

if digits > 991129567482 * 10**64:
    if digits > 993708179366 * 10**64:
        if digits > 995 * 10**73:
            q3 = 0
            # multiplier_q3 = 1
        else: # digits <= 995 * 10**73
            q3 = 1
            # multiplier_q3 = 1.0013
            digits = digits + 0 * digits // 10 + 0 *
               digits // 100 + 1 * digits //1000 + 3 *
               digits // 10000
    else: # digits <= 993708179366 * 10**64
        if digits > 992418035920 * 10**64:
            q3 = 2
            # multiplier_q3 = 1.00260169
            digits = digits + 0 * digits // 10 + 0 *
               digits // 100 + 2 * digits //1000 + 6 *
               digits // 10000 \
                           + 0 * digits // 100000 + 1 *
                              digits // 1000000 + 6 *
                              digits //10000000 + 9 *
                              digits // 100000000
        else: # digits <= 992418035920 * 10**64
            q3 = 3
            # multiplier_q3 = 1.003905072197
            digits = digits + 0 * digits // 10 + 0 *
               digits // 100 + 3 * digits //1000 + 9 *
               digits // 10000 \
                           + 0 * digits // 100000 + 5 *
                              digits // 1000000 + 0 *
                              digits //10000000 + 7 *
                              digits // 100000000 \
                           + 2 * digits // 1000000000 +
                              1 * digits // 10000000000
                              + 9 * digits
                              //100000000000 + 7 *
                              digits // 1000000000000
else: # digits <= 991129567482 * 10**64
    if digits > 988557646937 * 10**64:
```

```
if digits > 989842771878 * 10**64:
    q3 = 4
    # multiplier_q3 = 1.0052101487908561
    digits = digits + 0 * digits // 10 + 0 *
       digits // 100 + 5 * digits //1000 + 2 *
       digits // 10000 \
                    + 1 * digits // 100000 + 0 *
                       digits // 1000000 + 1 *
                       digits //10000000 + 4 *
                       digits // 100000000 \
                    + 8 * digits // 1000000000 +
                       7 * digits // 10000000000
                       + 9 * digits
                       //100000000000 + 0 *
                       digits // 1000000000000 \
                    + 8 * digits //
                       10000000000000 + 5 *
                       digits // 100000000000000
                       + 6 * digits
                       //1000000000000000 + 1 *
                       digits //
                       10000000000000000
else: # digits <= 989842771878 * 10**64
    q3 = 5
    # multiplier_q3 = 1.00651692198428421293
    digits = digits + 0 * digits // 10 + 0 *
       digits // 100 + 6 * digits //1000 + 5 *
       digits // 10000 \
                    + 1 * digits // 100000 + 6 *
                       digits // 1000000 + 9 *
                       digits //10000000 + 2 *
                       digits // 100000000 \
                    + 1 * digits // 1000000000 +
                       9 * digits // 10000000000
                       + 8 * digits
                       //100000000000 + 4 *
                       digits // 1000000000000 \
                    + 2 * digits //
                       10000000000000 + 8 *
                       digits // 100000000000000
                       + 4 * digits
                       //1000000000000000 + 2 *
                       digits //
                       10000000000000000 \
                    + 1 * digits //
                       100000000000000000 + 2 *
                       digits //
                       100000000000000000 + 9 *
                       digits
                       //1000000000000000000 + 3
                        * digits //
                       10000000000000000000
```

```
    else: # digits <= 988557646937 * 10**64
        if digits > 987274190490 * 10**64:
            q3 = 6
            # multiplier_q3 = 1.007825393982863782406809
            digits = digits + 0 * digits // 10 + 0 *
               digits // 100 + 7 * digits //1000 + 8 *
               digits // 10000 \
                           + 2 * digits // 100000 + 5 *
                              digits // 1000000 + 3 *
                              digits //10000000 + 9 *
                              digits // 100000000 \
                           + 3 * digits // 1000000000 +
                              9 * digits // 10000000000
                              + 8 * digits
                              //100000000000 + 2 *
                              digits // 1000000000000 \
                           + 8 * digits //
                              10000000000000 + 6 *
                              digits // 100000000000000
                              + 3 * digits
                              //1000000000000000 + 7 *
                              digits //
                              10000000000000000 \
                           + 8 * digits //
                              100000000000000000 + 2 *
                              digits //
                              1000000000000000000 + 4 *
                              digits
                              //10000000000000000000 + 0
                               * digits //
                              100000000000000000000 \
                           + 6 * digits //
                              1000000000000000000000 + 8
                               * digits //
                              10000000000000000000000 +
                              0 * digits
                              //100000000000000000000000
                               + 9 * digits //
                              1000000000000000000000000
        else: # digits <= 987274190490 * 10**64
            q3 = 7
            # multiplier_q3 =
               1.009135566995041505323937851 7
            digits = digits + 0 * digits // 10 + 0 *
               digits // 100 + 9 * digits //1000 + 1 *
               digits // 10000 \
                           + 3 * digits // 100000 + 5 *
                              digits // 1000000 + 5 *
                              digits //10000000 + 6 *
                              digits // 100000000 \
                           + 6 * digits // 1000000000 +
                              9 * digits // 10000000000
```

```
                                    + 9 * digits
                                    //100000000000 + 5 *
                                    digits // 1000000000000 \
                                + 0 * digits //
                                    10000000000000 + 4 *
                                    digits // 100000000000000
                                    + 1 * digits
                                    //1000000000000000 + 5 *
                                    digits //
                                    10000000000000000 \
                                + 0 * digits //
                                    100000000000000000 + 5 *
                                    digits //
                                    1000000000000000000 + 3 *
                                    digits
                                    //10000000000000000000 + 2
                                     * digits //
                                    10000000000000000000 \
                                + 3 * digits //
                                    100000000000000000000 + 9
                                     * digits //
                                    1000000000000000000000 +
                                    3 * digits
                                    //10000000000000000000000
                                     + 7 * digits //
                                    100000000000000000000000
                                    \
                                + 8 * digits //
                                    1000000000000000000000000
                                     + 5 * digits //
                                    10000000000000000000000000
                                     + 1 * digits
                                    //100000000000000000000000000
                                     + 7 * digits //
                                    1000000000000000000000000000

# Now digits has already been updated
assert digits > 9949 * 10**72
assert digits <= 10**76

# We define z = one - updated_x
# Then we truncate z to 38 significant digits
# and we convert to float128
z_int = 10**76 - digits
len_z_int = number_of_digits(z_int)

z_string = str(z_int)
z_string = z_string[:38] if len_z_int > 38 else z_string.
   ljust(38, '0')

z = float128(int(z_string), len_z_int - 76 - 38)
```

```
    # Number of terms of the Taylor series:
    terms = 15  # If we use up to q3, 15 terms should suffice
        .
    result = z
    z_to_j = z
    for j in range(2, terms + 1):
        z_to_j = z_to_j * z
        result = result + z_to_j / j


    M = 38 # number of digits of precision that we work with.

    # ln10, ln2 and ln1.1 represented as float128 with M=38
        significant digits
    ln10_M =  23025850929940456840179914546843642076
    ln2_M =    69314718055994530941723212145817656807
    ln1dot1_M = 95310179804324860043952123280765092220
    ln10 = float128(ln10_M, -M+1)
    ln2 = float128(ln2_M, -M)
    ln1dot1 = float128(ln1dot1_M, -M-1)

    lnB = float128(139029051689914208654778774582468595 30,
        -39)
    lnC = float128(129915573162005011576055556588045287 11,
        -40)

    result = result + k * ln2 + q1 * ln1dot1 + q2 * lnB + q3
        * lnC - m10 * ln10
    real_result = - result

    return real_result
```