International Journal of Information Security (2024) 23:1149–1169
https://doi.org/10.1007/s10207-023-00781-0

REGULAR CONTRIBUTION

De Bello Homomorphico: Investigation of the extensibility of the OpenFHE library with basic mathematical functions by means of common approaches using the example of the CKKS cryptosystem

Thomas Prantl[1] · Lukas Horn[1] · Simon Engel[1] · Lukas Iffländer[1] · Lukas Beierlieb[1] · Christian Krupitzer[2] · André Bauer[3] · Mansi Sakarvadia[3] · Ian Foster[3] · Samuel Kounev[1]

Published online: 27 November 2023

Abstract
Cloud computing has become increasingly popular due to its scalability, cost-effectiveness, and ability to handle large volumes of data. However, entrusting (sensitive) data to a third party raises concerns about data security and privacy. Homomorphic encryption is one solution that allows users to store and process data in a public cloud without the cloud provider having access to it. Currently, homomorphic encryption libraries only support addition and multiplication; other mathematical functions must be implemented by the user. To this end, we discuss and implement the division, exponential, square root, logarithm, minimum, and maximum function, using the CKKS cryptosystem of the OpenFHE library. To demonstrate that complex applications can be realized with this extended function set, we have used it to homomorphically realize the Box–Cox transform, which is used in many real-world applications, e.g., time-series forecasts. Our results show how the number of iterations required to achieve a given accuracy varies depending on the function. In addition, the execution time for each function is independent of the input and is in the range of ten seconds on a reference machine. With this work, we provide users with insights on how to extend the original restricted function set of the CKKS cryptosystem of the OpenFHE library with basic mathematical functions.

Keywords Homomorphic Encryption · Performance · Accuracy · Basic Mathematical Functions · Division Function · Exponential Function · Square Root Function · Logarithm Function · Min/Max Function · Telescope · Time-Series Forecasting · Box–Cox Transformation

✉ Thomas Prantl
thomas.prantl@uni-wuerzburg.de

Lukas Horn
lukas.horn@uni-wuerzburg.de

Simon Engel
simon.engel@uni-wuerzburg.de

Lukas Iffländer
lukas.ifflaender@uni-wuerzburg.de

Lukas Beierlieb
lukas.beierlieb@uni-wuerzburg.de

Christian Krupitzer
christian.krupitzer@uni-hohenheim.de

André Bauer
andrebauer@uchicago.edu

Mansi Sakarvadia
sakarvadia@uchicago.edu

Ian Foster
foster@cs.uchicago.edu

Samuel Kounev
samuel.kounev@uni-wuerzburg.de

[1] University of Würzburg, Würzburg, Germany

[2] University of Hohenheim, Stuttgart, Germany

[3] University of Chicago, Chicago, IL, USA

1 Introduction

In recent years, the amount of data generated, stored, and consumed worldwide has increased rapidly. It is estimated that it will grow to 180 zettabytes by 2025.[1] To manage this volume of data, cloud computing is useful because it provides a scalable and elastic infrastructure; businesses and organizations can easily increase or decrease their computing resources as needed to handle changing data volumes. Cloud providers also offer a variety of storage solutions that can handle large amounts of data, such as object storage and data lakes. Marc Hurd (former co-CEO of Oracle Corporation) estimates that by 2025, 80% of enterprise data centers will be moving to cloud infrastructures.[2] The reasons for migrating to public clouds are manifold; for instance, businesses can reduce their capital expenses and operating costs associated with managing and storing large amounts of data as well as gain access to a wide range of tools and services for data analysis and processing. Further, it has been repeatedly shown that energy can be saved by using cloud computing [1].

To take advantage of cloud computing, the data must be entrusted to a third party. However, there are many fraud scenarios and/or problems with data jurisdiction. For example, the cloud provider could be in the same business as the user and exploit the uploaded data, or the provider could also sell the data to a competitor. Another scenario is that the cloud provider could be compromised, and the data would then be accessible to the intruder. Additionally, issues with data sovereignty may arise; the data will be stored and subject to the laws of the country where the cloud provider operates and may be accessible to government authorities under certain circumstances. These possibilities are a major concern for medical data, which is subject to the General Data Protection Regulation (GDPR), or any other sensitive data. The described issues are a major showstopper for the proliferation of cloud-based services, as shown by a survey in Germany [2], which found that 48% of German companies have concerns about the data security of clouds, and 40% of German companies therefore decide against using an external cloud. This limitation of data security also slows down the development of artificial intelligence (AI) models.

To still benefit from the features of cloud computing without the cloud provider being able to access the data, the user can implement security best practices and use encryption to protect sensitive data. One possible solution is the application of homomorphic encryption. Homomorphic encryption allows for computations to be performed on ciphertext, obtaining an encrypted result that can then be decrypted to get the result of the computation in plaintext. This is in contrast to traditional encryption, where the data need to be decrypted first before any computations can be performed on it. Simply put, the user can store and process data in a public cloud while the cloud provider has no access to it. Technically, the user can run databases or microservices in the cloud as well as train machine learning models, while preserving the privacy of all data stored in the cloud.

Although the idea of homomorphic encryption was already introduced back in 1978 [3], and the first implementation of this method was presented in 2009 [4], homomorphic encryption has only recently been made available to developers in the form of corresponding libraries. However, existing homomorphic encryption libraries (e.g., OpenFHE or SEAL) support only addition and multiplication so far; other relevant basic mathematical functions (such as square root and logarithm) must be implemented by the user. The realization of such additional functions needed for the implementation of the CKKS [5] cryptosystem of the OpenFHE library [6] is discussed in this article. Namely, we propose implementations for the following functions: division, exponential, square root, logarithm, minimum, and maximum function. Additionally, we compare them in terms of their computational efficiency.

More precisely, for each basic mathematical function, we first provide an overview of how the respective function is calculated in the literature for the non-homomorphic case. Then, for each suitable computation approach for the homomorphic case, we determine the required multiplicative depth, accuracy, and limitations (e.g., the approach only converges to the actual value in a very small interval). Based on this analysis, we select the best approach for each function and then implement and analyze it in terms of execution time and the required number of iterations to attain a certain accuracy. After examining the basic mathematical functions, we apply homomorphic encryption in a sophisticated use case, namely the Box–Cox transformation [7], which is used in many different domains, such as time-series forecasting [8]. Since time-series forecasts rely heavily on historical data, the integration of confidential data using homomorphic encryption to augment the dataset appears highly beneficial, especially for confidential data (e.g., medical, financial, or administrative).

In summary, the core contributions of this article are:

1. We review and analyze different approaches to approximate basic mathematical functions.
2. We implement the best-performing candidate for each basic mathematical function using the CKKS cryptosystem of the OpenFHE library and analyze the necessary execution time and number of iterations to attain a given accuracy.

[1] Statista: https://www.statista.com/statistics/871513/worldwide-data-created/.

[2] The Wall Street Journal: https://www.wsj.com/articles/BL-CIOB-11316.

3. We use the extended OpenFHE library to implement the Box–Cox transformation in a time-series forecasting scenario and evaluate its performance in a real-world use case.

The results of our contributions show that the number of iterations required to achieve a given accuracy varies depending on the function. For example, only the square root, logarithm, minimum, and maximum functions managed to achieve 95% accuracy in less than 10 iterations in almost all cases. For all methods, the execution time is independent of the input and increases with the number of iterations. While the growth rate increases with the number of iterations for the exponential function, it decreases for the division, square root, minimum, and maximum function. Overall, we were able to homomorphically implement all basic mathematical functions as well as the Box–Cox transformation. The execution times of the basic mathematical functions and the Box–Cox transformation are in the range of 10 s and 4 h, respectively, on our reference machine. We see potential for further optimization, such as performing the computations on GPUs instead of CPUs, as is common in the machine learning domain, or developing specialized hardware, as is common in the cryptography domain.

To the best of our knowledge, we are the first to provide (1) guidelines on how a wide range of basic mathematical functions can be homomorphically realized and (2) a performance analysis of the homomorphically realized basic mathematical functions in terms of computation times and accuracy. The methods we propose for the realization of the respective basic functions promise applicability to arbitrary ranges of values with as minimal multiplicative depth as possible. We see significant potential to use this broad set of functions as a basis for the homomorphic realization of complex machine learning applications, such as time-series forecasting or neural networks, which consist of a string of the basic mathematical functions realized by us.

The remainder of this article is structured as follows: First, we introduce the basics of homomorphic encryption in Sect. 2 and discuss related work in Sect. 3. Then, we introduce the basic mathematical functions we selected in Sect. 4 and their implementations in Sect. 5. The results are presented in Sect. 6. Finally, we conclude the article in Sect. 7.

2 Background

In this section, we first define the basic terms of a cryptosystem and then extend them to a homomorphic cryptosystem. Before we can define the term cryptosystem, we must first define the two terms plaintext and ciphertext. By plaintext, we mean all things that can be encrypted, such as texts, letters, numbers, or vectors. We call the encryption of a plaintext ciphertext. Based on these two terms and in accordance with [9], we can now define what a cryptosystem is.

Definition 1 A cryptosystem is a tuple $(\Sigma, \mathcal{G}, \mathcal{E}, \mathcal{D})$ with the following properties: Σ is a finite, non-empty set, also called alphabet. The plaintext space $\mathcal{P}$, the key space $\mathcal{K}$, and the ciphertext space $\mathcal{C}$ are subsets of the alphabet. $\mathcal{G}$ is a probabilistic algorithm that outputs a key pair (pk, sk) chosen according to some distribution. The key pk, also called the public key, is intended for encryption. The key sk, also called the secret key, is intended for decryption. $\mathcal{E}$ takes as input a key pk and a plaintext message m and encrypts it to a ciphertext c. $\mathcal{D}$ takes as input a key sk and a ciphertext c and outputs the plaintext m. Additionally, the tuple $(\Sigma, \mathcal{G}, \mathcal{E}, \mathcal{D})$ must satisfy the following condition, since otherwise it is not guaranteed that a ciphertext can be brought back to its original form. This condition describes that for every possible key pair of pk and sk, the plaintext m encrypted with pk is decrypted back into the plaintext m with the associated sk [9].

$$\forall m \in \mathcal{P} : \forall (pk, sk) \in \mathcal{K} : \mathcal{D}(sk, \mathcal{E}(pk, m)) = m$$

Having defined the basics of a cryptosystem, we now extend it in terms of homomorphism. The goal of homomorphic encryption is to perform operations on encrypted data such as addition, multiplication, and exponentiation. More precisely, homomorphic encryption aims at encrypting a plaintext, performing an operation on it, and decrypting it again, so that the result is the same as if the operation had been performed on the plaintext. Therefore, in accordance with [10], we extend Definition 1 to meet these new requirements.

Definition 2 A homomorphic cryptosystem is defined as a tuple $(\Sigma, \mathcal{G}, \mathcal{E}, \mathcal{D}, \mathcal{F}, Evaluate)$ with the following properties: $(\Sigma, \mathcal{G}, \mathcal{E}, \mathcal{D})$ is a cryptosystem and $\mathcal{F}$ is a set of functions that can be calculated by this cryptosystem. $Evaluate$ is an algorithm that, given a key k, a function $f \in \mathcal{F}$, and a cipher text c, calculates a new ciphertext c' of the same length, or written more formally $|c'| = |c|$ [10].

The last condition is necessary because otherwise, it would be possible to append the desired calculation at the end of the ciphertext and execute it when it is decrypted. There are currently four different types of homomorphic encryption schemes. Partially homomorphic encryption describes schemes that only support one type of operation (e.g., addition). Semi-homomorphic encryption, also known as "somewhat homomorphic encryption" (SWHE), includes a cryptosystem that can support multiple operations but only on a limited set of functions. Leveled fully homomorphic

encryption includes cryptosystems that can perform arbitrary computations but only on a limited depth that needs to be known in advance; usually, the depth corresponds to the number of multiplications. A system with a depth of n supports up to n multiplications per number. Going above these numbers can lead to arbitrary results or is not supported. Cryptosystems of this type are the current standard. Finally, fully homomorphic encryption supports arbitrary operations and unbounded depth. There are not many cryptosystems that fulfill this requirement, and they are usually extremely slow because complex computations are necessary to keep the noise of the ciphertext low, given that if the noise exceeds a certain threshold, it would no longer be possible to decrypt the ciphertext.

Finally, we would like to distinguish the term depth from iteration, as both terms are quite similar, but we use them to describe different things. By the term iteration we mean the number of iterations we have performed for an iterative procedure, whereas by the term depth we mean the number of consecutive multiplications performed.

3 Related work

In this section, we review related work and show the novelty and the necessity of our contributions. To do this, we go through the functions we implemented homomorphically one by one and compare our implementation to existing approaches. Namely, these functions are the division, exponential, square root, logarithm, minimum, and maximum function. In doing so, we do not go into detail about related work that performs the computation of the functions in an unencrypted fashion.

One of the first ideas for homomorphic computation of the division function came from Lauter et al. [11]. It consisted of computing the numerator and denominator separately and returning the result as a fraction. The representation of the result of the division function as a fraction is mathematically seen from the expressiveness equivalent to the representation of the result as a decimal number; however, exactly this computation of the decimal number is the actual task of the division function. Common implementations of the division function, such as in GNU C++, return a concrete number and not a fraction. Thus, in our opinion, the implementation of the division function in [11] is not complete as it is in our case. A computational method of division of integers was proposed by Okada et al. [12]. In summary, this approach attempts to determine all theoretically possible inverses of the denominator and then test through all of them to determine the candidate that is truly the inverse. We differ from this approach in that we allow division not only of integer but also of decimal numbers. This is not possible with the approach from [12], since there would be an infinite number of candidates for floating point numbers, which could theoretically be the inverse, and thus, testing all would take an infinite amount of time. Also, a computational method for the division of integers was proposed by Babenko et al. [13]. This is based on a variation of the Euclidean division algorithm. However, the authors assume that single encrypted bits can be compared with each other. They only make this assumption and do not explain how this works in detail, nor do they provide an implementation. Therefore, it remains open whether this calculation method will ever be realizable. The authors of [14] consider the Newton–Raphson method and the Goldschmidt division algorithm for computing the division function. Both approaches require a starting value x_0, which the authors select once depending on the interval. Therefore, the information for which value from the interval the Newton–Raphson method or the Goldschmidt division algorithm should be applied is not used by this fixed choice of the starting value x_0. We differ from this work in that: (1) We have provided an overview of possible division calculation methods in order to make a reasoned choice based on performance and accuracy. (2) The authors measured the computation times of the division method only on the interval $[0, 64]$, whereas we evaluated a much larger interval $[-100, 100]\backslash\{0\}$, which also contains negative values. (3) We compare the results of homomorphically and non-homomorphically calculated results of the division function to make statements about the accuracy of our calculation method. (4) We evaluate how many iterations it takes to achieve a certain accuracy on a given interval. (5) We choose the starting value x_0 depending on the concrete value for which the division should be calculated and not only on the basis of the interval, from which the value originates, for which the division is to be calculated. In addition, we would like to mention the work of Thijs Veugen [15] and Ugwuoke et al. [16], which also include a method for computing the division function. However, since they rely on performing computations in an unencrypted manner, we do not discuss them in detail. We distinguish ourselves from these two works in that we perform the computation of the division function completely encrypted.

The authors of [14] deal not only with the calculation of the division function but also with the root function. In doing so, they propose to compute the root function using the Newton–Raphson method, just as we do. However, they do not address how to determine the initial value x_0 for the root function. Beside this, we also differ from [14] in the same ways that we did for the division function. Shortell et al. also compute the root function in [17] using the Newton–Raphson method. However, a fixed initial value x_0 is calculated for a certain interval for this purpose. Thus, the root can be computed only for small numerical ranges. Our approach of the dynamic calculation of the starting value, however, also supports the calculation of the root for larger numerical values.

Further works [18–21] focus on the computation of the square root. However, the authors do not compute the root directly but only the inverse of the root. We also use this idea, but mainly differ from these approaches in that we propose a novel approach to find a start value that only needs one homomorphic multiplication. Furthermore, we provide a detailed evaluation of the precision and performance of this approach on different intervals. In addition, we would like to mention the work in [22], which also deals with the computation of the square root function, however, in an unencrypted scenario.

With regard to the exponential function, literature already exists that deals with its homomorphic calculation. For example, several works [5, 23–25] compute the exponential function homomorphically via the Taylor series, as we do. Our work extends this existing literature by providing an overview of the methods for computing the exponential function and a reasoned choice of the Taylor series based on it. As an additional extension, our work offers an analysis of how many terms of the Taylor series one must compute for a certain accuracy, as well as a more detailed analysis of the computation times of the exponential function.

In the homomorphic encryption context, we found with [17] the only one paper that discusses the computation of the natural logarithm. The authors suggest to calculate the natural logarithm via Taylor series. We had also considered this approach to calculate the natural logarithm; however, we discarded this approach because it is only accurate for a small range around the development point [26]. Instead, we propose a new approximation method for the natural logarithm, which provides higher accuracy for wider ranges. In addition to presenting a new calculation method for the natural logarithm, we also evaluate it in terms of calculation times and accuracy.

With regard to the minimum and maximum function, there are several works in the literature [27–36] that cover this topic. In [36], the most efficient method for the homomorphic implementation of the maximum and minimum function was presented so far, which is why we have directly adopted this method. We extend the existing literature by a more detailed analysis of the required calculation times and by an analysis of how many iterations are required for a certain accuracy.

We would also like to mention that a number of works already exist (e.g., [37–39]) that have implemented subsets of the function set considered by us in order to realize the computations required for working with different models, such as neural networks, for example. In these works, there is usually no analysis of the required multiplicative depth as well as the performance of the individual functions. Instead, the latter are mostly based on approximations by means of polynomials, which are rather untypical for the computation of more complex functions. For these reasons, we do not discuss these works in more detail here.

We would also like to mention works like [40], which makes it possible to choose during the encryption phase whether the cryptosystem should have the property homomorphism or non-malleability. This is not necessary for us. Since the homomorphic property is essential for us and we therefore do not need this choice, we do not go into this work in more detail.

Finally, our work is the first to cover a broader range of mathematical functions (including the division function, root function, exponential function, logarithm function, as well as the maximum and minimum function), while assessing different approaches for the calculation of these functions and analyzing the proposed implementations uniformly with regard to the required calculation times and the number of iterations for a desired accuracy.

4 Methods for computing basic mathematical functions homomorphically

This section describes the set of identified methods for computing basic mathematical functions homomorphically. Specifically, we consider the following basic mathematical functions: division, exponential, square root, logarithm, minimum, and maximum. We first review the literature for common implementation methods and then select a method based on the criteria of multiplicative depth, accuracy, and constraints. In doing so, we determined the respective multiplicative depths ourselves and presented a novel method for calculating the logarithm.

4.1 Method selection for the division function

To realize the homomorphic computation of the division function $f(a, b) = \frac{a}{b}$, where a and $b \in \mathbb{R}$, we looked around for common methods to compute this function. The simplest and most common class of algorithms to compute the division function is the so-called digit recurrence method. Division is computed by iteratively applying the subtraction function [41]. However, this widespread class of division algorithms is not suitable for our scenario, since its program logic in the homomorphic case would depend on homomorphic encrypted values. Thus, it would not be feasible as the following example shows: The division a/b is computed by counting how many times one can subtract b from a so that the remainder is still greater than or equal to zero. Thus, after the first subtraction, one would have to test whether the remainder is greater than or equal to zero, which, as we will see later, can also be determined in the homomorphic case. However, the result of this comparison (i.e., whether the residual value is greater than or equal to zero) is also homomorphic encrypted, which means that the algorithm cannot

access the result in order to decide whether the computation should be continued.

Common methods for the approximation of functions like Taylor series or the Padé approximation are also out of question for the calculation of the division function, because these methods themselves fall back on the division function. Thus, from the possible approaches for the computation of the division function that we identified in the literature, only the Goldschmidt [42] and the Newton–Raphson [43] methods remained. These two approaches compute the division function by first iteratively computing the value of $\frac{1}{b}$ and then multiplying it by a. Thereby, both methods exhibit the same relative accuracy after a given number of iterations [42]. For this reason, we make the final choice between these two schemes based on the required multiplicative depth and the constraints of the two methods (see Table 1). Both methods are identical with respect to their constraints, namely that the initial value must be known before the function is implemented. Therefore, the selection was made on the basis of the respective required multiplicative depth. Since the Newton–Raphson method has a smaller depth, we selected it for the realization of the division function. The Newton–Raphson method is also the most frequently used approach for calculating the division function in the non-homomorphic case [44].

4.2 Method selection for the exponential function

To realize the exponential function, we first look at common implementations from practice and analyze them with respect to their use for homomorphic encryption. For example, the GNU C++ library implements the exponential function mostly by means of simple polynomials, which can be realized homomorphically. However, the GNU C++ implementations additionally also use operations like rounding or conditional statements [45]. We already discussed for the division function that conditional statements could not be implemented homomorphically. Hence, we, unfortunately, have to exclude the GNU C++ implementation of the exponential function for our homomorphic use case. For the same reason, we cannot use approaches that use look-up tables for the computation of the exponential function (e.g., [46]). Also commonly cited in the literature are hardware solutions for the computations of the exponential function (e.g., [47–49]), which we also exclude directly because we want to solve the computation of the exponential function hardware independent in software.

Thus only the computation via Taylor series, Padé approximation, or the Newton–Raphson approach remained from the procedures found by us for the computation of the exponential function. To choose between three procedures, we first consider Table 2, which lists the necessary multiplicative depth and restrictions of the three procedures. From this table, we can see that the Taylor series is more suitable than the Padé approximation and the Newton–Raphson approach for our application with regard to the multiplicative depth and limitations. This is because (1) for the Taylor series, unlike the Padé approximation, no additional parameters need to be computed and (2) the Taylor series requires the lowest multiplicative depth. According to Table 2, the Padé approximation requires a depth of $\lceil log_2(max(m,o))\rceil + d$, or if we use the Newton–Raphson method for division where d becomes $2*n+1$, a depth of $\lceil log_2(max(m,o))\rceil + 2*n+1$. The Newton–Raphson approach requires a depth of $(d+l)*n$, or if we again use the Newton–Raphson method for the division, a depth of $2*n^2+n+l*n$. In comparison, the Taylor series has a much shallower depth of $\lfloor log_2(n)\rfloor + 1$.

In addition to performance and constraints, we would also like to consider the accuracy of the methods for choosing a method to realize the exponential function. In the literature, one can often read that the Padé approximation has a better accuracy than the Taylor series (see, e.g., [50–52]). However, this is mainly the case when functions contain poles [53]. Since the exponential function has no poles, the accuracy of the Taylor series for computing the exponential function is very accurate for very large ranges of values, as we will see in the course of the paper. Furthermore, since the Taylor series does not rely on the accuracy of other functions compared to the Padé approximation, we rate the accuracy of the Taylor series higher than the accuracy of the Padé approximation. For the same reason, we also assume that the accuracy of the Taylor series is better than that of the Newton–Raphson method. Therefore, our final choice for realizing the exponential function falls on the Taylor series.

4.3 Method selection for the square root function

An overview of the methods that we could find in the literature for the computation of the root function is listed in Table 3. We have already excluded all methods that are not applicable in the homomorphic case. Thus, we excluded, for example, the computation method of the GNU C++ implementation of the root function [54], since their logic would depend on encrypted values. Assuming that for the division function again the Newton–Raphson method is employed, we can again replace d by $2*n+1$ in Table 3. This allows us to arrange the root calculation methods in ascending order with respect to the required multiplicative depth as follows for $n \geq 2$: Wilkes (depth: $3*n$) < Newton–Raphson (depth: $3*n+2$) < Halley (depth: $4*n$) < Heron (depth: $2*n^2+n$) < Bakshali (depth: $4*n^2+6n$). Thus, the Wiles method would be the best choice regarding the multiplicative depth required. However, it has a major disadvantage, as it is only suitable for values between 0 and 1. For this reason, we selected the next method in the ranking, which has the best accuracy compared to the other methods after a given number of iterations [55].

Table 1 Overview of methods for calculating the division including their required multiplicative depths and constraints

Method	Multiplicative depth	Limitations
Newton–Raphson [43]	$2 * n + 2$	Start value required
Goldschmidt [42]	$2^n + n$	Start value required

Here n stands for how many iterations are to be calculated for the respective method

Table 2 Overview of methods for calculating the exponential function including their required multiplicative depths and constraints

Method	Multiplicative depth	Limitations
Taylor series	$\lfloor log_2(n) \rfloor + 1$	
Padé-Approximation [42]	$\lceil log_2(max(m, o)) \rceil + d$	Additional coefficients required
Newton Raphson	$(d + l) * n$	Start value required

Here n stands for how many iterations are to be calculated for the respective procedure, m and o stand for the degrees of the two polynomials of the Padé-Approximation and l and d stand for the multiplicative depth of the implementation of the used logarithmic and division function

Table 3 Overview of methods for calculating the root including their required multiplicative depths and constraints

Method	Multiplicative depth	Limitations
Heron/Raphson [56]	$(d + 1) * n$	Start value required
Bakshali [57]	(2d + 4) * n	Start value required
Wilkes [58]	$3 * n$	Range limited to [0,1)
Halley [59]	$4 * n$	Start value required
Newton–Raphson [18]	$n * 3 + 2$	Start value required

Here, n stands for how many iterations are to be calculated for the respective procedure and d stands for the multiplicative depth of the implementation of the required division function

4.4 Method selection for the logarithm function

For the implementation of the logarithm function $log(x)$ with $x \in \mathbb{R}^+\backslash\{0\}$, we again studied the literature for different realization approaches. Here, we consider only the computation of the natural logarithm, since logarithms with other bases can be computed using the natural logarithm (i.e., $log_a(b) = \frac{log_e(b)}{log_e(a)}$, where e is Euler's number). In the following, if nothing else is indicated, logarithm is interpreted to refer to the natural logarithm.

According to [60, 61], the logarithm can be determined using the arithmetic–geometric mean, a power series, or a pre-calculated logarithm table. Of these possibilities, we could directly exclude the calculation with the help of look-up tables (e.g., as realized in the GNU C++ implementation [62]), since here again the logic would depend on homomorphic encoded values. Of the two remaining methods, the first computes the logarithm via the following approximation: $ln(x) \approx \frac{\pi}{2GM(1,2^{2-m}/x)} - m * ln(2)$. Here $GM(\dots)$ represents the computation of the geometric mean over its input values, p specifies the desired precision in bits, and the parameter m must be chosen so that the following inequality is satisfied: $x * 2^m > 2^{p/2}$.

The calculation by power series can be realized using a Taylor series. The "normal" Taylor series has a problem with the logarithm computation, because it is exact only for a very small value range around the development point [26]. However, this problem can be circumvented using the following relation presented in the NIST Handbook [63]: $ln(\frac{1+x}{1-x}) = 2 * \sum_{m=0}^{\infty} \frac{x^{2m+1}}{2m+1}$ for $x \in [-1, 1]$. While this equation also has the issue that its accuracy is limited to a fairly small interval, we can get around this problem by clever rewriting. Thus, when calculating $ln(z)$ with $z \in \mathbb{R}^+\backslash\{0\}$, we substitute the variable z with the equivalent fraction $\frac{1+\frac{z-1}{z+1}}{1-\frac{z-1}{z+1}}$. Substituting $\frac{z-1}{z+1}$ by the variable x afterward, we obtain $ln(z) = ln(\frac{1+x}{1-x})$. Since it is true that for $z \in \mathbb{R}^+\backslash\{0\}$, it holds that $-1 < x = \frac{z-1}{z+1} < 1$ [3] we can extend the relation as follows: $ln(z) = ln(\frac{1+x}{1-x}) = 2 * \sum_{m=0}^{\infty} \frac{x^{2m+1}}{2m+1}$ with $x = \frac{z-1}{z+1}$. In the following, we refer to this kind of logarithm calculation as the modified Taylor series.

To make a selection from these presented methods, we first consider the information in Table 4, which shows the corresponding required multiplicative depths and constraints. Here, we have additionally included the Padé approximation, since it could in principle also be applied for the computation

[3] Since $z \in \mathbb{R}^+\backslash\{0\}$ it holds $z > 0$. Multiplied by -2 it follows that $-2z < 0$. If one subtracts -1 from both sides, $-2z - 1 < -1$ follows. This inequality can obviously be extended as follows: $-2z - 1 < -1 < 1$. If we only add z everywhere, we get: $-z - 1 < z - 1 < z + 1$. Since $z > 0$ and therefore also $z + 1 > 0$, we can safely divide each by $z + 1$ and get $\frac{-z-1}{z+1} < \frac{z-1}{z+1} < \frac{z+1}{z+1}$. Truncated, this gives the inequality we are looking for: $-1 < \frac{z-1}{z+1} < 1$.

of the logarithm function. If one assumes that for the exponential and root function, the methods determined before are used, then based on the values in the table, it can be concluded that the necessary multiplicative depth of the modified Taylor series (depth: $\lceil log_2(2*n+1)\rceil + 3 + d$) is smaller than the depth of the arithmetic–geometric mean approach (depth: $3*n+2+2*d$) and the Newton–Raphson method (depth: $n*\lceil log_2(n)\rceil + d*n$). In addition, the accuracy of the arithmetic–geometric mean approach or Newton–Raphson method depends on the accuracy of either the root and division function or the exponential and division function. In contrast, the accuracy of the modified Taylor series depends only on the division function. Only the Padé approximation can have a lower multiplicative depth than the modified Taylor series if $max(m,o) < 2*n+1$. However, since the parameters m and o stand for the used polynomials of the Padé approximation, on whose size the accuracy of the approximation depends, the Padé approximation can undercut the modified Taylor series here only if corresponding accuracy is sacrificed. Since also additional coefficients have to be calculated for the Padé approximation, the modified Taylor series is the best choice with respect to depth and accuracy.

4.5 Method selection for the maximum and minimum function

For the maximum and minimum function, we refer to [36], where an approach for the homomorphic computation is proposed. The authors compute the maximum function as follows: $max(a,b) = \frac{a+b}{2} + \frac{\sqrt{(a-b)^2}}{2}$, where a and $b \in \mathbb{R}$. Accordingly, the minimum function is computed as $min(a,b) = a + b - max(a,b)$, where a and $b \in \mathbb{R}$. The authors also compare their approach against other methods used in the literature [27–35]. Based on this comparison, the following conclusions are drawn: (1) the authors' computational method for the maximum and minimum function is more efficient than other common polynomial approximation methods such as Taylor, least square, and minimax approximations and (2) it achieves (quasi-)optimal asymptotic computational complexity. For these reasons, we directly adopt the method from [36] for the maximum and minimum function.

5 Homomorphic implementation of basic mathematical functions

Now that we selected suitable methods for computing the division, exponential, square root, logarithm, maximum, and minimum functions in a homomorphic manner, we describe how we implemented the individual functions. We focus on the implementation of the division, square root, exponential, and logarithm functions, since the remaining minimum and maximum functions can be computed based on these functions.

5.1 Implementation of the division function

To realize the division function $f(a,b) = \frac{a}{b}$, with a and $b \in \mathbb{R}$, we selected the Newton–Raphson method in the previous section because it performed best in terms of multiplicative depth and accuracy. We first calculate the inverse of b using the Newton–Raphson method and then multiply it with a, that is, $f(a,b) = \frac{1}{b} * a$. The Newton–Raphson method allows us to determine the zeros of a function $g(x)$. Thus, we can use the Newton–Raphson method to compute the inverse of b if we find a corresponding function $g(x)$ that is zero at $x = \frac{1}{b}$. Probably the simplest function that satisfies this condition is the function $g(x) = \frac{1}{x} - b$. In order to determine the zero for $g(x)$, the Newton–Raphson method provides an estimate x_0 and then iteratively applies the rule from Eq. 1. As the number of iterations increases, the value of x_n approaches the zero point of $g(x)$ under the assumption that the estimated starting value x_0 was appropriately good. Thus, the rule from Eq. 1 applied to our function $g(x)$ leads to Eq. 2. It is important to mention that in Eq. 2, only multiplication and addition occur as operations, which are already supported by current homomorphic encryption libraries.

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

$$\begin{aligned} x_{n+1} &= x_n - \frac{g(x_n)}{g'(x_n)} = x_n - \frac{\frac{1}{x_n} - b}{\frac{-1}{x_n^2}} = x_n + x_n - x_n^2 * b \\ &= x_n * (2 - x_n * b) \end{aligned} \tag{2}$$

Thus, for the use of the Newton–Raphson method, we only have to select a suitable initial value x_0. Here, suitable means that x_0 should be as close as possible to the value $1/b$. To obtain this estimate in the homomorphic case is very problematic because: (1) the value of b is homomorphically encrypted and therefore not known and (2) the estimation procedure may only use the operations addition and multiplication. In the literature, there are two ways in which the initial value x_0 is determined. The first approach is to assume that b originates from a certain interval, for which one tries to choose a fixed initial value x_0 so that the Newton–Raphson method works as well as possible on this interval [64]. The second approach is to also assume that b originates from a fixed interval and that an auxiliary function $h(b) = x_0$ can be set up, with the help of which good initial values can be obtained for the interval [65]. The two approaches thus differ in whether only a fixed starting value x_0 is specified for an interval or whether the starting

Table 4 Overview of methods for calculating the natural logarithm function including their required multiplicative depths and constraints

Method	Multiplicative depth	Limitations
Modified Taylor series	$\lceil log_2(2 * n + 1) \rceil + 3 + d$	
Padé Approximation [42]	$\lceil log_2(max(m, o)) \rceil + d$	Additional coefficients required
Newton–Raphson	$(d + e) * n$	Start value required
Arithmetic–geometric Mean	$s + 2d$	Start value required

Here, n stands for how many iterations are to be calculated for the respective procedure, m and o stand for the degrees of the two polynomials of the Padé approximation, and d, e and s stand for the multiplicative depth of the implementation of the used division, exponential, and square root function

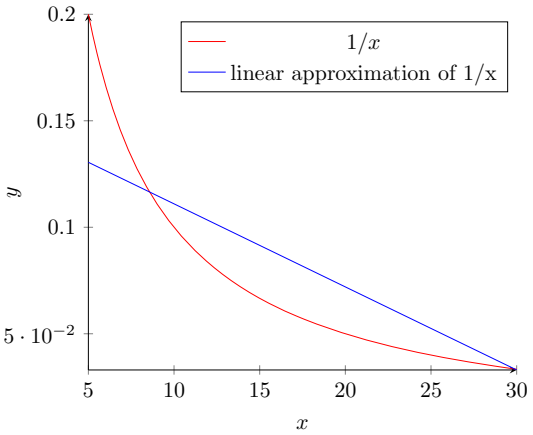

Fig. 1 Illustration of the linear approximation of the division function on the interval [5, 30]

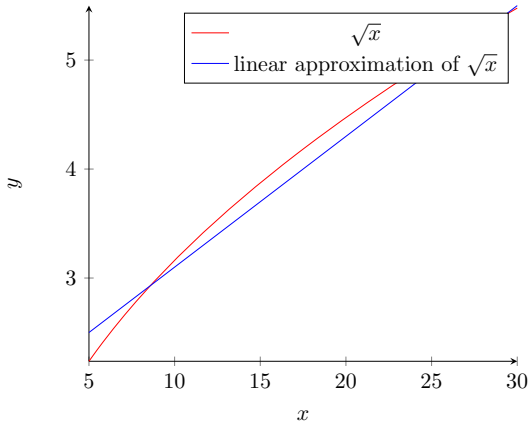

Fig. 2 Illustration of the linear approximation of the root function on the interval [5, 30]

value is determined dynamically for an interval by means of an auxiliary function $h(b) = x_0$. We experimented with both approaches and finally decided to use the second one because we could achieve better accuracy with it. As an auxiliary function, we used a simple linear approximation of $1/b$, since this was already sufficient to calculate the division function very accurately. The linear approximation of $1/b$ is illustrated for $b \in [5, 30]$ in Fig. 1. In this figure, the function $l(x) = 0.15 - 0.0039 * x$ was used as an example for a linear approximation. The initial value x_0 for the calculation of $1/b$ with $b \in [5, 30]$ would be calculated in this case as follows: $l(b) = 0.15 - 0.0039 * b$. To determine the linear auxiliary function for the inverse function we used a brute-force approach. For the known interval, we generate a balanced set of sample values (independent of the real ones) and try out gradient and axis parameters for the linear function in a certain range. After trying out all of them, we select the ones that produced the most precise result. Through this process, we get precise parameters without gaining any knowledge of the data set except the interval range.

5.2 Implementation of the square root function

For the implementation of the root function $f(a) = \sqrt{a}$ with $a \in \mathbb{R}$, we decided to also resort to the Newton–Raphson method. Thus, we must again first find a function $g(x)$ that has its zero at $x = \sqrt{a}$. The simplest and most intuitive function for which this is the case is probably the function $g_1(x) = x^2 - a$. However, this would lead to a division in every step of the iteration, which should be avoided to keep the approximation error small. Therefore, we calculated $1/\sqrt{a}$ which is the zero of the function $g_2(x) = \frac{1}{x^2} - a$. Having this result we only have to multiply it with a to obtain $\sqrt{a}$. In order to calculate $\sqrt{a}$ with the help of the Newton–Raphson method and the function $g_2(x)$, we need an initial value x_0, to which we have to iteratively apply the rule from Eq. 1. With our function $g_2(x)$, this rule takes the form from Eq. 3. The iteration rule from Eq. 3 only requires multiplication and additions and contains two fixed constants. However, this is no problem since we are capable of doing ciphertext–plaintext multiplication and addition.

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)} = x_n * \left(1.5 - \frac{a}{2} x_n^2\right) \tag{3}$$

Thus, for the calculation of the root function by means of the Newton–Raphson method, we only have to select a suitable initial value in each case. Analogous to the division function, we have the following two possibilities: (1) determine a fixed initial value x_0 for an interval, or (2) calculate the initial value x_0 dynamically by means of a auxiliary function $h(x)$. In order to choose between the two approaches,

we experimented with both approaches and decided to use the second approach because we could achieve better results with it. As an auxiliary function, we again chose a linear approximation, since this was already sufficient to calculate the root function very accurately. A linear approximation for the interval $a \in [5, 30]$ is shown in Fig. 2. Here, the root function is approximated by the linear auxiliary function $l(x) = 0.12 * x + 1.9$ as an example. In this case, the starting value for the computation of the root of $a \in [5, 30]$ would be calculated as follows: $l(a) = 0.12 * a + 1.9$. To determine the linear auxiliary function for the square root function, we again used the brute-force approach explained in Sec. 5.1.

5.3 Implementation of the exponential and logarithmic function

For the implementation of the exponential function, we use the Taylor series, which is also how the exponential function is defined. In contrast to the Newton–Raphson method, we do not need an estimated starting value x_0 but only the value x for which the function e^x is to be evaluated and the number of terms of the Taylor series that are to be considered. For simplicity, we refer to the terms of the Taylor series as iterations in the rest of the paper. The evaluation of the Taylor series itself involves only divisions, multiplications, and additions. Since we already implemented all these operations, we only had to string them together for the implementation of the exponential function. To be able to calculate larger values for e^x more easily, we used an additional mathematical relation: $e^{a+b} = e^a * e^b$. This allows us to reduce higher powers of e^x to smaller powers, (e.g., instead of e^x compute $e^{0.5*x}$ and then multiply the result by itself). More formally, we calculate e^x as follows: $e^x = \Pi_{i=1}^{r} e^{(x/r)}$ with $r \in \mathbb{N}$. Here, the value r must be specified beforehand for the implementation, depending on which range of values is considered. This technique for reducing the exponent is in principle applicable to all realizations of the exponential function. The r multiplications of $e^{x/r}$ increase the multiplicative depth by the term $\lceil log(r) \rceil$. For this, n, the number of iterations needed for the particular realization method of the exponential function, decreases. Therefore, r should be chosen in such a way that the multiplicative depth saved by the smaller iteration depth is larger than the multiplicative depth required for the r multiplications. After various trials, we found that for $r = 32$, we achieved good results for our measurements.

The situation is similar for the implementation of the logarithm. Our chosen method for the logarithm is based only on the operations addition, multiplication, and division. To compute $log(x)$ based on the logarithm with smaller input values, we exploit the following mathematical relationship: $log(a * b) = log(a) + log(b)$. If we now rewrite x as $x * \frac{n}{n}$ with $n \in \mathbb{N}$, we can rewrite $log(x)$ as follows: $log(x) = log(x * \frac{n}{n}) = log((\frac{x}{n}) * n) = log(\frac{x}{n}) + log(n)$. Thus, the computation of $log(x)$ can be traced back to the computation of $log(\frac{x}{n})$, which must then be added to the value of $log(n)$. Since the value for n must be fixed for the implementation before the program is executed, the value $log(n)$ can be pre-computed in the non-homomorphic case and then stored homomorphically encrypted. For our measurements later, we chose $n = 10$ because we achieved good results with it.

6 Evaluation

Now that we explained in detail the homomorphic realization of the division function, root function, exponential function, logarithm function, and minimum and maximum functions, we analyze these functions regarding their computation times and the number of required iterations for a given accuracy. We first look at each function individually and then examine combinations of them for composed functions. Finally, we evaluate a homomorphic implementation of the Box–Cox transformation [7], which is often used in the context of time-series analysis as implemented in tools such as Telescope [8].

As a measurement setup, we use a HPE ProLiant DL360 Gen9 server. This server has 8 CPU cores with 2.6 GHz each (Intel (R) Xeon (R) CPU E5-2640 v3 @ 2.60 GHz) and 32 GB DIMM DDR4 RAM. We used Ubuntu 20.04 LTS as the operating system. For the implementation of the underlying homomorphic cryptosystem, we used the open-source library OpenFHE v1.0.3 [6] and its implementation of the CKKS cryptosystem.

6.1 Evaluation of the division function

We begin the analysis of the division function with the number of iterations required for a given accuracy. By iteration, we mean how many steps of the respective iterative computation procedure were executed. In the case of division, for example, the number of iterations would correspond to the value n in Eq. 2. Since we trace the computation of the division $\frac{a}{b}$ back to the computation of the inverse of b, which is then multiplied by a, we focus our analysis on determining the inverse of b. To do this, we must first specify the area from which b originates. At a first glance, this may seem unusual, since one does not have to make this specification in conventional programming languages. However, this is only because this is done automatically in conventional programming languages and one does not have to take care of it oneself in most cases. For the analysis of the required iterations, we consider all intervals from the following set:

$\{[a, b] | (a, b \in [-100, 100]) \backslash \{0\} \wedge (a < b) \wedge (a\%0.1 = 0) \wedge (b\%0.1 = 0)\}$.

In simpler terms, we tested the inverse determination on all possible intervals between −100 and 100 in 0.1 steps excluding 0. We chose the interval −100 to 100 because: (1) we wanted to test the inverse determination for both positive and negative values and (2) we preferred to analyze a smaller interval in detail rather than a larger one in a coarse-grained manner. In the future, we plan to further expand the range from which intervals can be taken. We therefore see our evaluation primarily as a proof of concept. The same applies to the following evaluations of the other basic mathematical functions.

For a specific interval, we then looked at all values in 0.1 steps; for example, for the interval [1,1.5], we tested the computation of the inverse of the values 1.1, 1.2, 1.3, 1.4 and 1.5. Furthermore, for our evaluation, we aimed at achieving an accuracy of 0.1 for the respective interval. For us, an accuracy of 0.1 means that for 95% of the values of the interval, the homomorphically computed inverse deviates from the true value of the inverse by a maximum of 0.1. Here, we have chosen the 95% hurdle following a 95% confidence interval.

The required number of iterations for an accuracy of 0.1 for intervals between −100 and 100 is illustrated in Fig. 3. We use −1 iterations as an encoding for invalid intervals. In Fig. 3, it is immediately noticeable that −1 iterations are required for all values from the triangle with the vertices (−100,−100), (100,100) and (100,−100). This is because there are invalid intervals in this area, to which we have directly assigned the value −1. An interval is invalid for us if its upper interval limit is smaller than its lower interval limit. For the valid intervals, the figure shows that the number of required iterations increases as the value of the respective interval approaches 0. Thus, the intervals whose upper interval boundary is at most −13, or whose lower interval boundary is at least 13, need only one iteration for an accuracy of 0.1. For intervals containing values between −13 and 13, the number of required iterations increases the closer their values are to zero. This is because the value of $\frac{1}{b}$ tends to infinity as the value of b approaches 0. This leads to the required iterations diverging toward infinity in this case. To illustrate this behavior graphically, we show the number of iterations up to 10 as a heatmap marking the area around 0, which requires more than ten iterations, with the color pink and the note "number of iterations > 10".

Now that we evaluated the number of iterations required for the inverse determination of b with an accuracy of 0.1 for intervals from the range [−100,100], we next consider the computation times. To this end, we measured the interval [−100,100] in 0.1 steps for the inverse determination of b and varied the number of iterations between 1 and 10 in one step for each value of b. We thus measured the times of inverse determination of $b = -100$ for $1, 2, \ldots, 10$ iterations and then measured the times of inverse determination of $b = -99.9$ for $1, 2, \ldots, 10$ iterations, and so on.

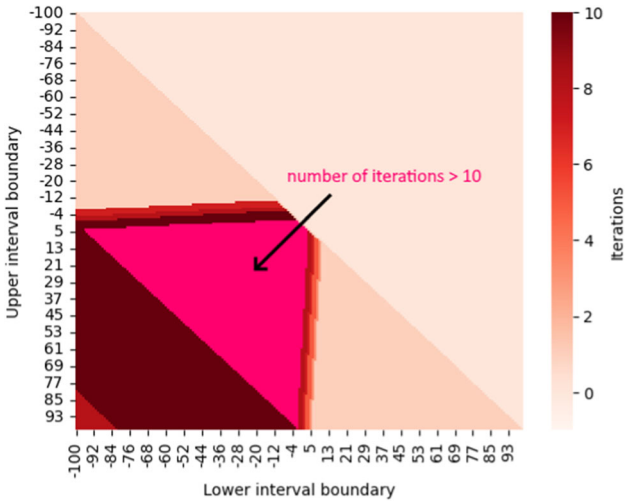

Fig. 3 Visualization of the required iterations to compute the inverse for values from different intervals with an accuracy of 0.1

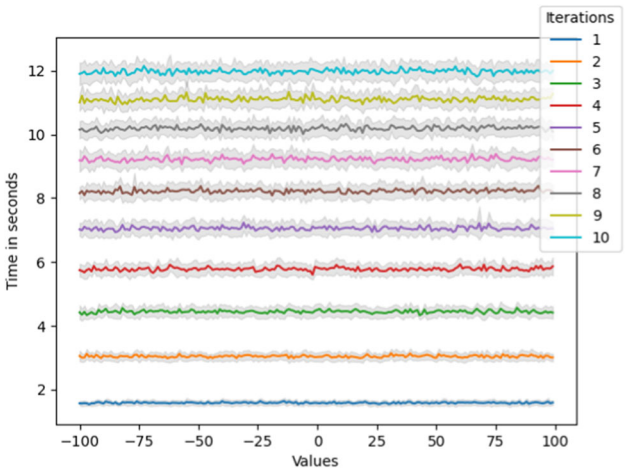

Fig. 4 Required time to determine the inverse of b, where $b \in [-100, 100]\backslash\{0\}$. The iteration depth of the iterative computation procedure was varied between 1 and 10

The computation times determined in this way are shown in Fig. 4, where we used the standard deviation as a measure of accuracy. Based on this figure, the following conclusions can be drawn: (1) The computation times of the inverse determination increase as the number of iterations increases. (2) Although the computation times increase with the number of iterations, this increase is not linear, but it continues to decrease with each iteration. This is because the homomorphic computations of addition and multiplication become faster as more of the provided multiplicative depth is already used up. (3) The computation time depends mainly on the number of iterations and not on the value whose inverse is to be determined. (4) The homomorphic inverse determination is in the range of seconds and is thus significantly slower than the non-homomorphic inverse determination, which typically requires significantly less than 1 s.

6.2 Evaluation of the exponential function

For the analysis of the exponential function, we proceed analogously to the division function. We again first consider the number of required iterations to achieve an accuracy of 0.1 for an interval and then evaluate the required computation times. For an accuracy of 0.1, we again assume that for 95% of the values of the considered interval, the homomorphic computation of the exponential function deviates from the true value by a maximum of 0.1. For computing e^x, we again specify a range from which the value x can be taken. We chose the interval [−30,30] because: (1) we want to consider both positive and negative values for x; (2) we prefer to analyze a small range in detail rather than a large range coarsely, and (3) with e^{30}, we are already in the two-digit trillion range. For the analysis of the required iterations, we tested the exponential function on all possible intervals between −30 and 30 in 0.1 steps, that is, we considered all intervals from the following set:

$$\{[a,b]|(a,b \in [-30,30])\} \wedge (a < b) \wedge (a\%0.1 = 0) \wedge (b\%0.1 = 0)\}$$

The number of required iterations to achieve an accuracy of 0.1 for intervals from the range [−30,30] is shown in Fig. 5. Here, the invalid intervals, that is, in the triangle with the vertices (−30,−30), (30,30), and (30,−30), are again assigned the value −1. The intervals in this triangle are invalid, as their upper limit is smaller than their lower limit in each case. Based on the figure, the following conclusions can be drawn: (1) As long as the upper interval boundary is negative, only one iteration is required for an accuracy of 0.1. This is because the function e^x converges to 0 very fast for $x \to -\infty$. (2) In the positive range, as the x value increases, the number of iterations required for an accuracy of 0.1 increases. (3) At least from an upper interval boundary of 20, more than 10 iterations are needed for an accuracy of 0.1, which we have marked in the figure with the color pink and the note "number of iterations > 10".

Now that we analyzed the required number of iterations to achieve an accuracy of 0.1 for intervals from the range [−30,30], we evaluate the computation times as shown in Fig. 6. For this purpose, we measured for the range [−30,30] in 0.1 steps in each case how long the computation of the exponential function requires for up to 10 iterations. Thus, for $x = -30$, we measured how long $1, 2, \ldots, 10$ iterations take, then how long $1, 2, \ldots, 10$ iterations take for $x = -29.9$, and so on. Here, again, we used the standard deviation as the accuracy measure. Figure 6 allows us to draw the following conclusions: (1) Computing one iteration and computing two iterations take the same amount of time. (2) From 3 iterations on, the higher the number of iterations, the higher the required computation time. (3) In contrast to the determination of the inverse, the distance between the computation times of two

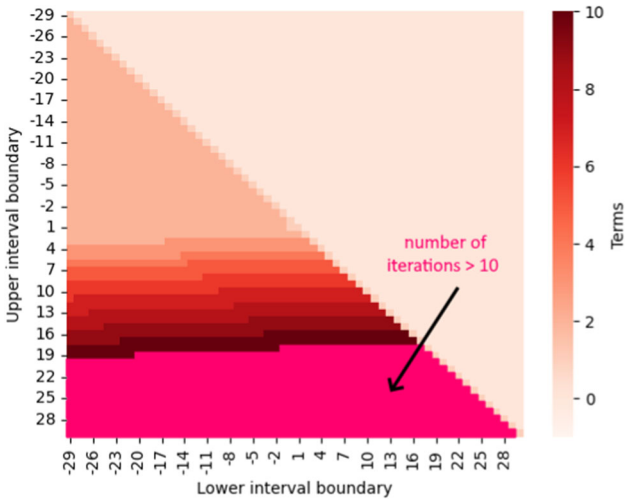

Fig. 5 Visualization of the required iterations to compute the exponential function for values from different intervals with an accuracy of 0.1

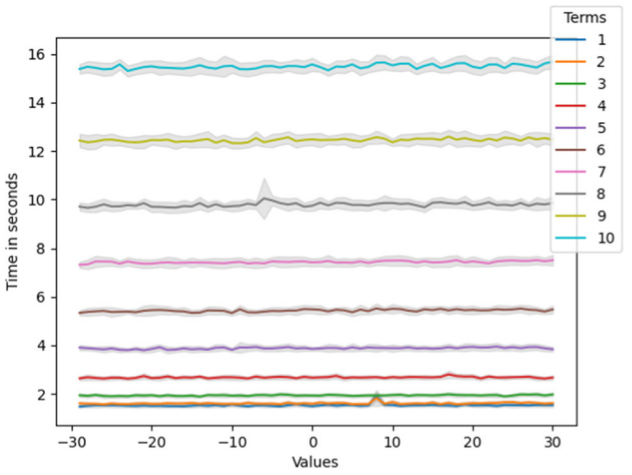

Fig. 6 Required times to determine e^x, where $x \in [-30, 30]$. The iteration depth of the iterative computation procedure was varied between 1 and 10

successive iterations increases from 3 iterations on. This is because, although the homomorphic operations addition and multiplication become faster with increasing iteration depth, the computation time per additional iteration also increases. Therefore, we explain the increasing distance between the computation times of two successive iterations by the fact that the computation effort per additional iteration increases more significantly than the basic homomorphic operations become faster. This was not the case with the inverse determination, since the computation expenditure per iteration was constant. (4) The computation times are independent of the concrete x value. (5) The homomorphic computation of 10 iterations for the exponential function is still in the (two-digit) second range, and it is clearly slower than the non-homomorphic computation, which typically requires less than 1 s.

6.3 Evaluation of the square root function

For the evaluation of the square root function, we also proceed analogously to the division function; we first consider the required number of iterations to achieve an accuracy of 0.1 for an interval and then evaluate the required computation times. Before conducting the measurements for the computation of $\sqrt{x}$, we again had to specify a range from which the value x can be taken. We chose the range [0,200] because: (1) we preferred to analyze a smaller range in detail rather than a larger range coarsely; (2) for the computation of the root generally only positive values come into question; and (3) the computation times for this range were already very complex, as we will see later. For the analysis of the required iterations, we considered all intervals from the set

$$\{[a,b]|(a,b \in [0,200])\} \wedge (a < b) \wedge (a\%0.1 = 0) \wedge (b\%0.1 = 0)\}.$$

That is, we tested the square root function on all possible intervals between 0 and 200 in 0.1 steps. The number of iterations required to achieve an accuracy of 0.1 for intervals from the range [0,200] is shown in Fig. 7. Here, we again assigned the invalid intervals in the triangle with the vertices $(0,0)$, $(200,200)$, $(200,0)$ the value of -1. Based on the graph, we can draw the following conclusions: (1) The number of iterations required for an accuracy of 0.1 increases when the upper interval boundary increases and/or the lower interval boundary decreases. (2) We can compute the root with an accuracy of 0.1 with a maximum of 10 iterations almost on the entire range considered. Only if the lower boundary is too close to 0, more than 10 iterations are needed. We have marked this area accordingly with the color pink and the note "number of iterations > 10".

The computation times for the root function $\sqrt{x}$ for intervals from the range [0,200] are illustrated in Fig. 8. We again used the standard deviation as a measure of accuracy. For this purpose, we measured the range [0,200] in 0.1 steps, varying the iterations between 1 and 10. Thus, for $x = 0$, we measured how long the root computation takes with $1, 2, \ldots, 10$ iterations each. Then, we measured how long $1, 2, \ldots, 10$ iterations take for $x = 0.1$, and so on. The depicted required computation times in Fig. 8 lead us to the following conclusions: (1) The computation times increase with the number of iterations and are independent of the concrete x value. (2) The distance between the computation times of two successive iterations decreases as the number of iterations increases. This is due to the fact that the computation costs per iteration are constant for the root function and the homomorphic operations addition and multiplication become faster with increasing depth. (3) The computation times required for the homomorphic root function are in the range of (sometimes

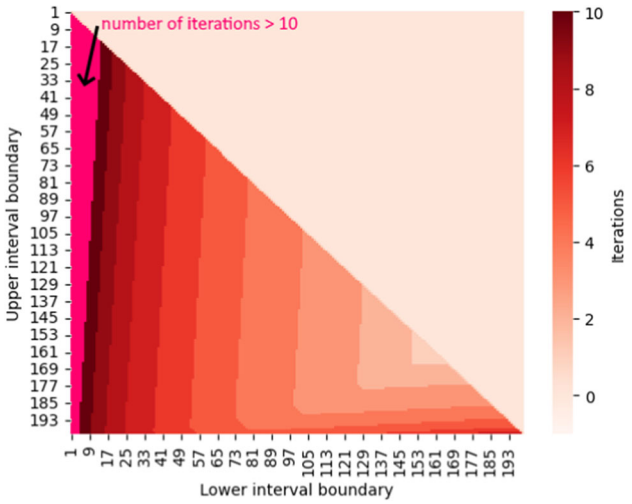

Fig. 7 Visualization of the required iterations to compute the square root function for values from different intervals with an accuracy of 0.1

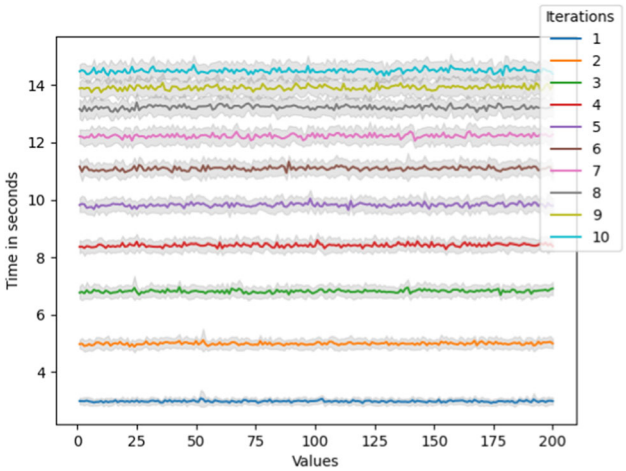

Fig. 8 Required times to determine $\sqrt{x}$, where $x \in [0, 200]$. The iteration depth of the iterative computation procedure was varied between 1 and 10

double-digit) seconds and are thus significantly longer than in the non-homomorphic case, where typically less than 1 s is required.

6.4 Evaluation of the logarithm function

The evaluation of the logarithm function $log(x)$ is also analogous to the division function. We again set a target accuracy of 0.1 and first specify the range from which the value x can be taken. For the same reasons as for the root function, we have chosen the range $]0, 200]$. In contrast to the root function, we exclude 0, given that $log(0)$ is not defined. From

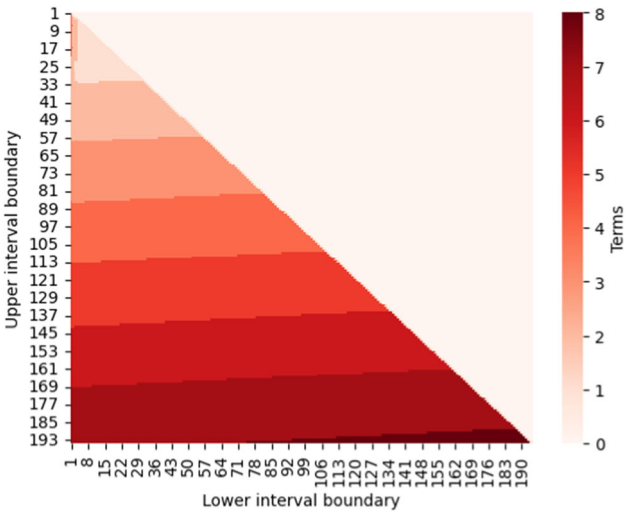

Fig. 9 Visualization of the required iterations to compute the logarithm function for values from different intervals with an accuracy of 0.1

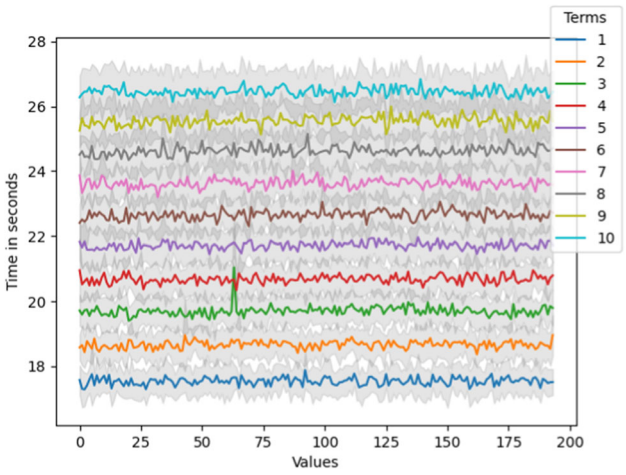

Fig. 10 Required times to determine $log_e x$, where $x \in]0, 200]$. The iteration depth of the iterative computation procedure was varied between 1 and 10

the range $]0, 200]$, we considered the following sets for our measurements:

$$\{[a, b] | (a, b \in]0, 200]) \wedge (a < b) \wedge (a\%0.1 = 0) \wedge (b\%0.1 = 0)\}.$$

To illustrate, we again measured all possible intervals between 0 and 200 in 0.1 steps. The number of iterations needed to compute $log(x)$ with an accuracy of 0.1 and $x \in]0, 200]$ are illustrated in Fig. 9. Here, all invalid intervals in the triangle (0,0), (200,200), (200,0) were again assigned the value of -1. This figure allows us to draw the following conclusions: (1) The number of iterations needed for an accuracy of 0.1 increases the larger the upper interval limit is, that is, the closer the value of x comes to the value 200. (2) We can always achieve an accuracy of 0.1 for the range under consideration with a maximum of 8 iterations.

The computation times for the logarithm function $logx$ for intervals from the range $]0, 200]$ are illustrated in Fig. 10, where we again use the standard deviation as a measure of accuracy. We measured the range $]0, 200]$ in 0.1 steps, varying the iterations between 1 and 10. Thus, for $x = 0.1$, we measured how long the computation of $log(0.1)$ takes with $1, 2, \ldots, 10$ iterations each. Next, we measured how long the computation of $log(0.2)$ needs for $1, 2, \ldots, 10$ iterations, and so on. Based on Fig. 10, we can draw the following conclusions: (1) The computation times are independent of the concrete x value and increase with the number of iterations. (2) The homomorphic computation time for the logarithm function is significantly higher than in the non-homomorphic case, that is, the time needed for the homomorphic computation is in the range of (sometimes double-digit) seconds compared to typically less than 1 s for the non-homomorphic case.

6.5 Evaluation of the maximum and minimum functions

For the evaluation of the maximum function $max(a, b)$, we first specify the range from which a and b can originate; more specifically, we take the range $[-10, 10]$. One reason for the choice of this interval is to consider positive values as well as negative values. On the other hand, in contrast to the previous functions, the number of test cases increases quadratically for the maximum function. For example, if we consider the interval $[0, 5]$ to be measured in 0.1 steps, there are $\frac{5-0}{0.1} * \frac{5-0}{0.1} = 2500$ test cases, whereas for the same interval for the root function there were only $\frac{5-0}{0.1} = 50$ test cases. Since our measurements for the single analysis of the function and its interaction altogether took several months, we had to concentrate on a smaller interval that we could measure in a fine-grained manner. Specifically, we considered the following sets:

$$\{[a, b] | (a, b \in] - 10, 10]) \wedge (a < b) \wedge (a\%0.1 = 0) \wedge (b\%0.1 = 0)\}.$$

The required number of iterations for an accuracy of 0.1 when computing the maximum function $max(a, b)$ with $a, b \in [-10, 10]$ is shown in Fig. 11. Here, we again assigned the value of -1 to the invalid intervals in the triangle $(-10,-10)$, $(10,-10)$, and $(10, 10)$, that is, the intervals where the upper bound is smaller than the lower bound. From Fig. 11, it can be seen that for the range $[-10, 10]$, we never needed more than 8 iterations for the computation of the maximum function.

The homomorphic computation times for the maximum function $max(a, b)$ with $a, b \in [-10, 10]$ are shown in Fig. 12. For overview reasons, we have chosen a 2D representation of the measured values, which consist of tuples

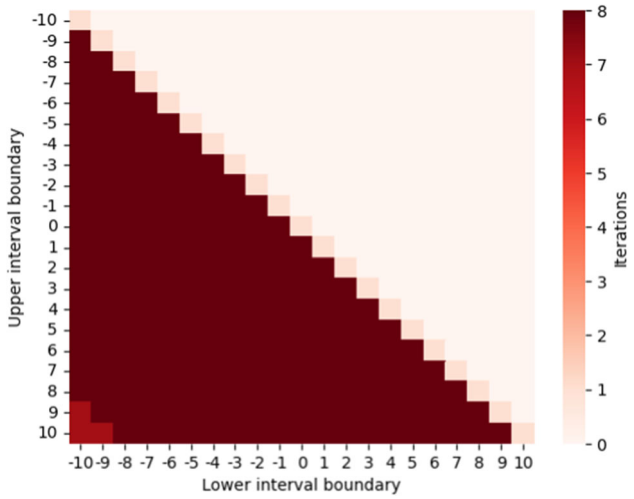

Fig. 11 Visualization of the required iterations to compute the maximum function for values from different intervals with an accuracy of 0.1

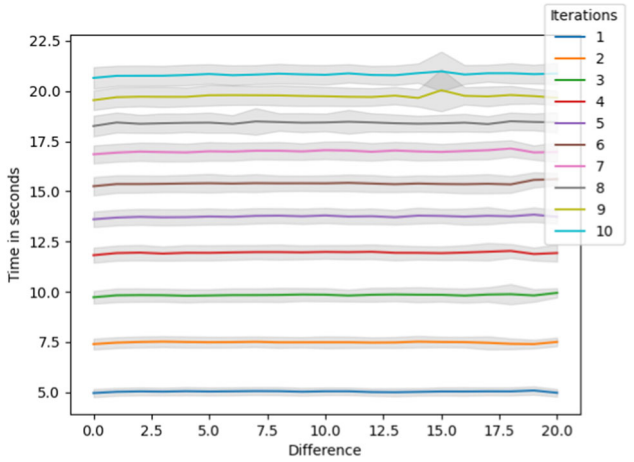

Fig. 12 Computation times to determine $max(a, b)$, where $a, b \in [-10, 10]$ and $a \leq b$. The iteration depth of the iterative computation procedure was varied between 1 and 10. The x-axis represents the difference between a and b

of 3 values (a, b, time required) each of which must be determined for different number of iterations. For this purpose, we plot on the x-axis the distance between a and b and on the y-axis the required time for 1,2,..., 10 iterations. Thus, each x value stands for a set of different cases, which have in common that the distance between a and b is equal; for example, an x-value of 10 stands for the following cases: $\{max(a, b)|a, b, \in [-10, 10]$ and $|a - b| = 10\}$. That these cases need the same computation times is due to the fact that the computation of the maximum function is traced back to the computation of the root function, which receives the squared value of the difference between a and b as input parameter. Thus, the same root computation is performed for all tuples (a, b) if they are equal with respect to the distance between a and b. The computation times of the maximum function $max(a, b)$ shown in Fig. 12 allow us to draw the following conclusions: (1) the computation times are independent of the choice of $a, b \in [-10, 10]$. (2) The computation times increase as the number of iterations increases. (3) The computation of the maximum function in the range [-10,10] takes between 5 s and 22 s. Thus, the homomorphic computation of the maximum function is significantly slower than its non-homomorphic computation, but it is still feasible in the range of seconds.

The evaluation of the minimum function was carried out in the same way as the evaluation of the maximum function. Since we trace the minimum function back to the maximum function and only need an additional addition and subtraction, the results of the analysis for the minimum function correspond to those for the maximum function. For this reason, we do not analyze the minimum function in detail here, but we show in the appendix the corresponding measurement results (cf. Figs. 17, 18).

6.6 Evaluation of combinations of the functions

To evaluate how our approach performs when combining multiple functions, we realized a complex use case homomorphically, namely the Box–Cox transformation. This transformation has applications in many fields, such as time-series forecasting [8]. In the following, we first present the concept of the Box–Cox transformation and then evaluate our homomorphic Box–Cox transformation in terms of performance and accuracy.

6.6.1 Concept of the Box–Cox transformation

The idea behind the Box–Cox transformation [7] is to transform a non-normal distribution into a normal-like distribution by means of Eq. 4 with a carefully chosen parameter λ.

$$Y = \begin{cases} \frac{y^{\lambda}-1}{\lambda}, & \text{if } \lambda \neq 0 \\ log(y), & \text{else} \end{cases} \tag{4}$$

An example of such a transformation can be seen in Fig. 13. The left part of the figure shows a non-normal distribution. If we apply the Box–Cox transformation to it, we get a distribution that resembles a normal distribution, as shown in the right part of the figure.

The biggest challenge in computing the Box–Cox transformation is to determine a suitable λ. We use Guerrero's method [66] for this task, which is also used, for example, in R Studio for the computation of lambda for the Box–Cox transformation [67]. The idea of the Guerrero method is to test different values for λ and choose the value that has the lowest coefficient of variation c for the sub-series of the time series ts [68]. By a time series, we mean a vector

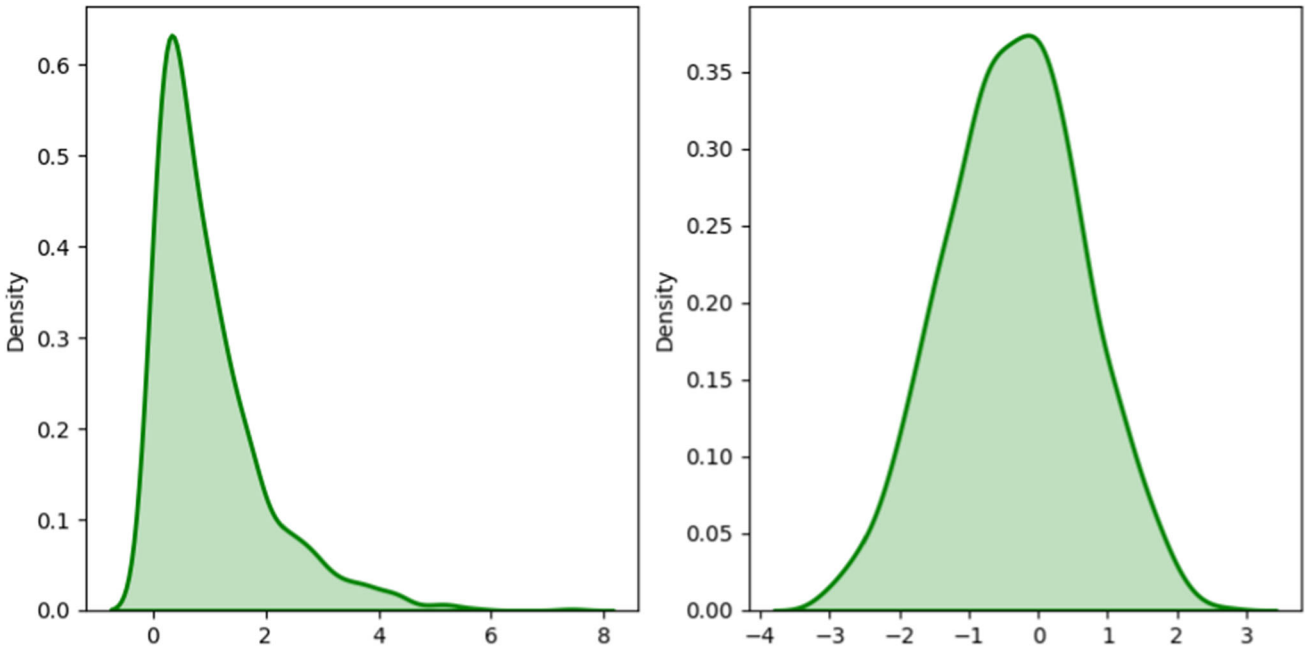

Fig. 13 An exemplary illustration of the transformation of a non-normal distribution (left) into a distribution that resembles a normal distribution (right) using the Box–Cox transformation (the example was created according to [70])

$ts = \left\{ \begin{pmatrix} v_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} v_n \\ t_n \end{pmatrix} \right\}$ (with $t_1 < t_2$, $t_2 < t_3$ and so on) that recorded the measured values $v_1, \ldots, v_n$ at certain times $t_1, \ldots, t_n$, respectively. To divide ts into sub-series, the measurements that belong to a period are combined to a set. As a concrete example of a time series to which we apply the Box–Cox transformation, we use the birth figures in New York [69]. A timestamp t_i would thus be a month of a year and the measured values v_i would be the number of children born in New York in this month. In our case, the period would be one year and thus have a length of 12. In the following, we refer to the sub-series for year i as period x_i. For the sub-series or periods $x_1, \ldots, x_n$ generated in this way, the parameter c must now be determined for different values of λ in order to subsequently select the value for λ that has the lowest value for c. The workflow for computing the parameter c for a specific λ is illustrated in Fig. 14. First, for each period x_i, we compute the ratio y_i of the standard deviation to its mean. Then, the value z_i is computed from each of the values y_i by exponentiating y_i with $(1-\lambda)$. For the z_i values computed in this way, we then determine their standard deviation and mean value, since the parameter c is computed from the ratio of these two values. In order to be able to perform these consecutive computations in a homomorphic setting, we had to bootstrap some intermediate results. Otherwise, the noise, which is part of the CKKS cryptosystem, would have become so large that the OpenFHE library would have aborted further computations, since their results could no longer be decrypted. The intermediate results that we bootstrapped and/or for which we had to apply bootstrapping during their calculation are marked in the figure with a red arrow 14.

At this point, we would like to point out that during the determination of the most suitable value for λ all values, except for the value for λ, are encrypted. The reason why we do not encrypt the respective value for λ is that (1) the final selected value of λ does not allow one to draw conclusions about the time series ts nor about the transformed time series, and (2) since the implementation of Box–Cox is supposed to be time-series-independent, appropriately common values have to be chosen for the implementation, which have to be tested in sequence. If the selected value for λ is to be kept secret, the concrete test values including the sequence would have to be kept secret. Given that the value of λ does not allow an attacker to draw any conclusions, we do not think it is necessary to keep the selected value for λ secret. Nevertheless, one could also perform the computation of z_i with an encrypted value for λ. To do this, one would have to rewrite z_i as follows: $z_i = \frac{y_i}{e^{\lambda * ln(y_i)}}$.

6.6.2 Evaluation of the homomorphic implementation of the Box–Cox transformation

To evaluate our homomorphic realization of the Box–Cox transformation in terms of performance and accuracy, we first present the workflow of our implementation. This is illustrated in Fig. 15 and consists of the sequential calculation of the parameters, $c_1 \ldots c_4$, the subsequent selection of the λ

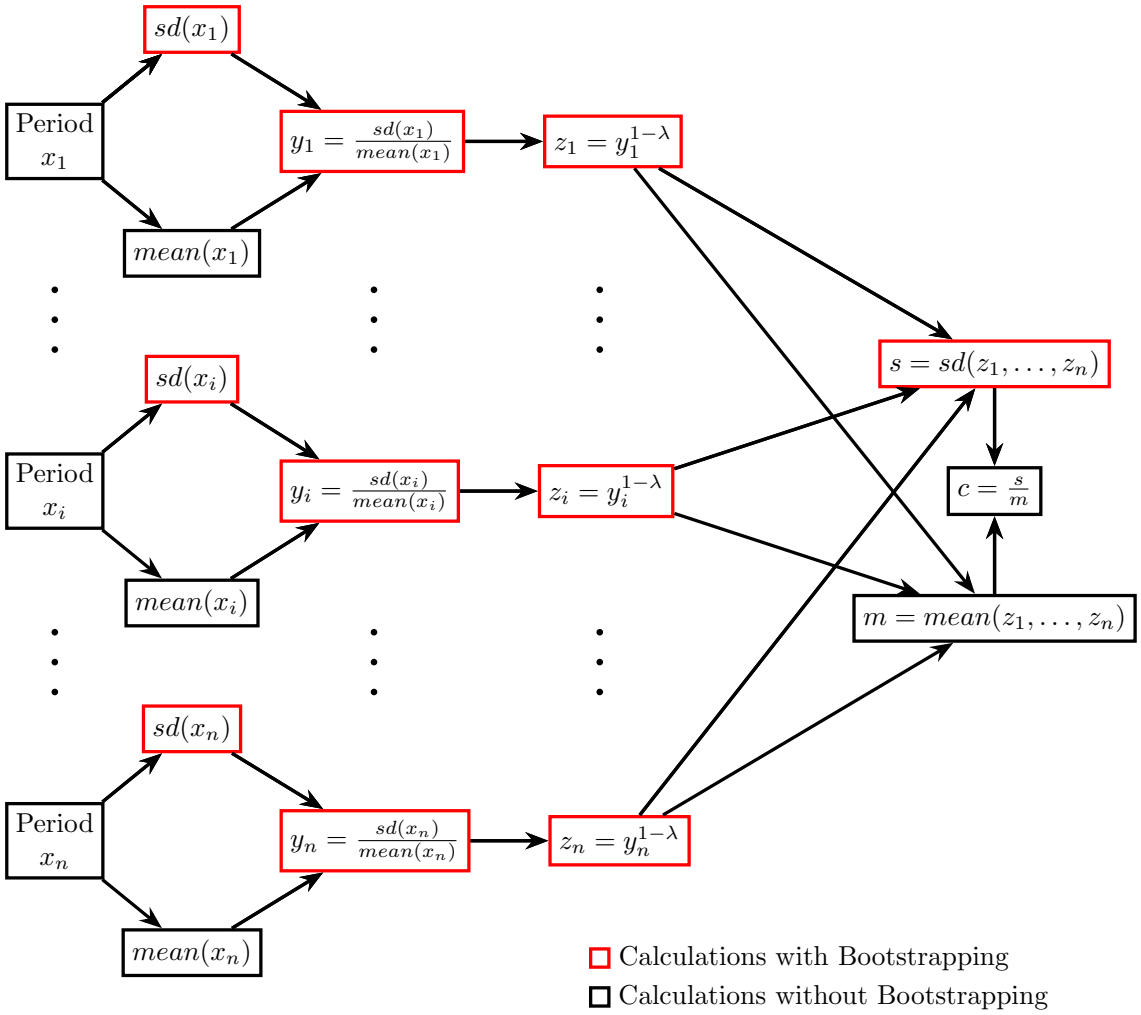

Fig. 14 Illustration of the calculation of the parameter c for the Guerrero method to determine the parameter λ for the box–cox transformation

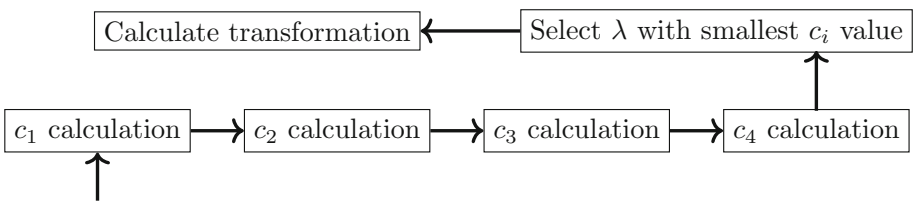

Fig. 15 Workflow of the calculation steps of your homomorphic Box–Cox transformation

with the lowest c_i value and the final calculation of the transformation of the original values using the selected λ. For each of these steps, we evaluate the required calculation time and the accuracy. The required times of the individual steps are listed in Table 5. It is striking in this table, that the most expensive calculation is the determination of the λ with the lowest c_i value, which, in the non-homomorphic case, is probably one of the cheapest calculations, since only the minimum of four numbers must be calculated, which can be realized by three comparisons. In total, we needed 4.09 ± 0.01 hours for the homomorphic calculation of the Box–Cox transformation. In the non-homomorphic case, on the other hand, the Box–Cox transformation only requires 3.4 ± 0.01 milliseconds. Thus, the calculation time of the Box–Cox transformation using our homomorphic realization increases by 432×10^6 percent compared to the non-homomorphic calculation. However, our implementation still offers some optimization possibilities, such as the parallel calculation of the c_i values, which are independent of each other. This would roughly shorten the calculation times in the homomorphic case to roughly 2.5 hours. The homomorphic realization of the Box–Cox transformation is thus, as expected, significantly slower than its non-homomorphic realization, but our measurements prove that the computations are still feasible within

Table 5 Calculation times of the different steps of our homomorphic realization of the Box–Cox transformation

Step	Required time in seconds
c_1 calculation	1806.48 ± 14.07
c_2 Calculation	1863.51 ± 17.97
c_3 Calculation	1570.91 ± 14.56
c_4 Calculation	1818.01 ± 24.97
λ Selection	6832.48 ± 9.23
Transformation Calculation	828.64 ± 2.41

Table 6 Accuracy of the homomorphically calculated intermediate results

λ	Non-homomorphically computed c value	Homomorphically computed c value
-1	0.16610	0.16379
0	0.14184	0.14068
1	0.15569	0.15566
2	0.19852	0.19841

a few hours, and homomorphic encryption thus represents a promising realistic option for implementing data protection for cloud applications, where data security is top priority.

In addition to the time required for the homomorphic calculation of the Box–Cox transformation, the accuracy achieved in the homomorphic variant is of course also important. To do this, we first look at the accuracy of the calculated intermediate results, which are listed in Table 6. From this table we can see that the homomorphic calculation of the c_i values is accurate to at least 2 decimal places. In our opinion, this is already an impressive accuracy, considering that the calculation of a c_i value is non-trivial, as shown in Fig. 14. The achieved accuracy could also be increased by using a higher iteration depth for the respective procedures, which would, however, increase the calculation times accordingly.

We achieved slightly worse accuracy when homomorphically selecting the λ value with the lowest c_i value. In the non-homomorphic case, this would have been $\lambda = 0$, but in the homomorphic case we calculated $\lambda = 0.1382$. However, we could have increased the accuracy at this point again by increasing the iteration depths of the respective underlying methods. However, we decided against such an increase, since on the one hand this would have increased the computation time accordingly and on the other hand, with this homomorphically calculated λ value, we can, in our opinion, achieve a fairly accurate overall approximation of the Box–Cox transformation. To show this, we consider Fig. 16, which illustrates the homomorphically calculated Box–Cox transformation with $\lambda = 0.1382$ and the non-homomorphically calculated Box–Cox transformation. In this figure, the mean deviation of the homomorphic variant from the non-homomorphic variant is $(2.22 \pm 1.74)\%$.

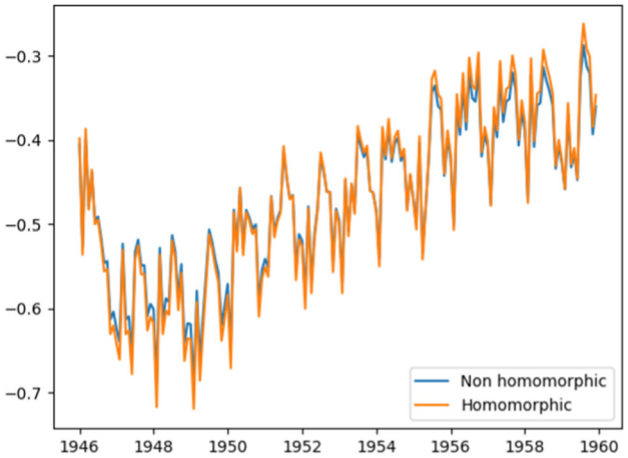

Fig. 16 Comparison of the non-homomorphically and homomorphically calculated Box–Cox transformation

7 Conclusion

Although cloud computing has proven helpful for managing large amounts of data, privacy and data security concerns remain an issue, especially with sensitive data such as medical records. To benefit from the features of cloud computing without the provider having access to the data, homomorphic encryption is an approach that allows the user to store and process data securely in the cloud. However, homomorphic encryption libraries only support addition and multiplication; other mathematical functions must be implemented by the user. To this end, we implemented and investigated basic mathematical functions, such as division, exponential, square root, logarithm, minimum, and maximum, using the CKKS cryptosystem implementation of the OpenFHE library. We then evaluated their performance in terms of accuracy and the computation time required to achieve it. Our results show how the number of iterations required to achieve a given accuracy varies depending on the function. To demonstrate that our implementations can also be used for more complex computations, we used them to implement the Box–Cox transformation in a homomorphic setting. This transformation is used in many real-world applications, such as time-series forecasting.

While homomorphic encryption is still relatively slow, it is a promising solution for preserving data privacy and security. Especially since we see potential to accelerate these computations, for example, by performing them on GPUs instead of CPUs, as is common in machine learning, or by developing special hardware for this purpose, as is common in cryptography. To this end, our future work will focus on implementing additional workflow tasks from the time-series domain and exploring homomorphic neural networks. Our ultimate goal is to create a user-friendly open-source tool that incorporates various mathematical functions and requires minimal

knowledge of homomorphic encryption. Users should be able to easily apply homomorphic computing, such as computing a root homomorphically by creating the corresponding object `rootObject = root(lower interval limit, upper interval limit, precision)` and calling `rootObject.calculateRoot(x)`. The bounds for the computation should only be set initially, and for each following computation, the bounds should be set automatically. Moreover, we plan to provide guidelines to assist users in selecting appropriate initial bounds. In addition, we plan to implement the activation functions of neural networks for the CKKS cryptosystem using the basic mathematical functions realized in this paper. As soon as we are able to homomorphically calculate activation functions with high accuracy, we plan to string them together into neural networks as a next step.

Funding Open Access funding enabled and organized by Projekt DEAL.

Research data policy and data availability statements All data are either included in the paper or can be found in the sources given.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

Appendix

See Figs. 17 and 18.

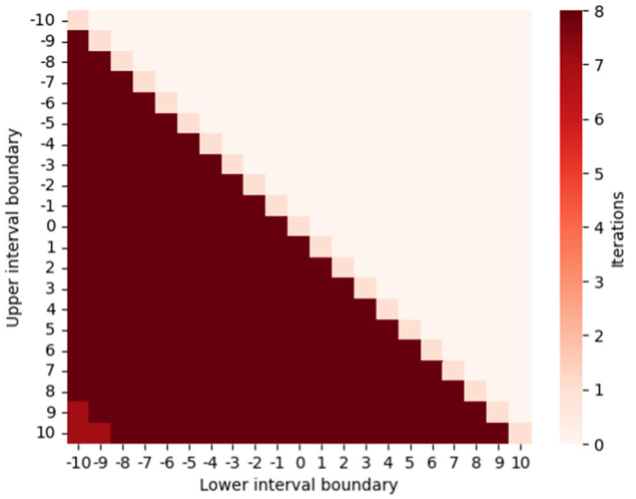

Fig. 17 Visualization of the required iterations to calculate the minimum function for values from different intervals with an accuracy of 0.1

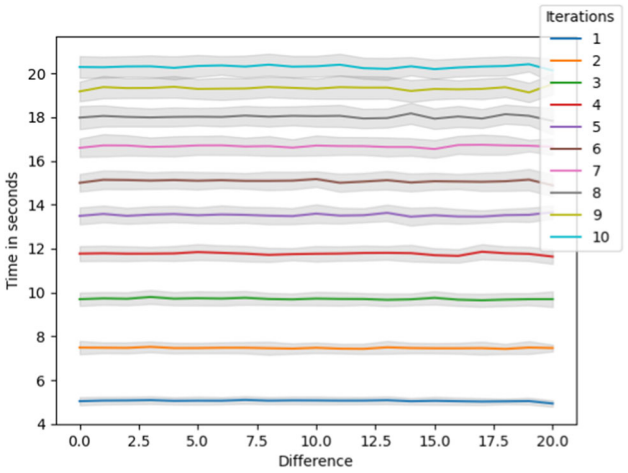

Fig. 18 Required times to determine $min(a, b)$, where $a, b \in [-10, 10]$ and $a \leq b$. The iteration depth of the iterative calculation procedure was varied between 1 and 10. The x-axis represents the amount of the difference of a and b

References

1. Park, J., Han, K., Lee, B.: Green cloud? An empirical analysis of cloud computing and energy efficiency. Manag. Sci. (2022)
2. Handelsblatt: Cloud-Computing in Deutschland: Statistik Zeigt Das Nutzungsprofil Deutscher Unternehmen. (2020). Handelsblatt. Online available under https://www.handelsblatt.com/adv/firmen/cloud-computing-deutschland-statistik.html. Accessed on 22 Jan 2023
3. Rivest, R.L., Adleman, L., Dertouzos, M.L., et al.: On data banks and privacy homomorphisms. Found. Secure Comput. **4**(11), 169–180 (1978)

4. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing, pp. 169–178 (2009)
5. Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic encryption for arithmetic of approximate numbers. In: Advances in Cryptology—ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3–7, 2017, Proceedings, Part I 23, pp. 409–437 (2017). Springer
6. OpenFHE organization: OpenFHE. OpenFHE organization. Online available under https://www.openfhe.org/. Accessed on 17 Jan 2023
7. Box, G.E., Cox, D.R.: An analysis of transformations. J. R. Stat. Soc.: Ser. B (Methodol.) **26**(2), 211–243 (1964)
8. Bauer, A., Züfle, M., Herbst, N., Kounev, S., Curtef, V.: Telescope: An automatic feature extraction and transformation approach for time series forecasting on a level-playing field. In: 2020 IEEE 36th International Conference on Data Engineering (ICDE), pp. 1902–1905 (2020). IEEE
9. Katz, J., Lindell, Y.: Introduction to Modern Cryptography, second edition edn. Chapman Hall, CRC Cryptography and Network Security. CRC Press, Boca Raton ; London ; New York (2015)
10. Gentry, C.: Computing arbitrary functions of encrypted data. Commun. ACM **53**(3), 97–105 (2010). https://doi.org/10.1145/1666420.1666444
11. Naehrig, M., Lauter, K., Vaikuntanathan, V.: Can homomorphic encryption be practical? In: Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop, pp. 113–124 (2011)
12. Okada, H., Cid, C., Hidano, S., Kiyomoto, S.: Linear depth integer-wise homomorphic division. In: IFIP International Conference on Information Security Theory and Practice, pp. 91–106 (2019). Springer
13. Babenko, M., Golimblevskaia, E.: Euclidean division method for the homomorphic scheme ckks. In: 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), pp. 217–220 (2021). IEEE
14. Cetin, G.S., Doroz, Y., Sunar, B., Martin, W.J.: Arithmetic using word-wise homomorphic encryption. Cryptology ePrint Archive (2015)
15. Veugen, T.: Encrypted integer division and secure comparison. Int. J. Appl. Cryptogr. **3**(2), 166–180 (2014)
16. Ugwuoke, C., Erkin, Z., Lagendijk, R.L.: Secure fixed-point division for homomorphically encrypted operands. In: Proceedings of the 13th International Conference on Availability, Reliability and Security, pp. 1–10 (2018)
17. Shortell, T., Shokoufandeh, A.: Secure signal processing using fully homomorphic encryption. In: International Conference on Advanced Concepts for Intelligent Vision Systems, pp. 93–104 (2015). Springer
18. Rahulamathavan, Y.: Privacy-preserving similarity calculation of speaker features using fully homomorphic encryption. arXiv preprint arXiv:2202.07994 (2022)
19. Qu, H., Xu, G.: Improvements of homomorphic evaluation of inverse square root. Available at SSRN 4258571
20. Panda, S.: Principal component analysis using CKKS homomorphic scheme. In: International Symposium on Cyber Security Cryptography and Machine Learning, pp. 52–70 (2021). Springer
21. Panda, S.: Polynomial approximation of inverse sqrt function for fhe. Cryptology ePrint Archive (2022)
22. Gusani, S.: Efficient implementation of homomorphic encryption and its application. PhD thesis (2015). https://doi.org/10.13140/RG.2.2.14049.92007
23. Khanna, S., Rafferty, C.: Accelerating homomorphic encryption using approximate computing techniques. In: ICETE (2), pp. 380–387 (2020)
24. Cheon, J.H., Han, K., Kim, A., Kim, M., Song, Y.: A full RNS variant of approximate homomorphic encryption. In: International Conference on Selected Areas in Cryptography, pp. 347–368 (2019). Springer
25. Li, B., Micciancio, D.: On the security of homomorphic encryption on approximate numbers. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 648–677 (2021). Springer
26. Abramowitz, M., Stegun, I.A.: Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables vol. 55. US Government printing office, Washington, DC, 20402 (1964)
27. Boura, C., Gama, N., Georgieva, M.: Chimera: a unified framework for b/fv, tfhe and heaan fully homomorphic encryption and predictions for deep learning. IACR Cryptol. ePrint Arch. **2018**, 758 (2018)
28. Bourse, F., Minelli, M., Minihold, M., Paillier, P.: Fast homomorphic evaluation of deep discretized neural networks. In: Annual International Cryptology Conference, pp. 483–512 (2018). Springer
29. Chatterjee, A., Sengupta, I.: Sorting of fully homomorphic encrypted cloud data: can partitioning be effective? IEEE Trans. Serv. Comput. **13**(3), 545–558 (2017)
30. Cheon, J.H., Kim, M., Kim, M.: Search-and-compute on encrypted data. In: International Conference on Financial Cryptography and Data Security, pp. 142–159 (2015). Springer
31. Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster packed homomorphic operations and efficient circuit bootstrapping for tfhe. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 377–408 (2017). Springer
32. Crawford, J.L., Gentry, C., Halevi, S., Platt, D., Shoup, V.: Doing real work with fhe: the case of logistic regression. In: Proceedings of the 6th Workshop on Encrypted Computing and Applied Homomorphic Cryptography, pp. 1–12 (2018)
33. Emmadi, N., Gauravaram, P., Narumanchi, H., Syed, H.: Updates on sorting of fully homomorphic encrypted data. In: 2015 International Conference on Cloud Computing Research and Innovation (ICCCRI), pp. 19–24 (2015). IEEE
34. Kocabas, O., Soyata, T.: Utilizing homomorphic encryption to implement secure and private medical cloud computing. In: 2015 IEEE 8th International Conference on Cloud Computing, pp. 540–547 (2015). IEEE
35. Togan, M., Morogan, L., Plesca, C.: Comparison-based applications for fully homomorphic encrypted data. Proc. Roman. Acad.-Ser. A: Math. Phys. Tech. Sci. Inf. Sci. **16**, 329 (2015)
36. Cheon, J.H., Kim, D., Kim, D., Lee, H.H., Lee, K.: Numerical method for comparison on homomorphically encrypted numbers. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 415–445 (2019). Springer
37. Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., et al.: Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. IEEE Access **10**, 30039–30054 (2022)
38. Al Badawi, A., Jin, C., Lin, J., Mun, C.F., Jie, S.J., Tan, B.H.M., Nan, X., Aung, K.M.M., Chandrasekhar, V.R.: Towards the alexnet moment for homomorphic encryption: HCNN, the first homomorphic CNN on encrypted data with gpus. IEEE Trans. Emerg. Top. Comput. **9**(3), 1330–1343 (2020)
39. Xie, P., Bilenko, M., Finley, T., Gilad-Bachrach, R., Lauter, K., Naehrig, M.: Crypto-nets: neural networks over encrypted data. arXiv preprint arXiv:1412.6181 (2014)
40. Bhat, R., Sunitha, N.R., Iyengar, S.S.: A probabilistic public key encryption switching scheme for secure cloud storage. Int. J. Inf. Technol. **15**(2), 675–690 (2023)

41. Obermann, S.F., Flynn, M.J.: Division algorithms and implementations. IEEE Trans. Comput. **46**(8), 833–854 (1997)
42. Markstein, P.: Software division and square root using gold schmidt's algorithms. In: Proceedings of the 6th Conference on Real Numbers and Computers (RNC'6), vol. 123, pp. 146–157 (2004)
43. Rodeheffer, T.: Software integer division. Technická Zpráva MSR-TR-2008-141, Microsoft Research (2008)
44. Karp, A.H., Markstein, P.: High-precision division and square root. ACM Trans. Math. Softw. **23**(4), 561–589 (1997)
45. codebrowser: E_expf.c Source Code [glibc/sysdeps/ieee754/flt-32/e_expf.c] - Codebrowser. (2022). codebrowser. Online available under https://codebrowser.dev/glibc/glibc/sysdeps/ieee754/flt-32/e_expf.c.html. Accessed on 09 Dec 2022
46. Schraudolph, N.N.: A fast, compact approximation of the exponential function. Neural Comput. **11**(4), 853–862 (1999)
47. Malík, P.: High throughput floating point exponential function implemented in fpga. In: 2015 IEEE Computer Society Annual Symposium on VLSI, pp. 97–100 (2015). https://doi.org/10.1109/ISVLSI.2015.61
48. Nilsson, P., Shaik, A.U.R., Gangarajaiah, R., Hertz, E.: Hardware implementation of the exponential function using Taylor series. In: 2014 NORCHIP, pp. 1–4 (2014). https://doi.org/10.1109/NORCHIP.2014.7004740
49. Dinechin, F.d., Pasca, B.: Floating-point exponential functions for dsp-enabled fpgas. In: 2010 International Conference on Field-Programmable Technology, pp. 110–117 (2010). https://doi.org/10.1109/FPT.2010.5681764
50. Zaninetti, L.: Padé approximant and minimax rational approximation in standard cosmology. Galaxies **4**(1) (2016). https://doi.org/10.3390/galaxies4010004
51. Bojdi, Z.K., Ahmadi-Asl, S., Aminataei, A.: A new extended pade approximation and its application. Adv. Numer. Anal. (2013)
52. Gupta, A., Gopakumar, A., Iyer, B.R., Iyer, S.: Padé approximants for truncated post-newtonian neutron star models. Phys. Rev. D **62**(4), 044038 (2000)
53. Wolfram: Padé Approximant. (2022). Wolfram. Online available under https://mathworld.wolfram.com/PadeApproximant.html, Accessed on 12 Dec 2022
54. codebrowser: E_sqrt.c Source Code [glibc/sysdeps/ieee754/dbl-64/e_sqrt.c] - Codebrowser. (2022). codebrowser. Online available under https://codebrowser.dev/glibc/glibc/sysdeps/ieee754/dbl-64/e_sqrt.c.html#__ieee754_sqrt, Accessed on 13 Dec 2022
55. Lawrence University: Newton's Method. (2011). Lawrence University. Online available under http://www2.lawrence.edu/fast/GREGGJ/Math420/Sections_2_3_to_2_5.pdf, Accessed on 15 Dec 2022
56. Steihaug, T., Rogers, D.: Approximating cube roots of integers, after heron's metrica iii. 20. arXiv preprint arXiv:1905.03547 (2019)
57. Kosheleva, O.: Babylonian method of computing the square root: justifications based on fuzzy techniques and on computational complexity. In: NAFIPS 2009–2009 Annual Meeting of the North American Fuzzy Information Processing Society, pp. 1–6 (2009). https://doi.org/10.1109/NAFIPS.2009.5156463
58. Cheon, J.H., Kim, D., Kim, D., Lee, H.H., Lee, K.: Numerical method for comparison on homomorphically encrypted numbers. In: Galbraith, S.D., Moriai, S. (eds.) Advances in Cryptology - ASIACRYPT 2019, pp. 415–445. Springer, Cham (2019)
59. Halley, E.: Methodus nova accurata & facilis inveniendi radices æqnationum quarumcumque generaliter, sine praviæ reductione. Philos. Trans. R. Soc. Lond. **18**(210), 136–148 (1707)
60. Muller, J.-M., Muller, J.-M.: Elementary Functions. Springer, Spring Street, New York, NY 100013 (2006)
61. Hart, J.F.: Computer Approximations. Krieger Publishing Co., Inc., 1725 Krieger Lane, Malabar, Florida, 32950 (1978)
62. codebrowser: E_log.c Source Code [glibc/sysdeps/ieee754/dbl-64/e_log.c] - Codebrowser. (2022). codebrowser. Online available under https://codebrowser.dev/glibc/glibc/sysdeps/ieee754/dbl-64/e_log.c.html. Accessed on 13 Dec 2022
63. Thompson, I.: NIST Handbook of Mathematical Functions, edited by Frank WJ Olver, Daniel W. Ronald F. Boisvert, Charles W. Clark. Taylor & Francis, Lozier (2011)
64. Kornerup, P., Muller, J.-M.: Choosing starting values for Newton–Raphson computation of reciprocals, square-roots and square-root reciprocals. PhD thesis, INRIA, LIP (2003)
65. Montuschi, P., Mezzalama, M.: Optimal absolute error starting values for Newton–Raphson calculation of square root. Computing **46**(1), 67–86 (1991)
66. Guerrero, V.M.: Time-series analysis supported by power transformations. J. Forecast. **12**(1), 37–48 (1993)
67. DescTools: Automatic Selection of Box Cox Transformation Parameter. DescTools. Online available under https://search.r-project.org/CRAN/refmans/DescTools/html/BoxCoxLambda.html. Accessed on 17 Jan 2023
68. Monash University, Clayton, Australia: Guerrero's Method for Box Cox Lambda Selection. Monash University, Clayton, Australia. Online available under https://github.com/tidyverts/feasts/blob/master/R/guerrero.R. Accessed on 21 March 2023
69. Rob J Hyndman: Nybirths.dat. Rob J Hyndman. Online available under https://robjhyndman.com/tsdldata/data/nybirths.dat, Accessed on 21 March 2023
70. GeeksforGeeks: Python | Box–Cox Transformation. (2022). GeeksforGeeks. Online available under https://www.geeksforgeeks.org/box-cox-transformation-using-python/. Accessed on 9 March 2023