Novel approximations of elementary functions in zero-knowledge proofs*

Kaarel August Kurik
Cybernetica AS
Tartu, Estonia
kaarel.august.kurik@cyber.ee

Peeter Laud
Cybernetica AS
Tartu, Estonia
peeter.laud@cyber.ee

ABSTRACT

In this paper, we study the computation of complex mathematical functions in statements executed on top of zero-knowledge proofs (ZKP); these functions may include roots, exponentials and logarithms, trigonometry etc. While existing approaches to these functions in privacy-preserving computations (and sometimes also in general-purpose processors) have relied on polynomial approximation, more powerful methods are available for ZKP. In this paper, we note that in ZKP, all *algebraic functions* are exactly computable. Recognizing that, we proceed to the approximation of transcendental functions with algebraic functions. We develop methods of approximation, instantiate them on a number of common transcendental functions, and benchmark their precision and efficiency in comparison with best polynomial approximations.

1 INTRODUCTION

Zero-knowledge proofs (ZKP) [19] are a cryptographic technique that allow Prover to convince Verifier that a certain statement holds, without disclosing *why* it holds. They are used in the construction of other cryptographic primitives, including signatures [5] or secure multiparty computation protocols [18], but as the technology matures, they are expected to find use, including independent use, in a larger variety of applications. Indeed, ZKP already have blockchain-based applications in privacy-preserving validation of the transactions [6]. As part of their "business logic", the transaction-validating statements mainly use integer arithmetic and comparisons as their computational subroutines. But with emerging or future applications of ZKP, we can expect many more kinds of computational operations to be relevant.

Privacy-preserving machine learning (PPML) [31] is expected to become a significant application area for ZKP, where Prover would convince the Verifier that the result of executing the model corresponds to the inputs of the model, while not revealing the inputs or any intermediate computations. If the model is a neural network, then this involves the execution of various network layers under ZKP, including the non-linear *activation functions.* These activation functions are often transcendental, having been built from exponential functions.

ZKP has been proposed as a tool to verify the claimed mileage of a vehicle in a certain territory in privacy-preserving manner [2]. This task requires the computation of the length of the vehicle trajectory from the sequence of coordinates the vehicles has been found on. Computing the lengths of lines requires the computation of square roots (if the lengths of several segments have to be added up), or, in case of larger areas where the curvature of Earth has to be taken into account, the computation of trigonometric functions. As the input to the statment proven on top of ZKP is the list of coordinates, the computation of these functions has to be happen on top of ZKP, too.

ZKP has also been proposed to assist in court cases involving copyright claims [26]. In these cases, the information content of one document with respect to another one has to be computed on top of ZKP. The definitions of these entropy- and Kolmogorov complexity-related notions often include logarithms.

We can imagine further possible cases, where complex mathematical functions have to be computed on top of ZKP. E.g. one may want to prove properties of hypothetical physical systems: prove that it is possible to build a system within the given constraints. Similarly, one may want to prove the existence of some financial set-up satisfying given constraints. In both cases, one may expect that exponentials have to be evaluated.

In this paper, we study how various non-polynomial functions may be evaluated or approximated in computations running on top of some ZKP protocol. We see that algebraic functions (e.g. square root) may be computed "exactly", i.e. up to the precision limit in representing fractional numbers (indeed, all the examples above require fractional numbers to be represented in some form). We then study methods of approximating transcendental functions. While these approximations have previously been based on piecewise polynomials, either evaluating them directly or optimizing their evaluation based on the shape of their coefficients, we are able to base our approximations on arbitrary algebraic functions. However, we find that there exist almost no methods for this approximation, i.e. for the derivation of the coefficients of the approximating function based on the approximated function and the upper bounds on the degrees of variables. Hence we use ad hoc adaptations of general optimization methods to come up with the coefficients. Comparing the computation costs and precision, we see that the approximating algebraic functions found by our method still beat the best polynomial approximations.

2 RELATED WORK

Numeric computations under zero-knowledge have been considered in the context of privacy-preserving execution of neural networks. Weng et al. [29] describe a system for this purpose; they report having implemented the representations of fractional numbers as fixed- or floating-point values, and transcendental functions (sigmoid, SoftMax) operating on them. The values are encoded as vectors of bits, and the implementations of functions conform

*Distribution Statement 'A' (Approved for Public Release, Distribution Unlimited). This research has been funded by the Defense Advanced Research Projects Agency (DARPA) under contract HR0011-20-C-0083. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

to the IEEE-754 standard. Other ZK neural network implementations (e.g. [24, 17]) only support piecewise polynomial activation functions, for example ReLU: $x \mapsto (x + |x|)/2$.

Angel et al. [1] consider zero-knowledge certificates of optimality for the solutions of numeric optimization problems. This is an interesting example of applying the compute-and-check paradigm (see Sec. 3.1 below) to numeric computations, making use of the primal-dual structure of such optimization problems. In this paper, we use the same paradigm to evaluate algebraic functions.

There exists a significant body of work for representing fractional numbers and approximating algebraic and transcendental functions in secure multiparty computations. Catrina et al. [12, 13] were perhaps the first to systematically represent fixed-point numbers in secret-sharing based secure MPC, and give protocols that perform arithmetic operations (including division) with them. Krips and Willemson [22] studied the combination of fixed- and floating-point operations in order to obtain the best performance for the latter, and also [23] proposed a Monte-Carlo like method for evaluating inverses of polynomial functions (e.g. square root). Kamm and Willemson [21] investigated the use of piecewise polynomials for the approximation of algebraic and transcendental functions. The polynomials were picked as *Chebyshev interpolations* [28, Chap. 2] of the functions being approximated. Dimitrov et al. [16] investigated the representation of fractional numbers as elements of a quadratic field, and implemented the arithmetic operations working with them. More recently, Catrina [11] has evaluated the performance of various protocols for evaluating polynomials over representations of fixed- and floating-point numbers.

Low-degree algebraic approximations of trigonometric and hyperbolic functions have been considered [3, 30] for obtaining analytical approximate solutions to transcendental equations from theoretical physics (quantum mechanics, electromagnetism, ealsticity), such that a useful explicit description of the solution in terms of the parameters of the equation is preserved. Finally, let us mention that a *rational* approximation of the sine function has been known since at least the 7th century [20].

3 COMPUTATION IN STATEMENTS PROVED IN ZK

In this paper, we consider computations via arithmetic circuits over a field $\mathbb{Z}_N$ for a sufficiently large prime N. The instance and the witness are given as inputs to that circuit (both of them occupying multiple input wires); the computation is deemed to *accept* if all outputs of the circuit are 0. The operations supported by the circuit are binary addition and multiplication, as well as unary addition and multiplication with constants.

The arithmetic circuit will serve as one of the inputs to a cryptographic protocol that executes this circuit in zero-knowledge manner. Our results do not depend on the choice of this protocol, although the support for some extra features may improve the size of the circuit.

3.1 Compute and check

An ubiquitous paradigm in the design of statements proved under ZK is *compute-and-check*. This paradigm applies to sub-computations that are inefficient or perhaps even impossible to express using only additions and multiplications, but whose outcome is simple and efficient to verify as correct with the help of these operations; perhaps with some extra evidence on the side. A folk example is *division* modulo N: instead of computing $z \leftarrow x \cdot y^{-1} \pmod N$ in the circuit, we let Prover to provide an extra input z to the circuit. The circuit will compute *and output* $y \cdot z - x$, i.e. the computation accepts only if z is indeed the ratio between x and y. We say that the circuit *verifies that* $x = y \cdot z$.

Another folk example is *bit extraction*: given $x \in \mathbb{Z}_N$, find the values $x_0, \ldots, x_{k-1} \in \{0, 1\} \subset \mathbb{Z}_N$ (for a suitable k) so, that $x = \sum_{i=0}^{k-1} 2^i \cdot x_i$. Using the compute-and-check paradigm, Prover will give $x_0, \ldots, x_{k-1}$ as extra inputs to the circuit, which then verifies that x is equal to their linear combination with powers of two, and each x_i satisfies $x_i = x_i^2$. Given that N is a prime number (or even just prime power), the latter implies that x_i is a bit. Bit extraction is often used to compare two numbers: in order to evaluate whether $x < y$, we compute the bitwise representations of both, and then execute a sub-circuit for comparison.

Third example of compute-and-check is making sure whether a value is 0: given $x \in \mathbb{Z}_N$, we want to find $b \in \{0, 1\} \subset \mathbb{Z}_N$, such that $b = 0$ iff $x = 0$. While we could extract the bits of x and then run a circuit for k-wise OR, or compute $b \leftarrow x^{N-1}$, there exists a more efficient method [27]. Namely, Prover will give b as an extra input to the circuit, and also a value $w \in \mathbb{Z}_N$, satisfying $w^{-1} \cdot x = 1$ if $x \neq 0$. Circuit verifies that $x \cdot b = x$ and $w \cdot x \cdot b = b$. Here we see an example of providing also some extra evidence besides the result of the computation.

3.2 Fractional numbers in ZK

The arguments and values of the functions that we evaluate are elements of the set of real numbers $\mathbb{R}$. These values have to be represented by (one or more) elements of $\mathbb{Z}_N$. In this paper, we consider *fixed point* representation, where an element $a \in \mathbb{Z}_N$, interpreted as an integer between $-\lfloor N/2 \rfloor$ and $\lfloor N/2 \rfloor$, corresponds to the number $a/2^{\mathsf{pp}} \in \mathbb{R}$ for some $\mathsf{pp} \in \mathbb{N}$. Any real number $x \in [-N/2^{\mathsf{pp}+1}, N/2^{\mathsf{pp}+1}]$ is thus represented by an element of $\mathbb{Z}_N$ that is closest to $x \cdot 2^{\mathsf{pp}}$.

The addition of fixed point representations, and the multiplication of them with constant integers is straightforward, corresponding to the same operations in $\mathbb{Z}_N$. The *multiplication* of fixed point representations is slightly more complex, requiring the result to be rescaled. Compute-and-check is helpful here: given two representations $a, b \in \mathbb{Z}_N$, we are looking for $c \in N$, such that $a \cdot b \approx c \cdot 2^{\mathsf{pp}}$. If we round down (i.e. towards $-\infty$), then Prover can simply provide c as an additional input to the circuit, which checks that $a \cdot b \geq c \cdot 2^{\mathsf{pp}}$ and $a \cdot b < (c + 1) \cdot 2^{\mathsf{pp}}$. These two checks are equivalent to $T := a \cdot b - c \cdot 2^{\mathsf{pp}} \in [0, 2^{\mathsf{pp}} - 1]$, which amounts to the check that the extraction of T into pp bits is successful.

These computations show another important limit on the sizes of values that can be represented: the results of the multiplications $a \cdot b$ and $c \cdot 2^{\mathsf{pp}}$ must still be integers between $-\lfloor N/2 \rfloor$ and $\lfloor N/2 \rfloor$; rollovers modulo N invalidate their soundness. Hence we require all these values to be between $-\sqrt{N/2}$ and $\sqrt{N/2}$. Checking that the values actually fall into this range is application-specific: it may be necessary to introduce explicit checks; it may also turn out that due to the nature of computations, all (or some) intermediate values

```
Input: End-points $l, u \in \mathbb{R}$ of search interval with $l \leq u$
Input: Invariant $Inv : \mathbb{R} \times \mathbb{R} \to$ Bool
Assumption: $Inv(l, u) \wedge \forall d \in [l, u] : (Inv(l, d) \vee Inv(d, u))$
Input: Termination condition $TC : \mathbb{R} \times \mathbb{R} \to$ Bool
Output: $a, b \in \mathbb{R}$, s.t. $l \leq a \leq b \leq u$ and $TC(a, b)$
1 $a, b \leftarrow l, u$;
2 while $\neg TC(a, b)$ do
3     $d \leftarrow (a + b)/2$;
4     if $Inv(a, d)$ then $b := d$ else $a := d$;
5 return $a, b$
```

Algorithm 1: Binary search: BinarySearch

are guaranteed to be sufficiently small. For example, the size of the output of the ReLU function is never bigger than the size of the input.

Division of fixed point representations is almost as simple as multiplication: given $a, c \in \mathbb{Z}_N$, we are looking for $b \in \mathbb{Z}_N$, such that we again have $a \cdot b \approx c \cdot 2^{\mathsf{pp}}$. With Prover providing b as an extra input to the circuit, we verify that $T := c \cdot 2^{\mathsf{pp}} - a \cdot b$ has the same sign as a, and $|T| < |a|$. Interestingly, the *square root* can be similarly computed: given a fixed point representation $c \in \mathbb{Z}_N$ of some number (assuming $c \geq 0$), the representation of its square root is an element $a \in \mathbb{Z}_N$ that is non-negative and satisfies $0 \leq c \cdot 2^{\mathsf{pp}} - a^2 < 2a + 1$. Given a, these checks can be performed by the circuit.

The technique of evaluating square roots may be extended to the evaluation of arbitrary *algebraic functions*. Recall the definition:

Definition 3.1. A continuous function $y : I \to \mathbb{R}$ (where I is an interval in $\mathbb{R}$) is an *algebraic function* if there is some $P \in \mathbb{R}[X, Y]$ such that $P \neq 0$ and $P(x, y(x)) = 0$ for all $x \in I$. For such y and P, we say that *y is carved by P*.

Here $\mathbb{R}[X, Y]$ is the set of two-variable (denoted X and Y) polynomials with coefficients in $\mathbb{R}$. Multiple distinct definitions of the term "algebraic function" are found in the literature, commonly omitting the requirement of continuity, requiring P to be irreducible, or allowing the "function" to be an arbitrary element of a suitable field extension. [15, p. 45] An algebraic function in our sense is a piecewise algebraic function in the continuous irreducible sense.

The evaluation of $P(x, y)$ in compute-and-check fashion is given in Alg. 3, with subroutines in Alg. 1 and Alg. 2. In these algorithms, we are introducing a convention for denoting the values and operations in compute-and-check procedures, similarly to the existing conventions for privacy-preserving computations, where it is typical to denote private values (i.e. values managed by the cryptographic protocol for preserving the privacy during the computation) by putting them in (single or) double square (or angle) brackets. In our convention, the values managed by the ZK protocol are put in double square brackets. The visibility of values is indicated by colors, with red being visible to Prover only, green being visible to both Prover and Verifier at the time of computation, blue being constants available at the time of preparing the computation, and black denoting either basic constants or unknown (or irrelevant) visibility.

Alg. 3 (lines 1–4) shows that given x, Prover first finds y, such that either $P(x, y) = 0$, or $P(x, y)$ and $P(x, y + 2^{-\mathsf{pp}})$ have opposite

```
Input: Degree $d \in \mathbb{N}$, coefficients $[\![\vec{c}]\!] \in \mathbb{R}^{d+1}$ of $P \in \mathbb{R}[X]$
Input: Argument $[\![x]\!] \in \mathbb{R}$
Output: The value $[\![y]\!]$, where $y = P(x)$
1 $[\![y]\!] \leftarrow [\![c_{d+1}]\!]$;
2 for $i = 1$ to $d$ do $[\![y]\!] := [\![x]\!] \cdot [\![y]\!] + [\![c_{d-i+1}]\!]$ ;
3 return $[\![y]\!]$
```

Algorithm 2: Evaluating a one-variable polynomial: EvalP

```
Input: Degrees $\mathsf{xd}, \mathsf{yd} \in \mathbb{N}$ of $P \in \mathbb{R}[X, Y]$
Input: Coefficients $\mathbf{A} \in \mathbb{R}^{(\mathsf{xd}+1)\times(\mathsf{yd}+1)}$ of $P$
Input: Argument $[\![x]\!] \in \mathbb{R}$
Input: End-points $l, u \in \mathbb{R}$ of search interval with $l \leq u$
Assumption: $P(x, l) \cdot P(x, u) \leq 0$
Input: Precision (number of fractional bits) $\mathsf{pp} \in \mathbb{N}$
Output: $[\![y]\!]$, s.t. $l \leq y \leq u$ and $P(x, y) \approx 0$
1 $Inv \leftarrow (s, t) \mapsto P(x, s) \cdot P(x, t) \leq 0$;
2 $TC \leftarrow (s, t) \mapsto P(x, s) = 0 \vee P(x, t) = 0 \vee |t - s| \leq 2^{-\mathsf{pp}}$;
3 $a, b \leftarrow \mathsf{BinarySearch}(l, u, Inv, TC)$;
4 if $P(x, a) = 0$ then $y \leftarrow a$ else $y \leftarrow b - 2^{-\mathsf{pp}}$;
5 $[\![y]\!] \leftarrow \mathsf{wire}(y)$;
6 $[\![x_0]\!], [\![x_1]\!] \leftarrow 1, [\![x]\!]$;
7 for $i = 2$ to $\mathsf{xd}$ do $[\![x_i]\!] \leftarrow [\![x]\!] \cdot [\![x_{i-1}]\!]$;
8 for $i = 0$ to $\mathsf{yd}$ do $[\![c_i]\!] \leftarrow \sum_{j=0}^{\mathsf{xd}} \mathbf{A}_{j,i} \cdot [\![x_j]\!]$;
9 $[\![z]\!] \leftarrow \mathsf{EvalP}(\mathsf{yd}, [\![\vec{c}]\!], [\![y]\!])$;
10 $[\![z']\!] \leftarrow \mathsf{EvalP}(\mathsf{yd}, [\![\vec{c}]\!], [\![y]\!] + [\![2^{-\mathsf{pp}}]\!])$;
11 assert$([\![z]\!] \cdot [\![z']\!] \leq 0)$;
12 return $[\![y]\!]$
```

Algorithm 3: Compute-and-check for an algebraic function

signs, where pp gives the desired precision of the result. Alg. 3 proposes a particular computation for Prover to find such y, but any other method for finding an approximate root of the polynomial $P(x, \cdot)$ may be used. Next (line 5), Prover lets y be another input to the circuit. Alg. 3 continues by checking that y is a good output. In lines 6–7, we compute the powers of x, and in line 8, the coefficients of the polynomial $P(x, \cdot)$. We evaluate (lines 9–10) $P(x, \cdot)$ at points y and $y + 2^{-\mathsf{pp}}$, and verify (line 11) that the results do not have the same sign. Algorithm for polynomial evaluation using the Horner scheme is given in Alg. 2.

In Alg. 2 and 3, the values $[\![x]\!] \in \mathbb{R}$ on the circuit are meant to be represented as fixed-point numbers, presumably with pp binary digits after the point. In this representation, the costly operations are multiplications and linear combinations of fixed-point numbers (even with public constants), requiring a range check. We perform $(\mathsf{xd} - 1)$ such operations in line 7, and $(\mathsf{yd} + 1)$ such operations in line 7 of Alg. 3. We also perform yd such operations in each of the two calls to Alg. 2. The multiplication in line 11 of Alg. 3 does not require a range check, because its outcome is not used in further computations (although the sign check may have similar costs).

In comparison, evaluating a polynomial with fixed-point coefficients and argument requires d costly operations (multiplications of fixed-point numbers), where d is the degree of the polynomial.

Hence, when we want to meaningfully compare the cost of evaluating a polynomial vs. the cost of evaluating an algebraic function, we assign d as to the polynomial, and xd + 3yd to the algebraic function.

4 APPROXIMATIONS OF TRANSCENDENTAL FUNCTIONS

The functions that we may want to compute, but are unable to compute exactly, have to be approximated. Established approximation theory provides a number of results regarding the approximation of continuous functions by polynomial or rational approximations. Among these are theorems characterising optimal approximations, algorithms for finding optimal approximations, efficiently calculable alternatives to optimal approximations, etc. As we saw in the last section, we are able to exactly compute a larger class of functions, thus we study the ways to approximate transcendental functions with algebraic functions.

A function $f : \mathbb{R} \to \mathbb{R}$ is always approximated in some *interval* $I \subset \mathbb{R}$. Given a function f and an approximation y of f, we will consider the infinity norm $\|f - y\| := \sup_{x \in I} |f(x) - y(x)|$ as the measure of approximation error which we desire to minimize.

Definition 4.1. A function $y \in C$ is an *optimal approximation to f in the class C* of functions if for all functions $z \in C$, we have $\|y - f\| \leq \|z - f\|$.

In the rest of the paper, C is typically either the class of algebraic functions of total degree (the largest sum of powers over all the mononomials occurring in P) at most d, or the class of algebraic functions of degree at most xd in X and at most yd in Y.

One concept that arises in the characterisation of optimal approximations is *equioscillation.*

Definition 4.2. A function $f : I \to \mathbb{R}$ is said to *equioscillate n times about the function $g : I \to \mathbb{R}$* if there exist n points $x_1 < \cdots < x_n$ in I and $\sigma \in \{-1, 1\}$ such that $f(x_i) - g(x_i) = \sigma(-1)^i \|f - g\|_\infty$ for all $i \in \{1, \cdots, n\}$.

Theorems exist for polynomial and rational approximations stating that being an optimal approximation with given degree bounds is equivalent to equioscillating a minimum number of times about the approximant, where the number of equioscillations depends only on the degree bounds and not on the approximant. We investigate analogous statements for algebraic approximations and find results suggesting, but not wholly confirming, that no such equivalence holds for algebraic functions.

A notable early result in approximation theory is Chebyshev's equioscillation theorem (attributed to Chebyshev but first systematically handled by others[28, p. 92-93]), which states that a polynomial $p : [a, b] \to \mathbb{R}$ is an optimal approximation to $f : [a, b] \to \mathbb{R}$ among polynomials of degree at most d if and only if p equioscillates about f at least $d + 2$ times.

4.1 Algebraic approximations

Definition 4.3. The algebraic function $y : I \to \mathbb{R}$ is *of degree d* if the smallest total degree of a polynomial $P \in \mathbb{R}[X, Y]$ that carves y is d.

We define $P\colon \operatorname{Poly}(\mathrm{xd}, \mathrm{yd})$ to mean that $P \in \mathbb{R}[X, Y]$ has degree at most xd in X and yd in Y. We define $P\colon \operatorname{Poly}(d)$ to mean that P has total degree *at most d*.

Analogously, we denote the fact that y is carved by a polynomial $P\colon \operatorname{Poly}(\mathrm{xd}, \mathrm{yd})$ as $y\colon \operatorname{Alg}(\mathrm{xd}, \mathrm{yd})$, and the fact that y is of degree *at most d* by $y\colon \operatorname{Alg}(d)$.

Theorem 4.4. *Given a continuous function $f : I \to \mathbb{R}$ (where I is a closed interval) and a fixed $d \in \mathbb{N}$, f has an optimal approximation in the class* $\operatorname{Alg}(d)$.

Proof. First note that the set of algebraic approximations of degree at most d is nonempty, since it contains the constant functions.

We may therefore choose a sequence $y_i\colon \operatorname{Alg}(d)$ such that the sequence of approximation errors $\|y_i - f\|$ approaches $m := \inf\{\|y - f\| \mid y\colon \operatorname{Alg}(d)\}$.

For each y_i we choose a corresponding $P_i \in \mathbb{R}[x, y]$ such that the 2-norm of the coefficients of P_i is 1. By compactness of the n-ball, we can find a subsequence of P_i converging to a polynomial $P \in \mathbb{R}[X, Y]$. Moreover, any such convergent subsequence converges uniformly over any compact subset of $\mathbb{R}^2$. Going forward we assume *without loss of generality* (WLOG) that the whole sequence P_i converges to P.

Let $S = \{(x, y) \in \mathbb{R}^2 \mid x \in I, |y - f(x)| \leq 2m\}$. We know that the graphs of the functions y_i eventually lie in S. We assume WLOG that they all do.

Since f is continuous and I is closed, we have that S is compact. It's known that the lengths of segments of real algebraic curves passing through a fixed compact set are uniformly bounded by a function of the curve's degree. Fix an upper bound L on the lengths of segments of degree at most d algebraic curves intersecting S.

We define $s_i\colon [0, L_i] \to S$ to be the unit speed parameterization of the graph of y_i such that $\pi_x \circ s_i$ is monotonically increasing. Such a parameterization exists for y_i since the graph of y_i is piecewise smooth with finitely many pieces.

Since $L_i \leq L$ for each $i \in \mathbb{N}$ we may define extensions of s_i in the form $s'_i\colon [0, L] \to S$ which traverse the graph of y_i at unit speed and then stand still at the point $s_i(L_i)$ for the remainder of $[0, L]$.

Since the sequence s'_i consists of 1-Lipschitz functions, then by a corollary of Arzelà–Ascoli, it has a uniformly convergent subsequence. We assume WLOG that s'_i converges to s'.

Since $P_i(s'_i(t)) = 0$ for $t \in [0, L]$, and P_i converges uniformly, we have that $P(s'(t)) = 0$ for $t \in [0, L]$. This means that s' traces out a segment of a zero-curve of P.

We can define the approximation error for a curve $g\colon [0, L] \to S$ as $\epsilon(g) := \sup\{|y - f(x)| \mid \exists t \in [0, L], g(t) = (x, y)\}$. If g traces the graph of a function, then the approximation error of g coincides with that of the function. Moreover, if g is the limit of a sequence of curves, then the error $\epsilon(g)$ is the limit of the errors of the sequence. We thus have that $\epsilon(s') = \lim_{i\to\infty} \epsilon(s'_i) = m$.

If $m = 0$, then the graph of f must coincide with the image of s', implying that f is an algebraic function and is its own optimal approximation. Going forward we assume that $m > 0$.

Since $\pi_x \circ s'_i$ is non-strictly increasing for all s'_i, we have that $\pi_x \circ s'$ is also non-strictly increasing. Note that if $\pi_x \circ s'$ were strictly increasing, then the image of s' would coincide with the graph of an optimal algebraic approximation given by the map $\pi_x(s'(t)) \mapsto \pi_y(s'(t))$.

It thus suffices to show that we may perturb (P, s') to (Q, r) such that $\pi_x \circ r$ is strictly increasing and $\epsilon(r) \leq \epsilon(s')$.

We first note that there are two kinds of obstacles to $\pi_x \circ s'$ being strictly increasing: either s' stands still at a point for some interval, or s' moves along a vertical line $x - k = 0$. The first case can be removed by reparameterization: we can thus assume WLOG that the first case does not occur, and having made this modification, s' is also a simple curve.

The second case can occur for finitely many distinct lines $x - k = 0$, such that $x - k$ is a factor of P for each such line. The main idea for defining (Q, r) is to slightly rotate all of these lines around appropriately chosen points, such that the resulting polynomial Q has a zero set that can be traversed by a curve r, which agrees with s' everywhere except in small neighborhoods of intervals on which s' traverses a vertical line, and for which $\pi_x \circ r$ is a strictly increasing function, from which the well-definedness of the map $\pi_x(r(t)) \mapsto \pi_y(r(t))$ is immediate. The rotations can be chosen so as to ensure that $\epsilon(r) \leq \epsilon(s')$, which guarantees the optimality of the algebraic approximation induced by r.

We examine the construction for a single line $x - k = 0$. Let $[a, b] \subseteq [0, L]$ be the interval over which $\pi_x \circ s' \equiv k$ and WLOG let $u = \pi_y(s'(a)) < \pi_y(s'(b)) = v$ (equality is excluded since s' is a simple curve). The cases $a = 0, b = L$ are handled by removing a terminal segment of the curve, so we may assume $[a, b] \subseteq (0, L)$.

Suppose first that $f(k) \geq v$. Fix some $\delta > 0$ such that $|x - k| \leq \delta \Rightarrow |f(x) - f(k)| \leq \frac{m}{2}$. Choose any $w \in (0, a)$ such that $\pi_x(s'(w)) > k - \delta$ and define $T : \mathbb{R} \to \mathbb{R}$ as the affine function passing through $s'(w) = (x_w, y_w)$ and (k, v). We may interpret T as a clockwise rotation of the line $x - k = 0$ about the point (k, v).

Let t' be the largest value of $t \in [w, a)$ for which $\pi_y(s'(t)) \geq T(\pi_x(s'(t)))$ and let $x' = \pi_x(s'(t'))$. We then define r in the region $[t', b]$ by $\pi_x(r(t)) = \frac{b-t}{b-t'}x' + \frac{t-t'}{b-t'}k$ and $\pi_y(r(t)) = T(\pi_x(r(t)))$. (In other words, as soon as the graph of T overtakes s' for good, r stops following s' and starts moving along T at an even pace until it meets up with s' again at $t = b$.)

Now consider any $x \in [x', k]$ and examine the errors of s', r at x. In case we have that $T(x) \leq f(k) - \frac{m}{2}$, then we have that $\pi_y(s') \leq T(x) = \pi_y(r) \leq f(k) - \frac{m}{2} \leq f(x)$, from which $\left|f(x) - \pi_y(r)\right| \leq \left|f(x) - \pi_y(s')\right| \leq m$ (leaving the curve time parameters implicit). If instead we have that $T(x) > f(k) - \frac{m}{2}$, then $f(x) - m \leq f(k) - \frac{m}{2} < T(x) = \pi_y(r) \leq v \leq f(k) \leq f(x) + \frac{m}{2}$, from which $\left|f(x) - \pi_y(r)\right| \leq m$. We thus have that $\left|f(x) - \pi_y(r)\right| \leq m$ in general at those x where $\pi_y(r)$ disagrees with $\pi_y(s')$, from which $\epsilon(r) \leq \epsilon(s')$.

The cases $f(k) \leq u$ and $u < f(k) < v$ are analogous, with rotations around the points (k, u) and $(k, f(k))$ respectively.

□

It's worth noting that the simplest analogue of Theorem 4.4 is not true for the alternative notion of algebraic functions where the polynomial is required to be irreducible. A simple counterexample is the case $f = |\cdot|, d = 2, I = [-1, 1]$, where the family $(x - y)(x + y) - 10^{-n}(x^2 + y^2 - 1) = 0$ has arbitrarily small error as n goes to infinity, but 0 error is not achievable.

The existence of results connecting equioscillation with optimality of approximation in the case of polynomial and rational approximations raises the question of whether there exist analogous results for algebraic approximations. We find results suggesting that the simplest analogues of equioscillation results cannot hold for algebraic approximations.

Theorem 4.5. *An algebraic function y: $\mathrm{Alg}(d)$ that equioscillates at least $d^2 + 2$ times about a continuous function $f : I \to \mathbb{R}$ is an optimal approximation to f.*

Proof. Assume to the contrary that z was a better approximation. Then $|z - f| < |y - f|$ at all points $x \in I$ where $|y - f|(x) = \|y - f\|$, of which there are at least $d^2 + 2$ such that $y - f$ alternates in sign for every pair of consecutive points. By IVT this implies that y intersects z in at least $d^2 + 1$ points. This is impossible by Bézout's theorem applied to algebraic curves of degree d. □

Theorem 4.6. *For each $d \in \mathbb{N}$ there is a continuous $f : I \to \mathbb{R}$ such that there is a strictly suboptimal degree d approximation to f that equioscillates $d^2 + 1$ times.*

This shows that $d^2 + 2$ is a tight lower bound for how much equioscillation is needed to show optimality in the absence of extra info about f.

Theorem 4.7. *For each $d \in \mathbb{N}$ there is a continuous $f : I \to \mathbb{R}$ with an optimal non-critical degree d approximation y that equioscillates exactly $\frac{d(d+3)}{2}$ times. f can be chosen so that if y is not an optimal approximation, then any optimal approximation has $\frac{(d-1)(d-2)}{2}$ node singularities.*

Conjecture 4.8. *For each $d \in \mathbb{N}$, there is a continuous $f : I \to \mathbb{R}$ with an optimal degree d approximation that equioscillates exactly $\frac{d(d+3)}{2}$ times.*

Theorem 4.9. *The error map is locally quasiconvex at its points of continuity.*

4.2 Approximating concrete functions

We have seen that algebraic approximations are possible. To compare their use in ZK statements with the use of polynomial approximations common in other privacy-preserving computation techniques, we have picked a number of useful functions, approximated them over polynomials and algebraic functions of various degrees, and determined the achievable precision for variously sized fixed-point representations of real numbers.

We have chosen to consider the following transcendental functions:

- the trigonometric functions $\sin(x), \cos(x), \sin\left(\frac{\pi}{2}x\right), \cos\left(\frac{\pi}{2}x\right)$;
- the inverse trigonometric functions arcsin, arccos;
- the exponential functions exp2 (i.e. $x \mapsto 2^x$), exp;
- the logarithms $\log_2$, ln;
- the complementary error function erfc;
  - the function $\frac{1}{64}\log_2(\mathrm{erfc}(8x))$, also referred to as `log2_erfc_scaled` (see discussion below).

Our approach to approximating them splits this set of functions into *primitive* and *composite* functions. Composite functions are implemented through the composition of primitive functions with simple affine transforms (e.g. the exponential function exp is implemented as $\exp(x) := \exp2(\log_2(e)x)$). Primitive functions are implemented as the composition of an algebraic approximation on a subinterval of the domain of the function, composed with some range reduction based on the special properties of the function.

The following functions are primitive, with following range reductions:

- exp2 is implemented as $\exp2(x) = \exp2(\lfloor x \rfloor) \cdot \exp2(\{x\})$, where $\exp2(\lfloor x \rfloor)$ is calculated via integer arithmetic and $\exp2(\{x\})$ via algebraic approximation on [0,1].
- $\sin\left(\frac{\pi}{2}x\right)$ is implemented by approximation on the interval $[0, 1]$. Elsewhere, it is implemented as

$$\sin\left(\frac{\pi}{2}x\right) = \begin{cases} \sin\left(\frac{\pi}{2}\{x\}\right) & (\lfloor x \rfloor \bmod 4) = 0 \\ \sin\left(\frac{\pi}{2}(1 - \{x\})\right) & (\lfloor x \rfloor \bmod 4) = 1 \\ -\sin\left(\frac{\pi}{2}\{x\}\right) & (\lfloor x \rfloor \bmod 4) = 2 \\ -\sin\left(\frac{\pi}{2}(1 - \{x\})\right) & (\lfloor x \rfloor \bmod 4) = 3 \end{cases} .$$

- $\log_2(x)$ is defined as $\log_2(x) = k + \log_2(y)$ where $x = 2^k \cdot y$ and $y \in [1, 2]$, where $\log_2(y)$ is calculated via approximation.
- $\arcsin(x)$ is calculated via approximation on $[0, \frac{\sqrt{2}}{2}]$, via $\arcsin(x) = \frac{\pi}{2} - \arcsin\left(\sqrt{1 - x^2}\right)$ on $[\frac{\sqrt{2}}{2}, 1]$ and via $\arcsin(x) = -\arcsin(-x)$ on $[-1, 0]$.

Also, the function $s(x) := \frac{1}{64}\log_2(\operatorname{erfc}(8x))$ is primitive; we approximate it on the segment $[0, \frac{3}{4}]$. We implement the complementary error function erfc as

$$\operatorname{erfc}(x) = \begin{cases} 0 & x > 6 \\ (\exp2(s(\frac{x}{8})))^{64} & 0 \le x \le 6 \\ 2 - \operatorname{erfc}(-x) & x < 0 \end{cases} .$$

The function $s(x)$ was chosen over erfc to reduce issues with branch isolation (see Sec. 5). Scaling the input by 8 and the output by $\frac{1}{64}$ was motivated by the desire to ensure that the absolute values of x, y over the approximation region would be bounded by 1, to ensure that no overflows occur while evaluating the approximation polynomial.

In the rest of the paper, the functions $\sin\left(\frac{\pi}{2}x\right)$ and $s(x)$ may also be called `sin_quarter` and `log2_erfc_scaled`, respectively.

4.3 Finding approximations

4.3.1 Polynomial approximations. We have used the Sollya tool [14] to compute the polynomial approximations of degree $d \in \{1, \ldots, 35\}$ for each primitive function. The exception is `log2_erfc_scaled`, where we only went up to degree 8 due to Sollya computing its approximations significantly slower compared to the remaining functions. Sollya implements the Remez algorithm [28, Chap. 10] for finding polynomial approximations of a given degree that are arbitrarily close to optimal, and outputs intervals which provably contain the absolute error of the output approximation.

The precision of each polynomial approximation was measured both by the Sollya software tool, which outputs bounds on the true error of the approximation evaluated with exact arithmetic, and by emulating the fixed-point arithmetic of ZK-SecreC (see Sec. 6) in a Julia program at various fixed-point precisions. The fixed-point error was estimated by evaluation on 20000 evenly spaced points on the range reduction domain of each function. The estimations of precision for polynomial approximation are depicted in Fig. 1.

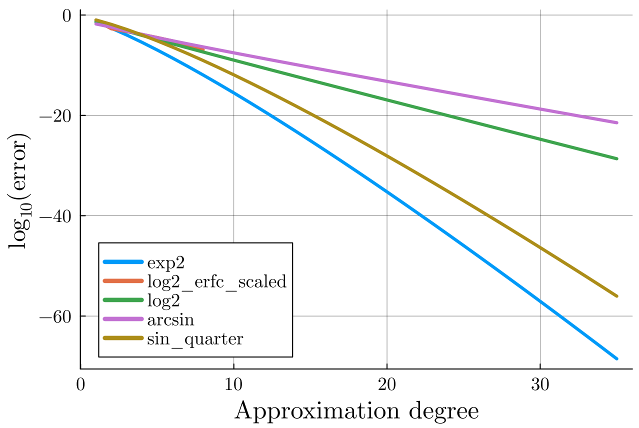

Figure 1: Absolute error of polynomial approximation of given degree computed by Sollya. Domains are the same as the base domains of the respective range reductions.

4.3.2 Algebraic approximations. For each primitive function, we computed algebraic approximations for all pairs of degrees (xd, yd), where xd + 3yd $\le$ 35. Algebraic approximations were computed through a zeroth-order line search procedure [7, Sec. 5.2.1] on the space of polynomials $\mathbb{R}[X, Y]$. The procedure is implemented with *ad hoc* optimizations which are expected to prevent convergence to a local optimum as the number of iterations goes to infinity, but which empirically aid convergence for low iteration counts. The procedure is detailed in Alg. 5, with the subroutine for finding the direction of improvement given in Alg. 4.

The precision of each algebraic approximation was similarly measured by evaluating it on 1600 evenly spaced points on the range reduction domain of each function, using 256-bit floating-point numbers. The results of measurement are depicted in Fig. 2.

The essential reasoning behind the line search procedure proceeds as follows. Let f be a differentiable function that is to be approximated on the interval I, and let $P(c) \in \mathbb{R}[X, Y]$ be a polynomial parameterized by its vector of coefficients $\vec{c}$, with the corresponding algebraic approximation $a(\vec{c}, x)$. We wish to decrease the error $\sup_{x \in I} |a(\vec{c}, x) - f(x)|$ by varying $\vec{c}$. Any decrease in this error must correspond to a decrease in the global maxima of $E_{\vec{c}}(x) := |a(\vec{c}, x) - f(x)|$. If such a decrease is achieved by continuous variation in $\vec{c}$, then the global maxima of the error function $E_{\vec{c}}$ move continuously in the plane with decreasing value in the y-coordinate, and $E_{\vec{c}}$ has vanishing derivative in x at these maxima with the possible exception of the domain boundary. Let some continuously varying global extremum of $E_{\vec{c}}(x) = a(\vec{c}, x) - f(x)$ be given as $(u(\vec{c}), v(\vec{c}))$. Any suitable descent direction $\vec{q}$ should satisfy the constraint $(\vec{q} \cdot \partial_{\vec{c}} v(\vec{c})) \operatorname{sign}(v(\vec{c})) < 0$ to ensure that the $|v(\vec{c})|$ decreases in the direction of $\vec{q}$ — in other words, $\vec{q}$ should be a descent direction relative to the gradients of all the global maxima of the error function.

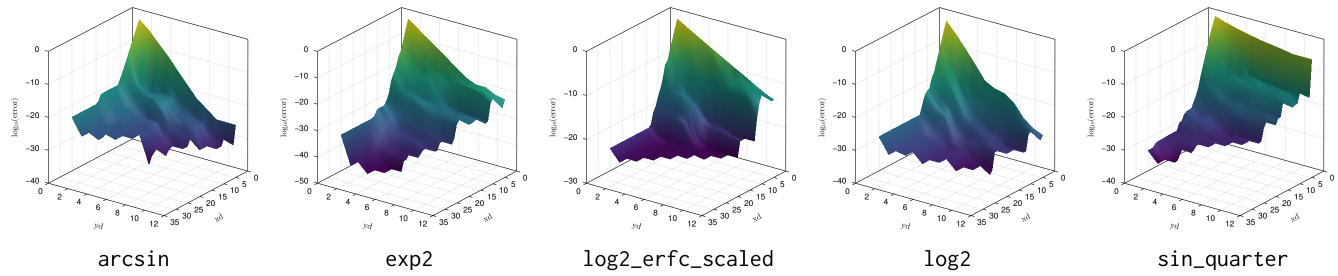

Figure 2: Absolute error of algebraic approximation of given degrees xd **and** yd **on the base domains of range reductions.**

```
Input: A vector of polynomial coefficients $c$ determining
       the polynomial $P$ and the algebraic approximation
       $a(c)$, a function to approximate $f$, a test set of
       domain points xs
Data: A regularization parameter $\rho \in (0, 1]$
Output: A good descent direction $-g$ for the approximation
        error $\|f - a(c)\| := \sup_{x \in \mathrm{xs}} |f(x) - a(c, x)|$ at the
        point $c$, a scalar tracking the scale of expected
        improvement $\alpha$
1  Set $y \longleftarrow a(c, \mathrm{xs})$ (interpreted as a vector with
   $y[i] = a(c, \mathrm{xs}[i])$).
2  Set $v \longleftarrow f(\mathrm{xs})$.
3  Set $\epsilon \longleftarrow y - v$.
4  Find all local maxima in $|\epsilon|$, meaning indices $i$ such that
   $|\epsilon[i]| \geq |\epsilon[i-1]|, |\epsilon[i+1]|$ and
   $|\epsilon[i]| > \min(|\epsilon[i-1]|, |\epsilon[i+1]|)$. Save these to an array $I$.
5  Set $n \longleftarrow \max(|\epsilon|)$.
6  Find all $i \in I$ such that $|\epsilon[i]| \geq \rho n$. Save these indices to $F$.
7  Define $Q[i] := \mathrm{sign}(\epsilon[F[i]]) \frac{-\partial_c P(\mathrm{xs}[F[i]], y[F[i]])}{\partial_y P(\mathrm{xs}[F[i]], y[F[i]])}$ for
   $i = 0, \ldots, \mathrm{len}(F) - 1$. (Note that $Q[i]$ is a vector of length
   $\mathrm{len}(c)$ for each $i$.)
8  Define $r[i] := |\epsilon|[F[i]] - n/2$ for $i = 0, \ldots, \mathrm{len}(F) - 1$.
9  Set $\alpha \longleftarrow \|r\|_2$.
10 Find a least-squares solution $g$ to the equation $Qg = r$.
11 Return $-g, \alpha$.
```

Algorithm 4: Finding the descent direction

Our line search procedure operates by fixing a parameter $\tau \in [0, 1]$, finding all the *local* maxima of the error function, picking out those local maxima which are at least τ times the error norm, and constraining the descent direction $\vec{q}$ to be a descent direction relative to the gradients of all these local maxima. Here the parameter τ may be interpreted as a regularization parameter: convergence far from a local optimum is easier to achieve for smaller values of τ, but reaching the optimum is prevented unless the value of τ is allowed to grow to 1 as the search progresses. Heuristically, the limited equioscillation that is required of the best approximation of a given pair of degrees can only be achieved when local maxima grow to meet shrinking global maxima in the course of descent.

Note first that $v(\vec{c}) = a(\vec{c}, u(\vec{c})) - f(u(\vec{c}))$. From this, we get

$$\begin{aligned} \partial_{\vec{c}} v(\vec{c}) &= \partial_{\vec{c}} a(\vec{c}, u(\vec{c})) + \partial_x a(\vec{c}, u(\vec{c})) \partial_{\vec{c}} u(\vec{c}) - f'(u(\vec{c})) \partial_{\vec{c}} u(\vec{c}) \\ &= \partial_{\vec{c}} a(\vec{c}, u(\vec{c})) + (\partial_x a(\vec{c}, u(\vec{c})) - f'(u(\vec{c}))) \partial_{\vec{c}} u(\vec{c}) \ . \end{aligned}$$

If $(u(\vec{c}), v(\vec{c}))$ is a local maximum of $E_{\vec{c}}(x) = a(\vec{c}, x) - f(x)$ in the interior of the domain, we have that $\partial_x a(\vec{c}, u(\vec{c})) - f'(u(\vec{c})) = 0$. The previous two identities show that $\partial_{\vec{c}} v(\vec{c}) = \partial_{\vec{c}} a(\vec{c}, u(\vec{c}))$. Given that $P(\vec{c})(x, a(\vec{c}, x)) = 0$, we have that $\partial_x P + \partial_y P \partial_x a = 0$. From $P(\vec{c})(u(\vec{c}), a(\vec{c}, u(\vec{c}))) = 0$ we have $\partial_{\vec{c}} P + \partial_x P \partial_{\vec{c}} u + \partial_y P(\partial_{\vec{c}} a + \partial_x a \partial_{\vec{c}} u) = \partial_{\vec{c}} P + \partial_y P \partial_{\vec{c}} a + (\partial_x P + \partial_y P \partial_x a) \partial_{\vec{c}} u = \partial_{\vec{c}} P + \partial_y P \partial_{\vec{c}} a = 0$. We thus have that $\partial_{\vec{c}} v = \partial_{\vec{c}} a = -\frac{\partial_{\vec{c}} P}{\partial_y P}$.

A similar analysis must also be carried out for the boundary of the domain, where $\partial_x a(\vec{c}, x) - f'(x) = 0$ may fail to hold. In this case we note that $\partial_{\vec{c}} u = 0$, from which $\partial_{\vec{c}} v = \partial_c a$ and $\partial_{\vec{c}} P + \partial_y P \partial_{\vec{c}} a = 0$, hence $\partial_{\vec{c}} v = -\frac{\partial_{\vec{c}} P}{\partial_y P}$ exactly as in the first case.

5 BOUNDING APPROXIMATION ERRORS

When evaluating an algebraic function f, satisfying $P(x, f(x)) = 0$ for some polynomial P, one has to be careful of *spurious branches*. Indeed, while $r = f(a)$ implies $P(a, r) = 0$, the opposite is not true, because $P(a, Y) \in \mathbb{R}[Y]$ may have several roots. The avoidance of spurious branches requires extra checks, and will be specific to the function f. E.g. in case of computing square roots we checked that the result is non-negative. Similarly restricting the position of points (a, r) works for other functions of interest.

A simple way to isolate the correct branch for evaluation is to find a rectangular region of the plane that covers the graph of the correct branch and is disjoint from other branches of the underlying polynomial – however, it may happen that no such rectangle exists. In such cases we may choose a cover of the correct branch consisting of rectangles, such that the cover is disjoint from other branches. This introduces additional steps to the evaluation of the approximation of the function f; these steps consist of obliviously selecting a rectangle (from the set of rectangles with public coordinates), and two range checks making sure that the point (a, r) is within that rectangle. Care must be taken to ensure that the bounds of the component rectangles require few bits to represent in fixed-point format, so that correctness would not require the use of a large number of fractional bits, creating a needless performance cost.

There is a second source for spurious evaluations of algebraic functions. Aside from spurious branches that exist in the underlying approximation, computations done with few fractional bits may

```
Input: An initial vector of polynomial coefficients $c_0$, a
       function to approximate $f$, a test set of domain
       points xs, the maximum number of iterations $k$, an
       initial gradient multiplier $\delta_0$
Data: A termination tolerance $\tau$, a perturbation parameter
      $\epsilon$, a line-search parameter $d$, a bound on iterations
      without sufficient impovement $n_f$
1  Initialize $c, \delta \leftarrow c_0, \delta_0$.
2  Initialize $e_p \leftarrow \|a(c_0) - f\| \coloneqq \sup_{x \in \mathrm{xs}} |a(c_0, x) - f(x)|$.
3  Initialize $c_b, e_b \leftarrow c_0, e_p$.
4  Initialize the number of steps since the last significant
     improvement $n \leftarrow 0$.
5  foreach $i = 1, \ldots, k$ do
6      Initialize $l \leftarrow 0$.
7      Set $v$ to the descent direction calculated by Alg. 4 and $\alpha$
         to the error scale.
8      Set $\delta_f, \delta_n \leftarrow \delta, \delta$.
9      Set $c_f, c_n \leftarrow c + \delta_f v, c + \delta_n v$.
10     Set $e_j \leftarrow \|a(c_j) - f\|$ for $j \in \{f, n\}$.
11     if $e_f < e_p$ then
12         while $e_n \geq e_f$ and $l \leq 2$ do
13             $\delta_n, e_n, c_n \leftarrow \delta_f, e_f, c_f$
14             $\delta_f \leftarrow \delta_f / d$
15             $c_f \leftarrow c + \delta_f v$
16             $e_f \leftarrow \|a(c_f) - f\|$
17             if $|e_p - e_f|/\alpha < \tau$ or $|e_n - e_f|/\alpha < \tau$ then
18                 $l \leftarrow l + 1$
19     else
20         while ($e_n \leq e_f$ or $e_n \geq e_p$) and $l \leq 2$ do
21             $\delta_f, e_f, c_f \leftarrow \delta_n, e_n, c_n$
22             $\delta_n \leftarrow d \cdot \delta_n$
23             $c_n \leftarrow c + \delta_n v$
24             $e_n \leftarrow \|a(c_n) - f\|$
25             if $|e_p - e_n|/\alpha < \tau$ or $|e_f - e_n|/\alpha < \tau$ then
26                 $l \leftarrow l + 1$
27     if $e_n < e_f$ then
28         $\delta, e_p, c \leftarrow \delta_n, e_n, c_n$
29     else
30         $\delta, e_p, c \leftarrow \delta_f, e_f, c_f$
31     if $e_p < e_b$ then
32         $c_b, e_b, n \leftarrow c, e_p, 0$
33     if $l \geq 2$ then
34         Set $q \leftarrow \epsilon$ and perturb $c$ by a random vector of
             length $q$. If the new error $e'$ is at most $4e_b$, set
             $e_p, \delta, l \leftarrow e', \delta_0, 0$. If not, then repeat the process
             with $q \leftarrow q/2$ each time until the condition is
             satisfied. If $q < \tau$ at any point, terminate algorithm.
35     If $n > n_f$, terminate algorithm.
36     Set $c \leftarrow c/\|c\|_2$.
37 Return $c_b, e_b$.
```

Algorithm 5: Finding the algebraic approximation of a function

introduce novel spurious results arising from numerical errors. If there are points (x, y) far from the desired branch, but where $|P(x, y)| < \epsilon$ for some small ϵ, then a numerical error that exceeds ϵ (while evaluating P) may introduce a spurious root of P near (x, y). To allow the end user of our algebraic approximations to avoid such issues, we have implemented a procedure which calculates an upper bound on the number of fractional bits required to ensure a given bound on the approximation error. The calculation of this bound accounts both for numerical spurious branches as well as numerical errors near the true branch.

We discuss the avoidance of these two kinds of spurious results in Sec. 5.1 and Sec. 5.2 below.

5.1 Isolating the branches of an algebraic function

Let $S \subseteq \mathbb{R}^2$ be a compact subset of the plane such that $\mathrm{graph}(y) = \{(x, y(x)) : x \in I\} \subseteq S$.

We define the functions $\delta(\varepsilon) \coloneqq \sup_{(x,z) \in S}\{|z - y(x)| : |P(x, z)| \leq \varepsilon\}$ and $L(d) \coloneqq \sup_{r \in \mathbb{R}}\{r : \delta(r) \leq d\}$.

To give a verbal interpretation, $\delta(\varepsilon)$ is the largest possible error in our approximation of y given an error of at most ε in our approximation of P. In a dual fashion, $L(d)$ is the least upper bound on the allowable errors in our approximation of P that guarantee an error of at most d in our approximation of y. (An error bound of *exactly* $L(d)$ is insufficient for topological reasons.)

Let $K, M \in \mathbb{R}$ be reals and C, E (C for *center*, E for *exterior*) be a cover of S (i.e. $S \subseteq C \cup E$) satisfying the following conditions:

- $(x, z) \in C \Rightarrow |P(x, z)| \geq K|z - y(x)|$
- $(x, z) \in E \Rightarrow |P(x, z)| \geq M$.

These sets are not guaranteed to exist for arbitrary P, y. It is sufficient that y be an isolated branch of P with $\forall x \in I, \partial_y P(x, y(x)) \neq 0$, and that S be sufficiently narrow around $\mathrm{graph}(y)$. Typically C will be a narrow band around $\mathrm{graph}(y)$ where we can choose $K = \inf\{|\partial_y P(x, z)| : (x, z) \in C\}$, and E will be the complement of C in S.

Now take any $r < M$ and let $(x, z) \in S$. We have that $|P(x, z)| \leq r \Rightarrow |P(x, z)| < M$, from which $(x, z) \notin E$, so $(x, z) \in C$. From this, $|z - y(x)| \leq \frac{|P(x,z)|}{K} \leq \frac{r}{K}$.

We derived that for $r < M$ and $(x, z) \in S$ that $|P(x, z)| \leq r \Rightarrow |z - y(x)| \leq \frac{r}{K}$, so by the definition of δ, we have that $\delta(r) \leq \frac{r}{K}$. We thus have that $r < M \Rightarrow \delta(r) \leq \frac{r}{K}$.

Fix any $d \in \mathbb{R}$. We have that $(r < M) \wedge (r \leq Kd) \Rightarrow \delta(r) \leq d$. From the definition of L we now have that $L(d) \geq \min(M, Kd)$.

We thus know that if we want our approximation of y to have error at most d, then all we need is our approximation error for P to be strictly less than $\min(M, Kd)$. The bound enforced by M accounts for spurious branches, while the bound of Kd accounts for errors near the true branch.

Suppose now that we have an approximation H of P. While the true roots of $H(x, \cdot)$ are approximations of the roots of $P(x, \cdot)$, and hence of $y(x)$, we may not have access to the true roots of H, and so must find approximate roots of an approximation, possibly compounding the error.

First let ε be an upper bound on the error $\|H - P\|_\infty$. We shall try to approximate $y(x)$ by finding values $y_a < y_b$ such that $H(x, y_a) \leq$

```
Input: An approximation polynomial $P$
Input: A rectangle $R = [a, b] \times [c, d]$ that intersects a
       desired branch
Input: An interval $L = [l_0, l_1] \subseteq \pi_y(R)$ determining the
       desired branch
Assumption: $P(a, y) = 0$ for exactly one $y \in L$
Output: A pair $(R', L')$, where $R'$ is the rectangle $R$,
        possibly shifted up or down by half the length of
        $\pi_y(R)$, and $L'$ is a subinterval of the middle two
        quarters of $\pi_y(R')$, such that $P(a, y) = 0$ for
        exactly one $y \in L'$
$q_1, q_3 \leftarrow 0.75c + 0.25d, 0.25c + 0.75d$;
$l'_0, l'_1 \leftarrow l_0, l_1$;
if $q_3 < l'_1$ then
    $n \leftarrow \max(q_3, l'_0)$;
    if $[n, l'_1]$ straddles the branch then
        return R shifted upward, $[n, l'_1]$
    $l'_1 \leftarrow q_3$;
if $l'_0 < q_1$ then
    $n \leftarrow \min(q_1, l'_1)$;
    if $[l'_0, n]$ straddles the branch then
        return R shifted downward, $[l'_0, n]$
    $l'_0 \leftarrow q_1$
return $R$, $[l'_0, l'_1]$
```

Algorithm 6: Fixing an invariant for the cover finding algorithm: FixInvariant

$0 \le H(x, y_b)$ and $|y_b - y_a|$ is small. (This assumes a certain sign convention for P, which may be chosen freely.)

Note that $|P(x, y_b) - P(x, y_a)| \le \sup_{(x,y)\in S}\{|\partial_y P(x, y)| : y \in [y_a, y_b]\} \cdot |y_b - y_a|$. Define $G := \sup_{(x,y)\in S} |\partial_y P(x, y)|$ and $\Delta := |y_b - y_a|$. Then $|P(x, y_b) - P(x, y_a)| \le G\Delta$.

We also have that $P(x, y_a) \le \varepsilon$, and $P(x, y_b) \ge -\varepsilon$. We can conclude from this that $|P(x, y_a)| \le \varepsilon + G\Delta$.

If we take y_a to be our approximation of $y(x)$, then we have $|y_a - y(x)| \le \delta(\varepsilon + G\Delta)$, from which $\varepsilon + G\Delta < L(d)$ is enough to guarantee an error of at most d. Since $L(d) \ge \min(M, Kd)$, this means all we need is $\varepsilon + G\Delta < \min(M, Kd)$.

In the fixed-point setting, we ordinarily have that $\Delta = 2^{-\mathrm{pp}}$. For a given algebraic approximation, the region S is chosen by Alg. 7 to be a suitable union of rectangles, and the constants G, M, K are computed via interval arithmetic in Julia, with the regions C, E being found automatically.

Note that for these error bounds to be sound, M, K may be replaced with underestimates, while G may be replaced with an overestimate.

5.2 Avoiding spurious roots from numeric errors

Suppose that we have two polynomials $P, P' \in \mathbb{R}[X]$, such that $\deg(P), \deg(P') \le d$, such that $P(x)$ represents exact polynomial evaluation at the point x and $P'(x)$ represents an approximate evaluation of P' at x via Horner's scheme using an approximate

```
Input: An approximation polynomial $P$
Input: A rectangle $[i_0, i_1] \times [j_0, j_1] = I \times J \subseteq \mathbb{R}^2$
       determining the isolation region
Input: An interval $L \subseteq J$ determining the desired branch
Assumption: $P(i_0, y) = 0$ for exactly one $y \in L$
Assumption: The desired branch has empty intersection
  with $\{i_0, i_1\} \times J$
Output: A list of rectangles $R_k \subseteq I \times J$ covering the
        intersection of the desired branch with $I \times J$ and
        intersecting no other branch
Define the partial derivative invariant function
  $PI(R) := 0 \notin \partial_y P(R)$;
Define the branch straddling invariant function $SI(R) :=$
  $P(\pi_x(R) \times \min \pi_y(R)) \cdot P(\pi_x(R) \times \max \pi_y(R)) \subseteq (-\infty, 0)$;
$k \leftarrow 0$;
Define $S(\lambda, R)$ as the uniform scaling of the rectangle $R$ by a
  factor of $\lambda$ about the center of its left edge;
Define $T(\lambda, R)$ as the scaling of the rectangle $R$ by a factor of
  $\lambda$ along the $x$-axis, keeping the left edge fixed;
Define $S'(\lambda, R, L)$ as FixInvariant($P, S(\lambda, R), L$), but with the
  rectangle intersected with $I \times J$;
Initialize $R_0, L_0 \leftarrow I \times J, L$;
while $\min \pi_x(R_k) < i_1$ do
    if $\neg(PI(R_k) \wedge SI(R_k))$ then
        while $\neg PI(R_k)$ do
            $R_k, L_k \leftarrow S'(0.5, R_k, L_j)$
        while $\neg SI(R_k)$ do
            $R_k \leftarrow T(0.5, R_k)$
    else
        $R', L' \leftarrow S'(2, R_k, L_k)$;
        while $PI(R') \wedge SI(R') \wedge R' \neq R_k$ do
            $R_k, L_k \leftarrow R', L'$;
            $R', L' \leftarrow S'(2, R', L')$;
        $R' \leftarrow T(2, R_k)$;
        while $PI(R') \wedge SI(R') \wedge R' \neq R_k$ do
            $R_k \leftarrow R'$;
            $R' \leftarrow T(2, R')$;
    $k \leftarrow k + 1$;
    $R_k, L_k \leftarrow (\pi_x(R_k) + \mathrm{diam}(\pi_x(R_k))) \times \pi_y(R_k), \pi_y(R_k)$;
    $R_k, L_k \leftarrow$ FixInvariant($P, R_k, L$);
return $[R_q : q \in 0 \ldots k]$
```

Algorithm 7: Finding a union of rectangles isolating a branch: BoxFinder

multiplication (such as fixed-point multiplication) with a constant error bound of α.

Let the coefficients of $P - P'$ be bounded in size by some $\beta > 0$. Then $|P(x) - P'(x)| \le \beta|x|^d + (\alpha + \beta)\sum_{k=0}^{d-1} |x|^k$. (This can be seen by noting that the error accumulation in one step of Horner's scheme is given by the function $f_{\mathrm{err}}(E) = E|x| + \alpha + \beta$, where E is the error at the current step and $f_{\mathrm{err}}(E)$ the error at the next. The bound given is equivalent to $f^n_{\mathrm{err}}(\beta)$.)

Applying the same reasoning for two-variable polynomials $Q, Q' \in \mathbb{R}[X,Y]$ with $\deg_x(Q), \deg_x(Q') \leq m$ and $\deg_y(Q), \deg_y(Q') \leq n$, first applying Horner's scheme along x and then along y, we find the error bound $|Q(x,y) - Q'(x,y)| \leq (\beta \sum_{k=0}^{m} |x|^k + \alpha \sum_{k=0}^{m-1} |x|^k) \cdot \sum_{k=0}^{n} |y|^k + \alpha \sum_{k=0}^{n-1} |y|^k$.

If we are evaluating Q and Q' on S, we have bounds on the coordinates of the input point $x' = \sup_{(x,y)\in S} |x|$ and $y' = \sup_{(x,y)\in S} |y|$. According to the logic of 5.1, we now have $\varepsilon \leq (\beta \sum_{k=0}^{m} x'^k + \alpha \sum_{k=0}^{m-1} x'^k) \sum_{k=0}^{n} y'^k + \alpha \sum_{k=0}^{n-1} y'^k$, with values $\alpha = 2^{-\mathsf{pp}}, \beta \leq 2^{-\mathsf{pp}}$ (with the latter bound coming from the truncation of the coefficients of Q). This allows us to explicitly confirm that the condition $\varepsilon + G\Delta < \min(M, Kd)$ holds for a given d.

6 BENCHMARKING

In order to benchmark and compare them, we have implemented the approximations for the transcendental functions listed in Sec. 4.2. The implementation consists of the following components.

- evaluator of algebraic functions, as specified in Alg. 3, and the range checker for isolating a single branch;
- preprocessor of arguments, and postprocessor of function values, as described in Sec. 4.2;
- The lists of coefficients of all approximations of all our primitive transcendental functions;
- The lists of rectangles for isolating the correct branch for each approximation.

We have used the domain-specific language ZK-SecreC [8, 9] for the implementation. We have chosen ZK-SecreC, because it provides us high-level means to express the computations we described in Sec. 3. In ZK-SecreC, both the computations performed by the circuit, and the computations performed locally by the parties (in particular, Prover) can be conveniently described. In fact, the standard library of ZK-SecreC contains the implementation of fixed-point numbers as described in Sec. 3: both the data structure, and the functions realizing the arithmetic operations.

ZK-SecreC compiler translates the programs into arithmetic circuits, and either expresses them SIEVE Intermediate Representation (IR) [10], or interfaces directly with a number of ZK protocol implementations. It allows the programs to be polymorphic over a number of parameters; our implementations are polymorphic over the modulus N that is used in the operations of the arithmetic circuit. Our implementation is also parametrized over the number of all bits (len) and fractional part bits (pp) in the representation of the fixed point number given as input, although a minimum number of bits are required for the integer part to avoid overflow in intermediate computations, where the minimum number of bits varies by function. A utility function is provided which allows the end user to compute an upper bound on the number of fractional bits required to achieve a desired absolute upper bound on the function's error in the main approximation region. The choice of field modulus N constrains the number of bits in the fixed point representation by the relation $\lfloor \log_2(N) \rfloor + 1 > 2 \cdot \mathsf{len}$ to ensure that no overflows occur in fixed point multiplication. This is the only constraint on the field modulus aside from those imposed by the ZKP backend.

The translation allows us to count the various kinds of operations that our implementations perform, and compare them on this basis. The interface with either Mac'n'Cheese [4] or EMP [32] allow us to measure the time that our approximations require. The generated IR, or the description of the circuit given to the interfaced backends may depend on the operations the latter support; e.g. in case the inequality checks, or permutations, or *Verifier's challenges* are directly supported by the back-end, our implementation may take advantage of them and reduce the effort spent on range checks. ZK-SecreC compiler produces the runtime(s) for Prover (and Verifier) that perform the local computations described in Sec. 3 while the protocol runs.

6.1 Precision

Figures 3–7 depict the achievable level of precision of the polynomial and algebraic approximations of the *primitive* functions (discussed in Sec. 4.2) on their intervals of approximation. The approximating polynomials and algebraic functions have been found in the way described in Sec. 4.3. Recall that the precision was defined as the maximum absolute error of the function value in this range.

The figures describe the obtainable approximation errors for different numbers of binary digits (pp) after the point, ranging from 20 to 60. Obviously, we cannot expect the absolute error to be smaller than $2^{-\mathsf{pp}}$, the decimal logarithm of which ranges from ca. -6 to ca. -18.

The horizontal axis in these figures corresponds to the "complexity" of evaluating the polynomial or the algebraic function in a ZKP protocol. Recall (Sec. 3.2) that for polynomials, the appropriate measure of "complexity" is their degree, while for algebraic functions defined by a two-variable polynomial $P(x,y)$, the measure is $\mathsf{xd} + 3\mathsf{yd}$ where xd and yd are the degrees of x and y in P.

Figures 3–7 show that as pp increases, polynomial approximations require increasing complexity to achieve the best possible precision. Moreover, different functions approach the maximum precision at very different speeds. In general, more "polynomial-like" functions (the ones where the coefficients of monomials in that function's Maclaurin series rapidly approach 0) converge faster. Similarly, algebraic functions require greater complexity to achieve the maximum possible precision for higher pp, but the convergence happens faster and with less variation for different functions.

The approximation errors have been estimated via 256-bit floating point arithmetic on 1600 evenly spaced points on the interval of approximation. While such estimations are not a guarantee of precision, they give us sufficient confidence that the behaviour of the approximation would not overly differ from the actual function.

6.2 Performance

We do not think that measuring the running time of our implementations of algebraic functions (together with pre- and postprocessing for the functions in Sec. 4.2) is the appropriate way to obtain empirical evidence on their execution performance. Indeed, these mathematical functions are low-level subroutines, not complete applications. The running time of an application that only invokes these functions is meaningfully compared only against itself, where the implementation of the functions is changed.

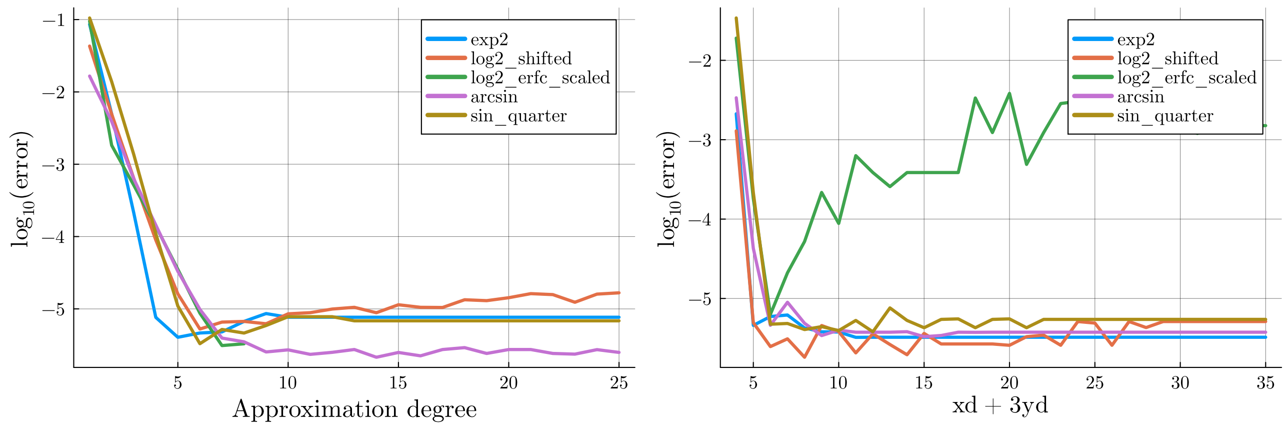

Figure 3: Absolute error of fixed point polynomial (left) and algebraic (right) approximation on base domain with pp = 20.

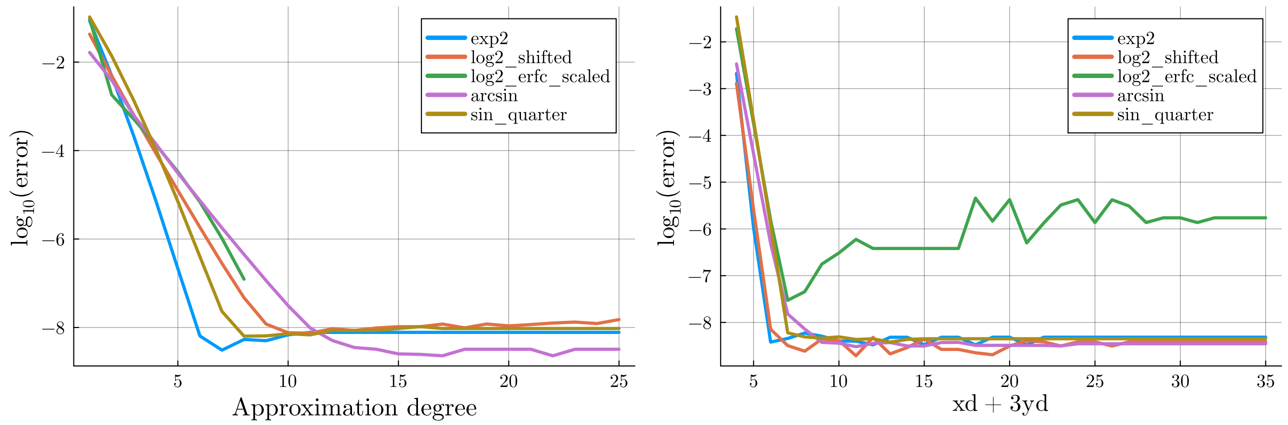

Figure 4: Absolute error of fixed point polynomial (left) and algebraic (right) approximation on base domain with pp = 30.

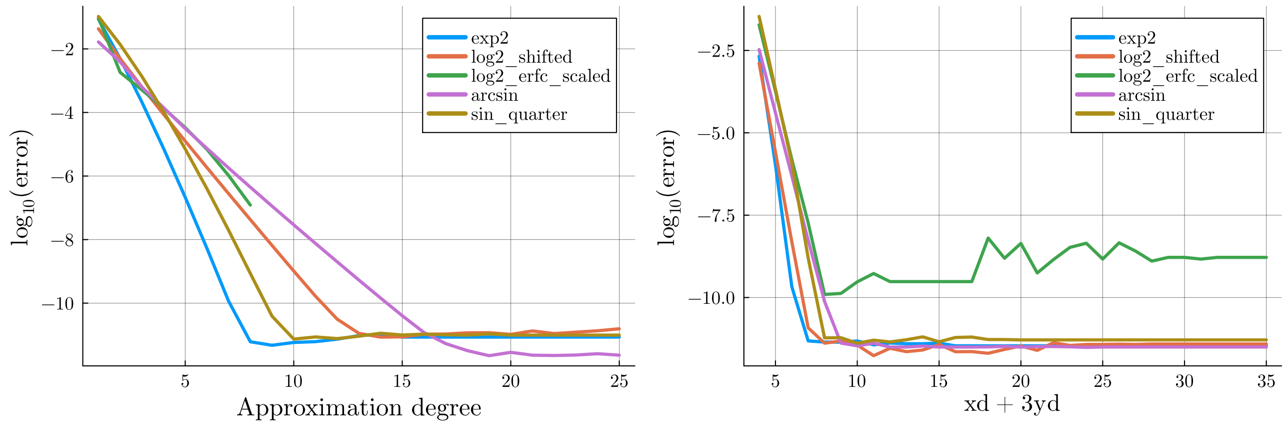

Figure 5: Absolute error of fixed point polynomial (left) and algebraic (right) approximation on base domain with pp = 40.

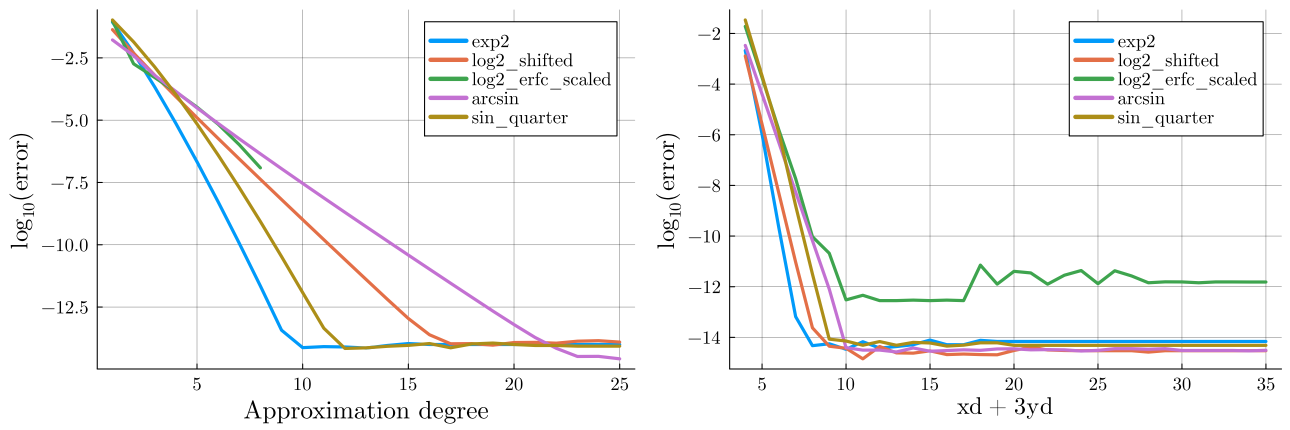

Figure 6: Absolute error of fixed point polynomial (left) and algebraic (right) approximation on base domain with pp = 50**.**

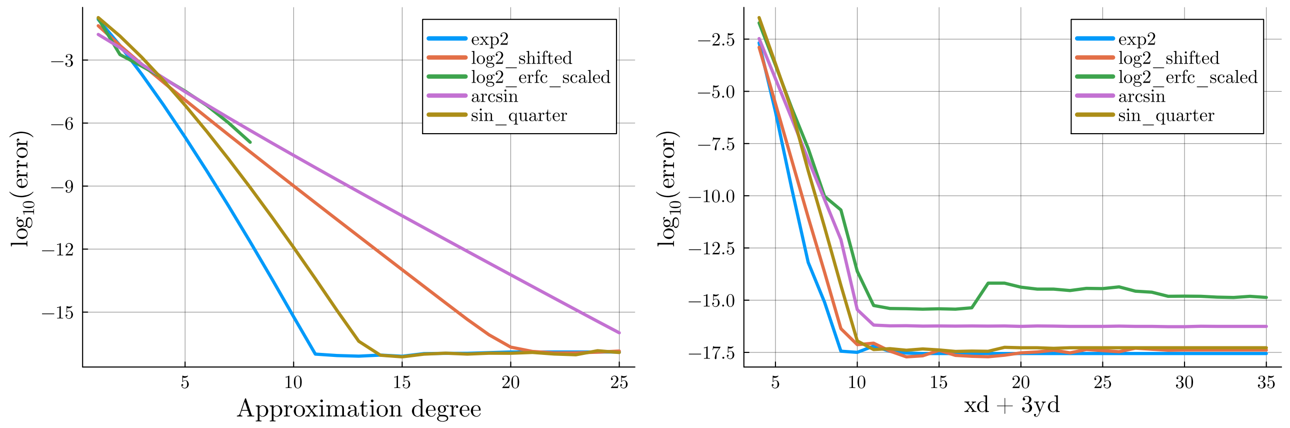

Figure 7: Absolute error of fixed point polynomial (left) and algebraic (right) approximation on base domain with pp = 60**.**

Instead, we will count the number of operations executed under the ZKP protocol, when the implementation of one of our mathematical functions is invoked by the statement that is to be proved. This number could be converted to the running time by considering the performance benchmarks of ZKP protocol implementations. The Mac'n'Cheese protocol is reported [4, Table 1] to be able to execute ca. 600 thousand multiplications per second in the field $\mathbb{Z}_{2^{61}-1}$, when running in a WAN setting (93 ms latency, 31.5 Mb/s bandwidth between Prover and Verifier), while EMP is reported [32, Table 2] to perform up to 8.9 million multiplications per second over the same field in a LAN-like setting. Our implementations also require Prover to locally solve polynomial equations, but there exist fast methods for doing it.

The number of arithmetic operations can be directly found from the representation of the computation in SIEVE IR [10] that is produced by ZK-SecreC compiler. Obviously, the cost of evaluating an algebraic function (Alg. 3) does not depend on the transcendental function that it approximates, although the pre- and postprocessing steps may depend on the function f that we actually want to evaluate. An arithmetic operation *op* in SIEVE IR is one of the following:

add defines a wire, where the value is equal to the sum of the values on the two input wires;

mul defines a wire, where the value is equal to the product of the values on the two input wires;

addc defines a wire, where the value is equal to the sum of the value on the input wire, and the constant that is part of the operation;

mulc defines a wire, where the value is equal to the product of the value on the input wire, and the constant that is part of the operation;

assert_zero checks that the value on a wire is equal to 0.

There are also operations for inputting values as part of the instance or part of the witness. Among these operations, **add**, **addc**, and **mulc** are *additive* operations that do not contribute significantly to the amount of resources necessary for evaluating the circuit on top of some ZKP protocol. The amount of resources mostly depends on the number of **mul** and **assert_zero** operations.

Counting the number of operations *op* in the SIEVE IR representation of the computation tells us that their number is equal to

$\#^{op}_{\mathrm{A}} = C^{op}_1(\mathrm{pp}+\mathrm{len})(\mathrm{xd}+3\mathrm{yd}) + C^{op}_2\mathrm{xdyd} + C^{op}_3\mathrm{xd} + C^{op}_4\mathrm{yd} + C^{op}_5\mathrm{len} + C^{op}_6$,

where the coefficients for each operation *op* are given in Table 1. This count includes the final range checks for picking the correct branch of the function (Sec. 5.1). We see that while the number of coefficients of an algebraic function is $\approx$ xd $\cdot$ yd, the number of costly operations does not depend on this quantity, being linear (but not multilinear) in xd and yd.

We can similarly count the number of operations for evaluating a polynomial (Alg. 2). We get

$$\#^{op}_{\mathrm{P}} = \hat{C}^{op}_1 d(\mathrm{pp}+\mathrm{len}) + \hat{C}^{op}_2 d + \hat{C}^{op}_3,$$

where the coefficients for each operation *op* are given in Table 2.

op	C^{op}_1	C^{op}_2	C^{op}_3	C^{op}_4	C^{op}_5	C^{op}_6
addc	0	0	1	4	0	17
add	2	1	2	4	20	4
mulc	2	1	4	9	16	16
mul	1	0	1	2	10	11
assert_zero	1	0	2	6	6	17

Table 1: Coefficients for polynomial giving number of gates in evaluation of an algebraic approximation of degree (xd, yd) **on a fixed-point value of type** (len, pp)**.**

Gate	$\hat{C}^{op}_1$	$\hat{C}^{op}_2$	$\hat{C}^{op}_3$
addc	0	2	0
add	2	1	0
mulc	2	2	1
mul	1	1	-1
assert_zero	1	2	0

Table 2: Coefficients for polynomial giving number of gates in evaluation of an polynomial approximation of degree d **on a fixed-point value of type** (len, pp)**.**

The preprocessing and postprocessing steps do depend on the function that we are approximating. On the other hand, they do not depend on the method of approximation (polynomial or algebraic). The cost of these steps for some of the functions is given in Table 3. For exp2 function, $8\mathrm{len}+\mathrm{pp}+3\max(\mathrm{pp},\mathrm{len}-\mathrm{pp})+4$ **mul**-operations are performed during pre- and postprocessing.

7 DISCUSSION

We have seen how to evaluate algebraic functions under ZK. We have also seen that algebraic approximations to transcendental functions is the way to go if the latter have to be evaluated under ZK; and we have seen a few concrete examples of approximation. The approximation methods have been described in Sec. 4.3 and Sec. 5, the first of them containing the method of finding the coefficients of an algebraic function, and the second allowing us to decide how good an approximation it is. In order to find a suitable approximation for an arbitrary function $y = f(x)$, we would follow the same methods. We would first decide the value pp; this value has likely been fixed by the rest of the computations. We would then run the coefficient finding method for small xd and yd, and find out the precision of approximation. If the precision is unsatisfactory, then we would increase xd and/or yd, and repeat. Note that Alg. 5 requires an initial approximation as one of its inputs; the coefficients found in previous iteration can serve as such input. Finally, having found a satisfactory approximation, we will use Alg. 7 for isolating the correct branch.

Function name	coef \ op	addc	add	mulc	mul
`arcsin`	len	0	30	26	15
	pp	0	2	2	1
	1	24	-1	19	11
`sin_quarter`	len	0	16	13	8
	pp	0	0	1	0
	1	14	1	10	6
`log2`	len	1	29	26	16
	pp	0	0	0	0
	1	12	5	18	6

Delta coefficients

Table 3: Coefficients for polynomial giving the number of gates for pre- and postprocessing the function argument and value, when working with fixed-point values of type (len, pp)**.**

How useful are our approximations, or Alg. 3 for evaluating algebraic functions, or even the fixed-point and floating-point representations of real numbers in computations run on top of ZKP protocols in general? When making a statement over reals, we always want to show that one value is smaller than another one (i.e. we never want to show the equality; we may want to show that some difference is less than some threshold, though). The errors introduced by the representations and approximations may flip the value of the statement. While for many kinds of statements over reals, it is actually possible to compute them in the way that we can be certain in the outcome [25], this greatly increases the complexity of computations. Instead, we perform the computations as-is, hoping that we are sufficiently precise to get the correct outcome. The approximations we introduce in this paper support that hope, because they can be very close to the real functions. In ZKP setting, one additionally has to worry about Prover's ability to introduce errors. Our methods of approximating transcendental functions and computing algebraic functions do not contain any points where additional errors could be introduced.

REFERENCES

[1] Sebastian Angel, Andrew J. Blumberg, Eleftherios Ioannidis, and Jess Woods. 2022. Efficient representation of numerical optimization problems for snarks.

Kevin R. B. Butler and Kurt Thomas, (Eds.) (2022). https://www.usenix.org/conference/usenixsecurity22/presentation/angel.

[2] Joshua Baron. 2023. I was told there would be blockchain: 5 Years of Real World Crypto at DARPA. Talk given at Real World Crypto Symposium. (2023).

[3] Victor Barsan. 2015. Algebraic approximations for transcendental equations with applications in nanophysics. *Philosophical Magazine*, 95, 27, 3023–3038. DOI: 10.1080/14786435.2015.1081425.

[4] Carsten Baum, Alex J. Malozemoff, Marc B. Rosen, and Peter Scholl. 2021. Mac'n'Cheese: Zero-Knowledge Proofs for Boolean and Arithmetic Circuits with Nested Disjunctions. In *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part IV* (Lecture Notes in Computer Science). Tal Malkin and Chris Peikert, (Eds.) Vol. 12828. Springer, 92–122. DOI: 10.1007/978-3-030-84259-8_4.

[5] Mihir Bellare and Shafi Goldwasser. 1989. New paradigms for digital signatures and message authentication based on non-interative zero knowledge proofs. In *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings* (Lecture Notes in Computer Science). Gilles Brassard, (Ed.) Vol. 435. Springer, 194–211. DOI: 10.1007/0-387-34805-0_19.

[6] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 459–474. DOI: 10.1109/SP.2014.36.

[7] Aharon Ben-Tal and Arkadi Nemirovski. 2023. Lecture notes: optimization iii. https://www2.isye.gatech.edu/~nemirovs/OPTIIILN2023Spring.pdf. (2023).

[8] Dan Bogdanov, Joosep Jääger, Peeter Laud, Härmel Nestra, Martin Pettai, Jaak Randmets, Ville Sokk, Kert Tali, and Sandhra-Mirella Valdma. 2022. ZK-SecreC: a domain-specific language for zero knowledge proofs. *CoRR*, abs/2203.15448. arXiv: 2203.15448. DOI: 10.48550/arXiv.2203.15448.

[9] Dan Bogdanov et al. 2024. ZK-SecreC: a Domain-Specific Language for Zero-Knowledge Proofs. In *Proceedings of CSF 2024 - 37th IEEE Computer Security Foundations Symposium*. To appear.

[10] Paul Bunn et al. 2022. SIEVE Intermediate Representation. https://github.com/sieve-zk/ir. (2022).

[11] Octavian Catrina. 2021. Complexity and performance of secure floating-point polynomial evaluation protocols. In *Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part II* (Lecture Notes in Computer Science). Elisa Bertino, Haya Schulmann, and Michael Waidner, (Eds.) Vol. 12973. Springer, 352–369. DOI: 10.1007/978-3-030-88428-4_18.

[12] Octavian Catrina and Claudiu Dragulin. 2009. Multiparty computation of fixed-point multiplication and reciprocal. In *Database and Expert Systems Applications, DEXA, International Workshops, Linz, Austria, August 31-September 4, 2009, Proceedings*. A Min Tjoa and Roland R. Wagner, (Eds.) IEEE Computer Society, 107–111. DOI: 10.1109/DEXA.2009.84.

[13] Octavian Catrina and Amitabh Saxena. 2010. Secure computation with fixed-point numbers. In *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers* (Lecture Notes in Computer Science). Radu Sion, (Ed.) Vol. 6052. Springer, 35–50. DOI: 10.1007/978-3-642-14577-3_6.

[14] S. Chevillard, M. Joldeş, and C. Lauter. 2010. Sollya: an environment for the development of numerical codes. In *Mathematical Software - ICMS 2010* (Lecture Notes in Computer Science). K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, (Eds.) Vol. 6327. Springer, Heidelberg, Germany, (Sept. 2010), 28–31.

[15] Richard Dedekind and Heinrich Weber. 2012. *Theory of Algebraic Functions of One Variable*. Trans. by John Stillwell.

[16] Vassil S. Dimitrov, Liisi Kerik, Toomas Krips, Jaak Randmets, and Jan Willemson. 2016. Alternative implementations of secure real numbers. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, (Eds.) ACM, 553–564. DOI: 10.1145/2976749.2978348.

[17] Boyuan Feng, Lianke Qin, Zhenfei Zhang, Yufei Ding, and Shumo Chu. 2021. Zen: an optimizing compiler for verifiable, zero-knowledge neural network inferences. Cryptology ePrint Archive, Paper 2021/087. https://eprint.iacr.org/2021/087. (2021). https://eprint.iacr.org/2021/087.

[18] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. Alfred V. Aho, (Ed.) ACM, 218–229. ISBN: 0-89791-221-7. DOI: 10.1145/28395.28420.

[19] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems (Extended Abstract). In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*. Robert Sedgewick, (Ed.) ACM, 291–304. DOI: 10.1145/22145.22178.

[20] Takao Hayashi. 2019. Bhaskara I. Encyclopedia Britannica, https://www.britannica.com/biography/Bhaskara-I. (2019).

[21] Liina Kamm and Jan Willemson. 2015. Secure floating point arithmetic and private satellite collision analysis. *Int. J. Inf. Sec.*, 14, 6, 531–548. DOI: 10.1007/S10207-014-0271-8.

[22] Toomas Krips and Jan Willemson. 2014. Hybrid model of fixed and floating point numbers in secure multiparty computations. In *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings* (Lecture Notes in Computer Science). Sherman S. M. Chow, Jan Camenisch, Lucas Chi Kwong Hui, and Siu-Ming Yiu, (Eds.) Vol. 8783. Springer, 179–197. DOI: 10.1007/978-3-319-13257-0_11.

[23] Toomas Krips and Jan Willemson. 2015. Point-counting method for embarrassingly parallel evaluation in secure computation. In *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers* (Lecture Notes in Computer Science). Joaquín García-Alfaro, Evangelos Kranakis, and Guillaume Bonfante, (Eds.) Vol. 9482. Springer, 66–82. DOI: 10.1007/978-3-319-30303-1_5.

[24] Tianyi Liu, Xiang Xie, and Yupeng Zhang. 2021. Zkcnn: zero knowledge proofs for convolutional neural network predictions and accuracy. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, (Eds.) ACM, 2968–2985. DOI: 10.1145/3460120.3485379.

[25] Kurt Mehlhorn and Stefan Näher. 1999. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press.

[26] Sarah Scheffler, Eran Tromer, and Mayank Varia. 2022. Formalizing human ingenuity: A quantitative framework for copyright law's substantial similarity. In *Proceedings of the 2022 Symposium on Computer Science and Law, CSLAW 2022, Washington DC, USA, November 1-2, 2022*. Daniel J. Weitzner, Joan Feigenbaum, and Christopher S. Yoo, (Eds.) ACM, 37–49. DOI: 10.1145/3511265.3550444.

[27] Srinath T. V. Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J. Blumberg, and Michael Walfish. 2012. Taking Proof-Based Verified Computation a Few Steps Closer to Practicality. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*. Tadayoshi Kohno, (Ed.) USENIX Association, 253–268. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/setty.

[28] L.N. Trefethen. 2013. *Approximation Theory and Approximation Practice. Other Titles in Applied Mathematics*. SIAM. ISBN: 9781611972405. https://books.google.ee/books?id=h80N5JHm-u4C.

[29] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. 2021. Mystique: efficient conversions for zero-knowledge proofs with applications to machine learning. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. Michael D. Bailey and Rachel Greenstadt, (Eds.) USENIX Association, 501–518. https://www.usenix.org/conference/usenixsecurity21/presentation/weng.

[30] Baisheng Wu, Weijia Liu, Zhijun Wang, and Xin Chen. 2018. Approximate expressions for solutions to two kinds of transcendental equations with applications. *J. Phys. Commun.*, 2, 055009. DOI: 10.1088/2399-6528/aac0e8.

[31] Runhua Xu, Nathalie Baracaldo, and James Joshi. 2021. Privacy-preserving machine learning: methods, challenges and directions. (2021). arXiv: 2108.04417 [cs.LG].

[32] Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. 2021. Quicksilver: efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, (Eds.) ACM, 2986–3001. DOI: 10.1145/3460120.3484556.