Dandelion: Certified Approximations of Elementary Functions

Heiko Becker
MPI-SWS, Germany

Mohit Tekriwal
University of Michigan – Ann Arbor, USA

Eva Darulova
Uppsala University, Sweden

Anastasia Volkova
Nantes Université, France

Jean-Baptiste Jeannin
University of Michigan – Ann Arbor, USA

Abstract

Elementary function operations such as *sin* and *exp* cannot in general be computed exactly on today's digital computers, and thus have to be approximated. The standard approximations in library functions typically provide only a limited set of precisions, and are too inefficient for many applications. Polynomial approximations that are customized to a limited input domain and output accuracy can provide superior performance. In fact, the Remez algorithm computes the best possible approximation for a given polynomial degree, but has so far not been formally verified.

This paper presents *Dandelion*, an automated certificate checker for polynomial approximations of elementary functions computed with Remez-like algorithms that is fully verified in the HOL4 theorem prover. Dandelion checks whether the difference between a polynomial approximation and its target reference elementary function remains below a given error bound for all inputs in a given constraint. By extracting a verified binary with the CakeML compiler, Dandelion can validate certificates within a reasonable time, fully automating previous manually verified approximations.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases elementary functions, approximation, certificate checking

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Exact computation in real-number arithmetic is in general too inefficient for most applications [7], and is thus typically replaced by finite-precision (floating-point or fixed-point) arithmetic. While arithmetic operations such as addition and multiplication are well-supported and efficient, real-world code often also needs to support elementary functions such as *sin* and *exp*. Such functions cannot be computed exactly on today's digital hardware and thus necessarily have to be approximated. For floating-point arithmetic, libraries provide general-purpose approximations for a limited set of formats, e.g. correctly rounded to single or double precision [23, 14]. However, these can be inefficient for applications that do not need quite as much accuracy, or that only need to work for a limited set of inputs [15, 22]. Furthermore, many applications, especially in the embedded systems domain, operate with fixed-point arithmetic, for which efficient library approximations do not exist [21].

When library function implementations are suboptimal, a possible solution is to generate approximations of elementary functions on demand with custom accuracy—exactly the accuracy that is needed by the application and its context. Indeed, automated algorithms exist for generating polynomial approximations of elementary functions with a given polynomial

degree or given bound on the *approximation error*. For instance, Remez-like algorithms [32] generate the best polynomial approximation, i.e. the one with the smallest approximation error for a given polynomial degree. We can, for example, approximate *exp* on $[0, 0.5]$ by $0.999 + 1.001 * x + 0.484 * x^2 + 0.215 * x^3$ with an approximation error of $2.63 * 10^{-5}$.

Remez-like algorithms are used in several automated tools for generating custom approximations [10, 22], however, these implementations are not formally verified. This is especially problematic because the algorithms are generally tricky to get right [28][1].

In this paper, we implement and prove correct *Dandelion*, the first fully automated and formally verified certificate checker for polynomial approximations computed by Remez-like algorithms. Dandelion is implemented and fully verified inside the HOL4 theorem prover [34]. A certificate for Dandelion consists of an elementary function f, an input interval I, and a polynomial approximation p and an approximation error ϵ returned by a Remez-like algorithm. We prove once and for all the correctness theorem that if Dandelion returns true for a certificate, the encoded error is a true upper bound to the difference between the elementary function and the encoded polynomial: $\max_{x \in I(x)} |f(x) - p(x)| \leq \epsilon$.

Dandelion's certificates are minimal, requiring only inputs and outputs of the approximation algorithm to be recorded. Additionally, Dandelion certifies the known best possible approximations of Remez-like algorithms, together, making it widely applicable. Previous work has either focused on manual proofs [19] or certifies only results of Chebyshev approximations, which are not as accurate as those computed by Remez-like algorithms [8]. The key challenge that Dandelion solves is *automation*; Dandelion requires no user interaction, making it the first fully automated validator for results of Remez-like algorithms.

One may think that verifying an implementation of a Remez-like algorithm should be favored over validating each run separately. However, correctness proofs for one implementation generally do not apply to other implementations, and thus would have to be re-done with every change. In contrast, by certifying only the end-result, Dandelion is indifferent to the implementation choices and thus immediately more widely applicable.

Harrison [19] has manually verified a polynomial approximation of the exponential function in HOL-Light [2]. The methodology presented is general, but was never automated. Dandelion borrows the high-level approach from Harrison's manual proof, automating the key ideas to validate results of Remez-like algorithms.

While the idea of automating an existing development may seem simple, we faced two major challenges to make automated certification practical. First, computations in theorem provers are generally slower than those in unverified tools, making certain designs impractical. Second, some definitions of Harrison use non-computable functions and thus cannot be used in an automated approach. To speed-up the computations, we extract Dandelion as a verified binary using the CakeML compiler [35]. The extracted binary enjoys the same correctness guarantees as our in-logic implementation, and makes checking a certificate fast: a single certificate is checked on average within 6 minutes. We overcome the problem of non-computable functions by identifying computable versions and proving equivalence between the computable and non-computable functions.

Dandelion can be used as a verifier for any Remez-like algorithm. In our evaluation, we use Dandelion to certify a number of approximations generated from FPBench [13] and the work by Izycheva *et al.* [21]. Our evaluation shows that certificate checking in Dandelion is fast, and that Dandelion certifies, for an elementary function f and polynomial p, approximation

[1] Muller warns: "[..] even if the outlines of the [Remez] algorithm are reasonably simple, making sure that an implementation will always return a valid result is sometimes quite difficult." ([28], page 52).

```
def polToCart_x(radius: Real,
  theta: Real): Real = {
 require(((1.0 <= radius) &&
  (radius <= 10.0) && (0.0 <= theta) &&
  (theta <= 360.0)))
 val pi = 3.14159265359
 val radiant = (theta * (pi / 180.0))
 (radius * cos(radiant))
}
```

(a) Example kernel using a elementary function

```
def polToCart_x(radius: Fixed,
 theta: Fixed): Fixed = {
 val pi = 3.14159265359
 val radiant = (theta * (pi / 180.0))
 val _tmp = (1.3056366443634033 +
  (radiant * (-1.2732393741607666 +
  (radiant * (0.2026423215866089 +
  (radiant * 3.3222216089257017e-09))))))
 (radius * _tmp)
  }
```

(b) Example with polynomial approximation for *cos*

```
cos_cert = <| f := Fun Cos (Var "radiant"); n := 32;
  (* p (x) ~ 1.305 - 1.273 * x + 0.202 * x^2 + 3.322 * 10^-9 * x^3 *)
  p := [5476237/4194304; -5340353/4194304; 1699887/8388608; 3740489/1125899906842624];
  ε := 766133524584849981160987377038947873961143126798 7/(25 * 10^48); (* ~0.306 *)
  I := [("radiant", (0, 314159265359/50000000000))]; (* ~ x in [0, 6.284] *) |>
```

(c) Certificate for the approximation of *cos* in the example

Figure 1 Example kernel using a elementary function (top-left), the kernel with a polynomial approximation (top-right), and the certificate for Dandelion (bottom)

errors on the same order of magnitude as the infinity norm ($\max_{x \in I(x)} |f(x) - p(x)|$). We also encode the original proof-goal of Harrison as a certificate—Dandelion reduces its proof to a single line of code.

Contributions

In summary, this paper provides the following contributions: We

- implement Dandelion[2], a verified certificate checker for polynomial approximations,
- extract a verified binary using CakeML to make certificate checking fast, and
- evaluate Dandelion's performance on a set of benchmarks and compare it against an automated, but not formally verified, theorem prover.

2 Overview

Before we dive into the technical details of Dandelion, we give an overview of our toolchain and the proofs that Dandelion performs automatically using the example in Figure 1. The starting point is the code in Figure 1a that converts polar to cartesian coordinates, and returns the resulting x component. This code could for example be part of an autonomous car or a drone, and inaccuracies in the conversion of coordinates may have catastrophic effects [29]. Chip sizes and energy budgets in these devices are usually small, and thus using a fully-fledged floating-point unit is not always possible. As an alternative, code is often implemented in fixed-point arithmetic, which, however, does not come with standard and efficient library implementations of elementary functions [21]. Hence, an engineer may approximate the function *cos* on line 8 in Figure 1a with a custom polynomial approximation

[2] The source code of Dandelion is publicly available at `https://github.com/HeikoBecker/Dandelion`.

shown in Figure 1b, for instance using the state-of-the-art synthesis tool Daisy [21][3].

Daisy internally calls a Remez-like algorithm to generate the polynomial approximation of *cos*, but the approximation algorithm and Daisy itself are not (formally) verified. With Dandelion, we can straight-forwardly instrument Daisy to generate the certificate shown in Figure 1c that encodes the elementary function to be approximated (`f`), the approximating polynomial (`p`), the approximation error (ε), and the range on which the approximation is supposed to be valid (`I`), and an additional parameter `n` which we explain later. Note that the input interval `I` recorded in the certificate captures the direct inputs to the elementary function *cos* and is thus different from the input interval in the `require` clause that captures inputs to the overall function `polToCart_x`. Dandelion validates this certificate in 31 seconds and proves the HOL4 theorem

Theorem 1. : $\forall x.\ x \in \mathtt{I}(x) \Rightarrow |cos(x) - \mathtt{p}(x)| \leq \varepsilon$

If the approximation error had not been correct, the binary would emit an error message, explaining which part of the validation failed.

The certificate in Figure 1c uses only a single elementary function. In general, Dandelion supports more complicated elementary function expressions, like $exp(x * \frac{1}{2})$, and $sin(x-1) + cos(x+1)$. Exactly as Remez-like algorithms, we only require the functions to be univariate, *i.e.* the certificate can only have a single free variable. Any approximation tool that can generate these certificates can be used to generate inputs for Dandelion, and Dandelion can be used independently of a particular approximation algorithm implementation.

The approach used by Dandelion has been layed out previously in a manual proof for the exponential function by Harrison [19] (Section 2.1 overviews the main theorems and ideas). The presented high-level approach is general, but a key challenge that Dandelion solves is to automate each step and extend them to more complex expressions (Section 2.2).

2.1 Manual Proof by Harrison

Overall, Harrison has manually verified an approximation by Tang [36] of the exponential function, showing that: $\forall x.x \in [-0.010831, 0.010831] \Rightarrow |((e^x - 1) - p(x)| \leq 2^{-33.2}$. The manual verification by Harrison is split into two steps. First, Harrison simplifies the overall proof goal to a proof about polynomials, by replacing $e^x - 1$ with a high-accuracy truncated Taylor series q. By truncating the series after the 7th term, the approximation error of the series becomes 2^{-58} and the overall proof-goal is reduced from $|((e^x - 1) - p(x)| \leq 2^{-33.2}$ with the triangle inequality to

$$|q(x) - p(x)| \leq 2^{-33.2} - 2^{-58} \tag{1}$$

The difference between $q(x)$ and $p(x)$ itself is a polynomial $h(x)$, and thus this first step reduces the overall proof goal to proving an upper bound on the polynomial h. As h represents the difference between two polynomial approximations, the points where h attains its maximum value (*i.e.* its extremal points) are those where the approximation error is the largest. It thus suffices to reason about the extremal points of h for proving the inequality.

To prove the polynomial inequality, Harrison proved two well-known mathematical theorems in HOL-Light. The first theorem proves that function f on the closed interval $[a, b]$ attains its extremal values either at the outer points or at the points where the first derivative is zero:

[3] Daisy can also synthesize suitable finite-precision types for the polynomial (not shown here), and generate certificates that formally verify the *roundoff* error bound of this polynomial implementation [6].

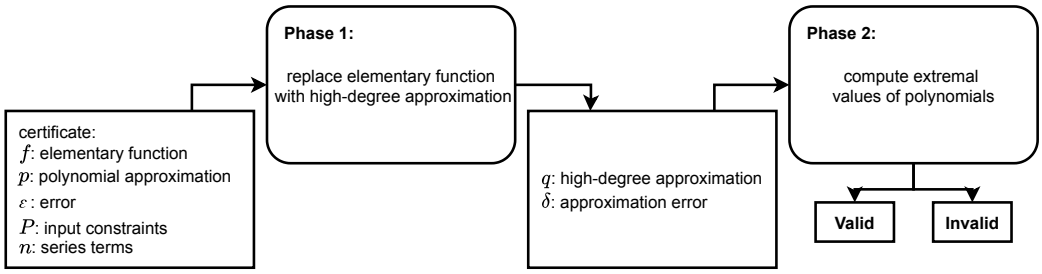

Figure 2 Overview of Dandelion toolchain

Theorem 2. Let p a differentiable, univariate polynomial, defined on $[a, b]$ and ub a real number, then

$$|\,p(a)\,| \leq ub \wedge |\,p(b)\,| \leq ub \wedge$$
$$(\forall x.\ a \leq x \leq b\ \wedge\ p'(x) = 0 \Rightarrow |\,p(x)\,| \leq ub) \Rightarrow$$
$$(\forall x.\ a \leq x \leq b \Rightarrow |\,p(x)\,| \leq ub)$$

The second theorem is called *Sturm's theorem* and proves that the exact number of zeros of a polynomial can be computed from the so-called Sturm sequence of polynomials:

Theorem 3. Let p a differentiable, univariate polynomial, defined on $[a, b]$. If p has non-zero values on both a and b, and its derivative is not the constant zero function, then we call *Sturm* ss the sturm sequence for p, and the set of zeros of p has size $V(a, ss) - V(b, ss)$.

The Sturm sequence ss of polynomial p is defined recursively as

$$ss_0 = p \qquad ss_1 = p' \qquad ss_{i+1} = -\text{rem}\,(ss_{i-1}, ss_i)$$

where rem computes the remainder of the polynomial division $\frac{ss_{i-1}}{ss_i}$. Computation stops once the remainder becomes the constant 0 polynomial. Function $V(a, ss)$ in Theorem 3 computes the number of sign changes when evaluating the polynomials in the list ss on value a.

To prove the final inequality, Harrison computes unverified guesses for both the Sturm sequence of $h'(x)$ and the zeros of $h'(x)$ using Maple, and manually validates them in HOL4 using Theorem 3. By knowing the number of zeros, and their values, Harrison then provably derives an upper bound on the extremal values of polynomial h using Theorem 2.

2.2 Automated Proofs in Dandelion

As in Harrison's approach, Dandelion splits the proof into two parts. In the first phase, Dandelion replaces elementary functions in the certificate by high-accuracy approximations computed inside HOL4. The second phase then proves that the approximation error is a correct upper bound on the extremal points of the resulting polynomial by finding zeros of the derivative and bounding the number of zeros with Sturm sequences. The key differences between Dandelion and the proof by Harrison is that Dandelion supports more elementary functions, *i.e. exp*, *sin*, *cos*, *ln*, and *arctan*, and that its certificate checking is fully automated and does not require any user interaction or additional proofs. Figure 2 gives an overview of the automatic computations done by Dandelion. We explain them at a high-level for our example from Figure 1.

To prove the overall correctness theorem (Theorem 1), Dandelion first computes a computable high-accuracy polynomial approximation for *cos*, denoted by q, using a truncated Taylor series of degree `n`, with an approximation error of `3.77e-3`. Generally, the certificate parameter `n` defines the number of series terms computed for the truncated Taylor series in Dandelion. Exactly as for Harrison's manual proof, Dandelion proves an upper bound on the difference between q and the target polynomial approximation p (Equation 1). Coming up with a general approach for computing accurate truncated Taylor series of arbitrary elementary functions was a major challenge for Dandelion—we implemented a library of general purpose Taylor series for the supported elementary functions. Given a target degree, Dandelion automatically computes a polynomial implementation and proves an approximation error for the truncated series. We explain the first phase in more detail in Section 3.

In the second phase, Dandelion computes an upper bound on the polynomial $h(x) = q(x) - p(x)$ exactly like Harrison, by reasoning about the zeros of its derivative h', using Sturm sequences to bound the number of zeros of h'. Based on the number of zeros, Dandelion uses an (unverified) oracle to automatically come up with a list of zeros of h'.

To prove the final bound on h, Dandelion checks for each zero of h' that the value of h at this point is smaller than or equal to the residual error $\varepsilon -$ `3.77e-3`. Harrison's definition of Sturm sequences is defined as a non-computable predicate, involving an existentially quantified definition of polynomial division. Key to Dandelion is the implementation of a computable version of polynomial division, as well as Sturm sequences, in combination with equivalence proofs relating them to Harrison's predicates. We explain how Dandelion computes the Sturm sequences automatically and how Dandelion estimates zeros of polynomials in more detail in Section 4.

Computing the Sturm sequence is the most computation intensive part of Dandelion, which we found to be impractical to do in logic. Thus we extract a verified binary using the CakeML compiler for the second phase of Dandelion only. For the extraction to work, we translated the HOL4 definitions of the second phase into CakeML source code via CakeML's proof-producing synthesis tool [4]. We explain the extraction with CakeML in Section 5.

3 Automatic Computation of Truncated Taylor Series

As in Harrison's manual proof, Dandelion replaces in a first step the elementary function in the certificate by a high-accuracy polynomial approximation. The crucial difference is that Dandelion automates all of the manual steps, which we explain next.

When checking a certificate for function f, with input range I, Dandelion automatically replaces every occurrence of an elementary function in f with a truncated Taylor series $t_{f,n}$. Below, we take f to be a single elementary function and will discuss the extension to more complicated elementary function expressions later. The parameter n of $t_{f,n}$ is part of the certificate and specifies the number of terms computed for the truncated series, *i.e.* if n is 32, Dandelion truncates the Taylor series of f after the 32nd term. The final result of the phase is a high-accuracy polynomial approximation of function f, $q_{f,n}$, and an overall approximation error $\delta_{f,n}$. For the simple case where f is a single elementary function, $q_{f,n}$ and $t_{f,n}$ are the same. Once we extend Dandelion to more complicated expressions, Dandelion combines different instances of $t_{f,n}$, into the final $q_{f,n}$, as we explain later. We implement the first phase in a HOL4 function `approxAsPoly` and prove soundness of `approxAsPoly` once and for all in HOL4:

Theorem 4 (First Phase Soundness)**.**

$$\texttt{approxAsPoly}\ f\ I\ n = \texttt{Some}(q_{f,n}, \delta_{f,n}) \Rightarrow (\forall x.\ x \in I \Rightarrow |\, f(x) - q_{f,n}(x)\,| \leq \delta_{f,n})$$

The theorem states that if `approxAsPoly` succeeds and returns $q_{f,n}$ and $\delta_{f,n}$, then the approximation error on input range I between $f(x)$ and $q_{f,n}(x)$ is upper bounded by $\delta_{f,n}$.

In the rest of this section, we first explain how Dandelion automatically computes truncated Taylor series for elementary functions like *sin* and *exp*, then we explain how Dandelion extends this approach in `approxAsPoly` to compute a single polynomial approximations of more complicated elementary function expressions like $exp(x * \frac{1}{2})$ via interval analysis and propagation of polynomial errors. Throughout this section, we use f to refer to the elementary function from the certificate, $t_{f,n}$ as truncated Taylor series, $\delta_{t,n}$ as the approximation error of the series, $q_{f,n}$ as polynomial approximation, and $\delta_{f,n}$ as the overall approximation error.

3.1 Truncated Taylor Series for Single Elementary Functions

Both $t_{f,n}$ and $\delta_{t,n}$ depend on the approximated elementary function f, as well as the number of series terms n from the certificate. Overall, Dandelion automatically computes a truncated Taylor series for the elementary functions *sin*, *cos*, *exp*, *arctan*, and *ln*[4]; the series expansions for *exp* and *ln* already existed in HOL4 prior to Dandelion and we port the series for *arctan* from HOL-Light. For *sin* and *cos* we prove series based on textbook descriptions. Formally, Dandelion proves a truncated Taylor series for each elementary function once and for all as

Theorem 5. $\forall x\ n.\ P(x) \Rightarrow f(x) = \sum_{i=0}^{n}(\frac{f^i\ 0}{i!} * x^i) + \delta_{t,n}(x)$

Here, f^i is the i-th derivative of f, and the approximation error $\delta_{f,n}(x)$ is soundly bounded from the remainder term of Taylor's theorem for input value x. Predicate P is a precondition constraining the interval on which function f can be approximated by the truncated series. When approximating an elementary function f by its truncated series, Dandelion always ensures that this precondition P is true: The series for *exp* requires inputs to be non-negative, the series for *ln* requires arguments greater than 1, and *arctan* requires arguments in $(-1, 1)$. The series for *sin* and *cos* have no preconditions.

At certificate checking time Dandelion automatically computes an upper bound to the approximation error $\delta_{t,n}(x)$. Since the second phase of Dandelion operates on polynomials, instead of truncated series, we prove once and for all a polynomial implementation of the truncated series from Theorem 5:

Theorem 6. $\forall n. \sum_{i=0}^{n}(\frac{f^i\ 0}{i!} * x^i) = t_{f,n}\ (x)$

Theorem 6 proves that t_n implements the truncated Taylor series on the left-hand side for an arbitrary number of approximation steps n. We prove versions of Theorem 6 for each elementary function supported by Dandelion. Finally, the proof of First Phase Soundness (Theorem 4) for a single elementary function is a simple combination of Theorems 5 and 6.

3.2 Approximations of More Complicated Expressions

Next, we explain how Dandelion uses truncated Taylor series to approximate more complicated elementary function expressions, using $exp(y * \frac{1}{2}) - 1$ on the interval $[1, 2]$ as an example. In

[4] Dandelion currently does not support *tan*, as a straight-forward reduction to *sin*(x)/*cos*(x) did not work out. We plan to incorporate a more direct series from HOL-Light in the future.

general, a Remez-like algorithm can return an approximation for a compound function or an expression, as long as it stays univariate. Compared to approximating individual functions, *e.g.* *exp*, an overall expression approximation can be more accurate, sometimes avoiding undesirable effects such as cancellation. Hence, Dandelion should also be able to certify those.

From Theorems 5 and 6 Dandelion knows how to automatically compute a polynomial approximation $t_{exp,n}$ and an approximation error $\delta_{exp,n}(x)$ for the exponential function for a given input range on the argument. In our example, the input argument is $y * \frac{1}{2}$, and thus the value of $\delta_{exp,n}(x)$ depends on the range of this expression, which Dandelion computes automatically using interval arithmetic [26].

As interval analysis, we reuse an existing HOL4 formalization [6], and extend it with range bounds for elementary functions. For our example Dandelion also needs to compute a range bound for $exp(y * \frac{1}{2})$. In general, because elementary functions are defined non-computably in HOL4, we have to rely on a trick to compute interval bounds. To compute interval bounds for elementary functions, Dandelion reuses our formalized truncated Taylor series. From Theorem 5 and Theorem 6, we derive for f that

$$|f(x) - t_{t,n}(x)| \leq \delta_{t,n} \tag{2}$$

From this inequality, we derive a bound on $f(x)$ in the interval $[a, b]$

$$t_{t,n}(a) \leq f(x) \leq t_{t,n}(b) + \delta_{t,n} \tag{3}$$

Equation 3 holds for monotone f only, and thus we cannot apply it to *sin* and *cos* as they are periodic. For both functions, interval analysis returns the closed interval $[-1, 1]$. Dandelion's interval analysis is proven sound once and for all in HOL4.

With the interval analysis, we can soundly compute a polynomial approximation for *exp*, $t_{exp,n}$ on the range of $y * \frac{1}{2}$. Dandelion automatically composes the polynomial $y * \frac{1}{2}$ with $t_{exp,n}$ to obtain a polynomial $q_{exp(y*\frac{1}{2}),n}$ with approximation error $\delta_{\exp,n}(x)$. However, we still need to come up with a polynomial approximation p and an approximation error for the full function $exp(y * \frac{1}{2}) - 1$. In our example, Dandelion treats the constant 1 as a polynomial returning 1, and automatically computes the polynomial difference of $q_{exp(\ldots),n}$ and $q_{1,n}$. The global approximation error δ for the difference of $q_{exp(\ldots),n}$ and $q_{1,n}$ depends on the approximation errors accumulated in both polynomials. In a final step, Dandelion automatically computes an upper bound on the global approximation error by propagating accumulated errors through the subtraction operation.

Generally, Dandelion implements an automatic approximation error analysis inside function `approxAsPoly` that propagates accumulated approximation errors. The propagation is implemented for basic arithmetic, and elementary functions to support *e.g.* expressions like $exp(x) + sin(x - 1)$.

Computing Propagation Errors for Sin and Cos

To accurately propagate approximation errors through *sin* and *cos*, our soundness proof assumes that the accumulated approximation error is contained in the interval $[0, \frac{\pi}{2}]$. This does not pose a true limitation of Dandelion as errors larger than $\pi/2$ would anyway be undesirable and impractical. For the correctness proof of `approxAsPoly` (Theorem 4), however, Dandelion must automatically prove that accumulated errors are less than or equal to $\pi/2$. This poses a challenge as in HOL4 π is defined non-computably using Hilbert's choice operator: if $0 \leq x \leq 2$ and $cos(x) = 0$, then π is $2 * x$. To solve this problem, we reuse

the truncated Taylor series of *arctan* and the fact that $arctan(1) = \pi/4$ to compute a lower bound r in HOL4, where $r \leq \pi$. At certificate checking time, when propagating the error $\delta_{f,n}$ through *sin* and *cos*, Dandelion checks $\delta_{f,n} \leq \frac{r}{2}$, which by transitivity proves that $\delta_{f,n} \leq \frac{\pi}{2}$.

3.3 Extending Dandelion's First Phase

All of the truncated Taylor series proven in Dandelion are for single applications of an elementary function. For a particular application it may be beneficial to add special cases to compute a single, more accurate, truncated Taylor series of an elementary function like $exp(sin(x))$ instead of computing a truncated series for each function separately.

In Harrison's original proof this would require manually redoing a large chunk of the proof work whereas for Dandelion such an extension amounts to 4 steps: Proving the truncated Taylor series as in Theorem 5, implementing and proving correct the polynomial $t_{f,n}$ as in Theorem 6, extending `approxAsPoly` with the special case for the new elementary function, and finally using the theorems proven for the first two steps to extend First Phase Soundness (Theorem 4) with a correctness proof for the new case. Complexity of the proofs only depends on the complexity of the series approximation. Dandelion then automatically uses the new series approximation whenever the approximated function is encountered in a certificate, and the global soundness result of Dandelion still holds without any required changes. The second phase directly benefits from adding additional approximations as more accurate Taylor series decrease the approximation error of the first phase.

4 Validating Polynomial Errors

For a certificate consisting of an elementary function `f`, polynomial approximation `p`, approximation error ε, input constraints `I`, and truncation steps `n`, the first phase of Dandelion computes a truncated Taylor series $q_{f,n}$ and an approximation error $\delta_{f,n}$, which is sound by Theorem 4. Both $q_{f,n}$ and `p` are polynomials, and following Harrison's terminology, we refer to their difference $q_{f,n}(x) - \mathtt{p}(x)$ as the error polynomial $h(x)$. In the second phase, Dandelion automatically finds an upper bound to the extremal values of $h(x)$ and compares this upper bound to the residual approximation error $\varepsilon - \delta_{f,n}$, which we refer to as γ. We prove soundness of the second phase once and for all as a HOL4 theorem:

Theorem 7 (Second Phase Soundness)**.**

$$\forall x.\ x \in \mathtt{I}(x) \Rightarrow |\, q_{f,n}(x) - \mathtt{p}(x) \,| \leq \gamma$$

Before going into the details of how Dandelion automatically validates the residual error γ, we quickly recall the key real analysis result which we rely on in this phase: on a closed interval $[a, b]$, a differentiable polynomial p can reach its extremal values at the outer points $p(a)$, $p(b)$, and the zeros of p's first derivative p' (Theorem 2). To find the extremal values of $h(x)$, Dandelion thus needs to automatically find *all* zeros of $h'(x)$.

Dandelion splits finding the extremal values and validating γ into three automated steps:

1. Compute the number of zeros using Sturm's theorem (Theorem 3 in Section 2)
2. Validate a guess of the zeros computed by an unverified, external oracle
3. Compute an upper bound on extremal values (using the validated zeros) and compare with γ

Conceptually, the second phase automates the main part of Harrison's manual proof, and the key step is computing Sturm sequences automatically in the first step. Next, we explain the ideas behind automating each of the steps.

```
sturm_seq (p, q, n) =
  if n = 0 then
    if (rm (p, (1/q[deg q]) * q) = 0 ∧ q <> 0) then SOME []
    else None
  else let g = - (rm (p, (1/q[deg q]) * q)) in
   if g = 0 ∧ ~ q = 0 then Some []
   else if (q = 0 ∨ (deg q < 3)) then None
   else case sturm_seq (q, g, n-1) of
     None => None
     |Some ss => Some (g::ss)
```

Figure 3 HOL4 definition of Sturm sequence computation

4.1 Bounding the Number of Zeros of a Polynomial

Dandelion bounds the number of zeros of a polynomial using Sturm's theorem (Theorem 3). A key challenge in developing this part of Dandelion was ensuring that the Sturm sequence is computable inside HOL4. In HOL-Light, Harrison defines Sturm sequences as a non-computable predicate `STURM` that existentially quantifies results, and thus can only be used to validate results in a manual proof.

In Figure 3, we show how Dandelion computes Sturm sequences. Function `rm (p,q)` computes the remainder of the polynomial division of `p` by `q`, `deg p` is the degree of polynomial `p`, and `q[n]` is the extraction of the `n`-th coefficient of `q`. As each polynomial division operation decreases the degree of the result by at least 1, the Sturm sequence for a polynomial p has a maximum length of $\deg(p) - 1$, as computation starts with p and its first derivative p'. Function `sturm_seq` is therefore initially run on polynomial $h'(x)$, $h''(x)$, and $\deg(h') - 1$.

If `sturm_seq(deg h'-1, h', h'')` returns list `sseq`, the complete Sturm sequence is `h'::h''::sseq`, and Dandelion computes the number of zeros of e' as its variation on the input range, based on Theorem 3.

We have proven once and for all that the results obtained from `sturm_seq(n, p', p'')` satisfy Harrison's non-computable predicate `STURM`. Thus we can reuse Harrison's proof of Sturm's theorem (Theorem 3). Harrison's Sturm sequences also use on a non-computable predicate for defining the result of polynomial division, and we prove it equivalent to a computable version in Dandelion, inspired by the one provided by Isabelle/HOL [3]. Ultimately, Dandelion uses these two equivalence proofs to reuse Harrison's proof of Sturm's theorem which we ported from HOL-Light.

4.2 Finding Zeros of Polynomials

Given the numbers of zeros nz for $h'(x)$, Dandelion next finds their values. As zero-finding is highly complicated even in non-verified settings, Dandelion uses an external oracle to come up with an initial guess of the zeros. These initial guesses are presented as a list of confidence intervals $[a, b]$, where $h'(x)$ has a zero between a and b. Further, the algorithm computing the confidence intervals need not be verified, as the result can easily be validated by Dandelion. To validate a list of guesses Z, Dandelion again relies on a result from real-number analysis, proven by Harrison: A function f has a zero at point x, if its first derivate f' changes sign at this point.

Dandelion validates the confidence intervals Z using a computable function that checks automatically for each element $[a, b]$ in Z, that $h''(a) * h''(b) \leq 0$, which is equivalent to a sign-change in the interval. If the number of zeros found is nz, Dandelion checks that this

sign change occurs at least nz times in Z.

4.3 Computing Extremal Values

In the final step, Dandelion uses the validated confidence intervals Z which contain all zeros of $h'(x)$ to compute an upper bound to the extremal values of $h(x)$. For interval $[a, b]$, Dandelion would ideally bound the error of $h(x)$ in $[a, b]$ as the maximum of $h(a)$, $h(b)$, and $h(y)$, where y is a zero of $h'(x)$. However, we have only confidence intervals for the zeros, and not their exact values available. Therefore, Dandelion's computation of an upper bound to $e(x)$ is more involved, and we base it on a theorem of Harrison. Harrison's theorem is a generalization of Theorem 2 for polynomial p, with derivative p', on interval $[a, b]$:

Theorem 8.

$$
\begin{aligned}
&(1)\quad (\forall x.\ a \le x \le b \wedge f'(x) = 0 \Rightarrow \exists (u, v).\ (u, v) \in Z \wedge u \le x \le v) \wedge \\
&(2)\quad (\forall x.\ a \le x \le b \Rightarrow |\, p'(x)\,| \le B) \wedge \\
&(3)\quad (\forall [u, v].\ [u, v] \in Z \Rightarrow a \le u \wedge v \le b \wedge |\, u - v\,| \le e \wedge |\, f(u)\,| \le K) \ \Rightarrow \\
&\qquad \forall x.\ a \le x \le b \Rightarrow |\, p(x)\,| \le \max(|\, f(a)\,|, |\, f(b)\,|, K + B * e)
\end{aligned}
$$

The theorem can be used to prove an upper bound on the error polynomial $h(x)$ which then can be compared to the residual error γ. For Dandelion, we automatically computed the values described by the assumptions to compute an overall bound on $h(x)$. We implement this computations in a function `validateErr`, and explain each of its computation steps on a high-level, based on the assumptions of Theorem 8.

The first assumption (1) from Theorem 8 states that the confidence intervals in Z contain only valid zeros and has been established automatically by the previous step. Based on assumption (2), Dandelion computes B by evaluating $|h'(x)|$ on $\max(|a|, |b|)$. Following assumption (3), Dandelion computes K as the maximum value of evaluating the error polynomial h on the lower bounds of the confidence intervals in Z, and a value e as the maximum value of $|u - v|$ for each $[u, v]$ in Z. Dandelion then computes the overall bound on the error polynomial h as $\max(h(a), h(b), K + B * e)$. To validate the residual error γ, it then suffices to check $\max(h(a), h(b), K + B * e) \le \gamma$.

Overall, we prove once and for all soundness of Dandelion as

Theorem 9 (Dandelion Soundness)**.**

$$\texttt{Dandelion(f,p,I,}\varepsilon\texttt{,n)} = \texttt{true} \ \Rightarrow\ (\forall x.\ x \in \texttt{I} \Rightarrow |\texttt{f}(x) - \texttt{p}(x)| \le \varepsilon)$$

The proof of Theorem 9 uses the triangle inequality to combine the theorem First Phase Soundness (Theorem 4) with the theorem Second Phase Soundness (Theorem 7).

5 Extracting a Verified Binary with CakeML

Computations performed in interactive theorem provers are known to be slower than those in unverified languages. To alleviate this performance problem, the proof-producing synthesis [4] implemented in the CakeML verified compiler [35] translates HOL4 functions into their CakeML counterpart, with an equivalence proof. These translated CakeML functions are compiled into machine code with the CakeML compiler, and as CakeML is fully verified, the machine code enjoys the same correctness guarantees as its HOL4 version.

During an initial test run we noticed that the Sturm sequence computations in the second phase are the most computation intensive task of Dandelion. HOL4 represents real-numbers

as (reduced) fractions during computation, and we noticed that in Dandelion their size still grew quite large, leading to a single multiplication taking up to 6 hours. Therefore, we use CakeML's proof-producing synthesis to extract a verified binary for the computations described in Section 4. To communicate results of the first phase with the binary, we implemented an (albeit unverified) lexer and parser that reads-in results from the first phase.

6 Evaluation

We have described how Dandelion automatically validates polynomial approximations from Remez-like algorithms. Next, we demonstrate Dandelion's usefulness with three separate experiments, by demonstrating that Dandelion fully automatically

1. validates certificates generated with an off-the-shelf Remez-like algorithm (Section 6.1)
2. validates certificates for more complicated elementary function expressions (Section 6.2)
3. validates certificates for less-accurate techniques (Section 6.3)

All the results we report in this section where gathered on a machine running Ubuntu 20.04, with an 2.7GHz i7 core and 16 GB of RAM. All runnning times are measured using the UNIX `time` command as elapsed wall-clock time in seconds.

6.1 Validating Certificates of a Remez-like Algorithm

In our first evaluation, we show that Dandelion certifies accurate approximation errors from an off-the-shelf Remez-like algorithm. We generate certificates by combining the Daisy static analyzer with the Sollya approximation tool [10], and extend Daisy with a simple pass that replaces calls to elementary functions with an approximation computed by Sollya. As a Remez-like algorithm, we use the `fpminimax` [9] function in Sollya. Our pipeline is benchmarked on numerical kernels taken from the FPBench benchmark suite [13], and the benchmarks used by an unverified extension of Daisy with approximations for elementary functions [21]. These benchmarks represent kernels as they occur in *e.g.* embedded systems, and thus they benefit from custom polynomial approximations. The original work by Izycheva *et al.* [21] synthesizes polynomial approximations whose target error bounds are usually larger than those inferred by Sollya. Hence, Dandelion could validate Daisy's bounds as well, but for the sake of the evaluation we choose more challenging, tighter bounds. For each benchmark, Daisy creates a certificate for each approximated elementary function, amounting to a total of 96 generated certificates.

Sollya's implementation of `fpminimax` can be configured to use different degrees for the generated approximation and different formats for the coefficients of the approximation. In our evaluation we approximate elementary functions with a degree 5 polynomial, storing the coefficients with a precision of 53 bits. All input ranges used in the certificates are computed by Daisy without modifying them, except for *exp*, where we disallow negative intervals, *i.e.* if Daisy wanted to approximate on $[-x, y]$, we change it to $[0, y]$ as Dandelion currently does not support negative exponentials. This can be fixed by straight-forward range reductions that are independent to the approximations computed by Remez-like algorithms. For each such approximation, Daisy creates a certificate to be checked by Dandelion.

The MetiTarski automated theorem prover [33] is a tool that provides the same level of automation as Dandelion, albeit not being fully verified. In general, MetiTarski checks real-number inequalities that may contain elementary functions, thus we compare the number of certificates validated by Dandelion to those that can be checked by MetiTarski.

Our results are given in Table 1. The left-most column of Table 1, contains the name of the elementary function approximated by Daisy, and the second column, labeled with a #

Function	#	Dandelion			MetiTarski	
		Verified	HOL4(s)	Binary(s)	Verified	Time(s)
atan	2	1	11.62	20.36	2	6.80
cos	28	25	202.35	251.33	25	3.15
exp	21	18	39.58	212.54	10	5.35
ln	8	0	X	X	5	4.99
sin	31	27	17.83	295.66	25	6.32
Total	96	71			67	

Table 1 Overview of certificaes generated by Daisy and validated with Dandelion and MetiTarski

contains the number of certificates generated for the elementary function, with unique input ranges. The next three columns, headed "Dandelion", contain the number of certificates validated by Dandelion, the average HOL4 running time for the first phase in seconds, and the average running time of the binary for the second phase in seconds. The final two columns, headed "MetiTarski", contain the number of certificates validated by MetiTarski, and the average running time in seconds.

Our evaluation truncates Taylor series after 32 terms in `approxAsPoly`. In general, we found six times the degree of the computed approximation to be a good estimate for when to truncate Taylor series in Dandelion. Our use of 32 instead of 30 is a technical detail, as some Taylor series require both the number of series terms n, as well as $\frac{n}{2}$ to be even. In general, the number of series terms has to be significantly higher than the degree of the approximated polynomial, to make the approximation error of the first phase almost negligible.

Overall, we notice that Dandelion can validate more certificates than MetiTarski. However, MetiTarski as a non-formally verified tool validates certificates that are currently out of reach for Dandelion. This is mostly due to the first phase of Dandelion. Even though we used general, widely known truncated Taylor series for all supported elementary functions, Dandelion fails to validate certificates for the *ln* function. We have inspected the generated certificates, and Dandelion cannot compute a high-accuracy polynomial approximation for 5 of them because they do not satisfy the precondition of Dandelion's Taylor series. For the remaining 3 certificates, we ran into issues with Sollya's computation of the confidence intervals for the zeros. On a high-level, the problem originates from the derivative of the error polynomial $h(x)$ being very close to 0, leading to a huge number of zeros being found, *i.e.* computation not terminating within a reasonable amount of time. To demonstrate that Dandelion still certifies errors for the *ln* function, we add an example in Section 6.2.

While Dandelion cannot certify errors for the *ln* function in this part of the evalution, this is not a conceptual limitation, as polynomial approximations are commonly paired with an argument reduction strategy. While verification of these strategies is orthogonal to validating results of an Remez-like algorithm, they could be used to reduce the input range of the approximated elementary function into a range that Dandelion can certify. More generally, we have done the heavy lifting of automating the computations and implementing the general framework, such that adding more accurate Taylor series to Dandelion amounts to mere proof engineering, modulo coming up with Taylor series in the first place. For the certificates for *arctan*, *cos*, *sin*, and *exp*, we notice that the average running time is in the order of minutes, making certificate checking with Dandelion's verified binary feasible.

We compare the approximation errors recorded in the certificates with the infinity norm computed by Sollya, which is the most-accurate estimate of the approximation error [11].

Function	Range	Deg.	Prec.	∞-norm	Error	HOL4	Binary
$\cos(x+1)$	$[0, 2.14]$	5	53	3.06E-5	3.06E-5	169	63
$\sin(x-2)$	$[-1, 3.00]$	5	53	2.05E-3	2.91E-3	93	68
$\ln(x+\frac{1}{10})$	$[1.001, 1.1]$	3	32	1.08E-7	1.08E-7	2775	1773
$\exp(x*\frac{1}{2})+\cos(x*\frac{1}{2})$	$[0.1, 1.00]$	5	53	2.03E-9	4.45E-9	711	5
$\arctan(x)-\cos(\frac{3}{4}*x)$	$[-0.5, 0.5]$	5	53	1.18E-5	1.18E-5	24	2308

Table 2 Functions approximated with Sollya using `fpminimax` and certified with Dandelion

Overall, Dandelion certifies an approximation error in the same order of magnitude as the infinity norm for 61 certificates. For the remaining 10, the error is a sound upper bound. In general, infinity norm-based estimates are known to be the most accurate and their verification requires more elaborate techniques than Sturm sequences [11]. Consequently we would not expect Dandelion to be able to always certify infinity norms.

We ran the evaluation for an approximation degree of 3, with precisions of 53 and 23 each to measure the influence of those parameters. Overall, the running time significantly decreases when decreasing from degree 5 to 3, going from average running times of minutes to average running times of seconds. Decreasing the precision of the coefficients further speeds up evaluation, though not as significant as decreasing the degree did. This suggests that higher coefficient accuracies can easily be used for generating polynomials with Remez-like algorithms, and lower degree polynomials should be preferred for fast validation.

6.2 Validating Certificates for Elementary Function Expressions

Next, we show that Dandelion can also certify approximation errors for complicated elementary function expressions. We validate with Dandelion approximation errors for random examples involving elementary functions and arithmetic. Polynomial approximations are again generated by Sollya.

An overview of our results is given in Table 2. The table shows the approximatied function, then Sollya's parameters (the input range, the target degree (Deg.), the target precision (Prec.)), and then the infinity-norm (∞-norm) of the approximation. The final columns summarize the Dandelion results, giving the certified approximation error, and the running time in seconds of the first phase (HOL4) and the second phase (Binary).

Overall, we notice that the certified approximation error is on the same order of magnitude as the infinity norm for all examples. We also notice that performance for both phases varies across the different examples. For the first phase this is often due to how the input ranges are encoded in the certificate. We noticed that HOL4 is very sensitive to how the fractions representing real numbers are encoded when performing computations. Similarly, performance of the second phase greatly varies depending on the complexity of the error polynomial computed by the first phase. We observe that perfomance improves with both smaller degree polynomials, and smaller representations of the polynomial coefficients.

The results in Table 2 exclude examples where two elementary functions are composed with each other, *e.g.* as in $exp(sin(x-1))$. This is because Dandelion computes the global high-accuracy approximation in the first phase via polynomial composition of an approximation for *exp* and *sin*. While this is theoretically supported by Dandelion, we found that the polynomial composition leads to an exponential blow-up in the degree of the error polynomial. Even for the innocuously looking example $exp(sin(x-1))$, the second phase could not validate a polynomial approximation within 24 hours. This clearly motivates the

Function	Range	Deg.	Prec.	∞-norm	Error	HOL4	Binary
$\cos(x)$	$[0, 2.14]$	5	53	3.17E-7	3.22E-7	169	63
$\sin(x+2)$	$[-1.5, 1.5]$	5	53	4.47E-4	7.60E-4	142	138
$\sin(3*x) + \exp(x * \frac{1}{2})$	$[0, 1]$	3	53	2.45E-2	2.48E-2	54	1897
$\exp(x) - 1^*$	$[0.003, 0.01]$	3			$2^{-33.2}$	133	<1

Table 3 Chebyshev approximations certified with Dandelion

use of more elaborate Taylor series if compound elementary functions need to be certified by Dandelion. In general, settings where elementary functions like those in Table 2 are used could potentially be made more accurate and be validated faster with custom Taylor series.

6.3 Validating Certificates for Simpler Approximation Algorithms

Remez-like algorithms are known to be the most accurate approximation algorithms. However, less accurate approaches are still in use today, and as such interesting targets for verification. Bréhard *et al.* [8] certify Chebyshev approximations in the Coq theorem prover, where their approach requires some manual proofs. We demonstrate that Dandelion also certifies Chebyshev approximations on some random examples by computing Chebyshev approximations with Sollya's function `chebyshevform`. The results are shown in Table 3.

Again, we first give Solya's parameter and the infinity norm, then we give the error certified by Dandelion, and the execution times for the first and second phase.

We also include the approximation certified by Harrison [19], labeled with a *. The polynomial has degree 3, but we leave the precision empty as it is not generated by Sollya, and we do not provide an infinity norm. The only difference to the proof from Harrison is that we prove the bound only for positive x, as Dandelion currently does not handle exponentials on negative values. The lower bound of the range is 0.003, instead of 0 to rule out a 0 on the lower bound (as $\exp(x) - 1 = 0$ for $x = 0$), which we must exclude by Theorem 8. Harrison's manual proof of the polynomial approximation then reduces to a single line running Dandelion on the encoding.

7 Related Work

Throughout the paper, we have already hinted at the immediate related work. Next, we explain the key conceptual differences between Dandelion and the immediate related work and put Dandelion into the greater context. In general, Dandelion touches upon two key research areas in interactive and automated theorem proving: techniques for approximating elementary functions and techniques for proving theorems involving real-numbered functions.

Approximating Elementary Functions The work on approximating elementary functions can be distinguished among two axes: whether or not the work provides rigorous machine-checked proofs, and whether the work is fully automated or requires user intervention.

Fully automated, rigorous machine-checked proofs, similar to Dandelion are provided by the work by Bréhard *et al.* [8]. They develop a framework for proving correct Chebychev approximations of real number functions in the Coq theorem prover [1]. Also in Coq, Martin-Dorel and Melquiond [25] verify polynomial approximations using the CoqInterval [24] package. They develop a fully automated tactic for proving approximations inside floating-point mathematical libraries correct. A key difference between Dandelion and both these

tools is that they cannot certify approximations computed by Remez-like algorithms, which can in general provide more accurate approximations.

For manual proofs, versions of the exponential function have been verified by Harrison [18], and Akbarpour *et al.* [5]. The manual proof by Harrison [19] layed out the foundations for Dandelion. The work has also been extended by Chevillard *et al.* [11]. Instead of verifying approximation errors for polynomials, they use so-called sum-of-squares decompositions [20] to certify infinity norm computations. A major limiting factor for their work was finding accurate enough Taylor polynomials which we found to not be a major issue for our approach.

Coward *et al.* [12] use the MetiTarski automated theorem prover [33] to verify accuracy of hardware finite-precision implementations of elementary functions. While as automated as Dandelion, MetiTarski does not provide rigorous machine-checked proofs. A major conceptual difference is that their MetiTarski verification reasons about bit-level accuracy of the hardware implementation, while Dandelion reasons about real-number functions and polynomials. Together with a verified roundoff error analysis like FloVer [6], Dandelion could be extended to verify finite-precision implementations of elementary functions, and together with Daisy [21] verification could possibly be lifted to entire arithmetic kernels.

A different style of unverified approximations is provided by Lim *et al.* [23]. Instead of computing specialized polynomial approximations, they focus on correctly-rounded, general purpose approximations. These approximations are not build for specific use-cases, but should rather be seen as replacements for the functions provided in mathematical libraries. At the time of writing, their approach is not formally verified, but they do provide a pen-and-paper correctness argument for their code generation. The CR-libm [14] library also provides unverified alternatives of correctly rounded mathematical libraries, and Muller [28] gives a general overview of the techniques for implementing elementary functions.

(Automated) Real-Number Theorem Proving Dandelion heavily relies on HOL4's support for real-number theorem proving. Below we list some alternatives for proving properties of real-numbers in both interactive and automated theorem proving systems. In the HOL-family of ITP systems, Harrison [20] has formalized sum-of-squares certificates for the HOL-Light [2] theorem prover. His approach relies on semidefinite programming to find a decomposition of a polynomial into a sum-of-squares polynomial. Both Isabelle/HOL and PVS have been independently extended with implementations of Sturm sequences [16, 30, 31]. Their main focus is not on verification of polynomial approximations, they rather use Sturm sequences to prove properties about roots of polynomials, non-negativity, and monotonicity.

Previously we have already mentioned the MetiTarski automated theorem prover [33], as an example of an automated theorem prover for real-numbered functions. However, MetiTarski is not the only automated prover for real-numbered functions. Real-numbered functions are also supported by *e.g.* dReal [17], and z3's SMT theory for real-numbers [27].

8 Conclusion

We have presented Dandelion, the first verified and fully automated certificate checker for polynomial approximations of elementary functions computed with Remez-like algorithms. Dandelion splits the validation task into two clearly separated phases: The first phase replaces elementary functions by high-accuray Taylor series, and the second phase uses Sturm's theorem and an external oracle to validate the approximation error. Our evaluation has shown that Dandelion certifies approximation errors computed by an off-the-shelf Remez-like algorithm, and Dandelion also certifies approximation errors for Chebyshev approximations.

References

1 The Coq Proof Assistant. URL: https://coq.inria.fr.

2 The HOL-Light Proof Assistant. URL: https://www.cl.cam.ac.uk/~jrh13/hol-light/.

3 The Isabelle/HOL Proof Assistant. URL: https://isabelle.in.tum.de/.

4 Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. Proof-Producing Synthesis of CakeML from Monadic HOL Functions. *Journal of Automated Reasoning (JAR)*, 64(7), 2020. doi:10.1007/s10817-020-09559-8.

5 Behzad Akbarpour, Amr Abdel-Hamid, Sofiène Tahar, and John Harrison. Verifying a Synthesized Implementation of IEEE-754 Floating-Point Exponential Function using HOL. *The Computer Journal*, 53:465–488, 05 2010. doi:10.1093/comjnl/bxp023.

6 Heiko Becker, Nikita Zyuzin, Raphaël Monat, Eva Darulova, Magnus O Myreen, and Anthony Fox. A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10. IEEE, 2018.

7 Hans-J. Boehm. Towards an API for the Real Numbers. In *Programming Language Design and Implementation (PLDI)*, 2020. doi:10.1145/3385412.3386037.

8 Florent Bréhard, Assia Mahboubi, and Damien Pous. A Certificate-Based Approach to Formally Verified Approximations. In *Conference on Interactive Theorem Proving (ITP)*, pages 1–19, 2019.

9 Nicolas Brisebarre and Sylvain Chevillard. Efficient polynomial L-approximations. In *IEEE Symposium on Computer Arithmetic (ARITH)*, pages 169–176. IEEE, 2007.

10 S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An Environment for the Development of Numerical Codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

11 Sylvain Chevillard, John Harrison, Mioara Joldeş, and Ch Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoretical Computer Science*, 412(16):1523–1543, 2011.

12 Samuel Coward, Lawrence Paulson, Theo Drane, and Emiliano Morini. Formal Verification of Transcendental Fixed and Floating Point Algorithms using an Automatic Theorem Prover. *Formal Aspects of Computing (in press)*, 2022.

13 Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In *Numerical Software Verification (NSV)*, 2016. doi:10.1007/978-3-319-54292-8_6.

14 Catherine Daramy, David Defour, Florent de Dinechin, and Jean-Michel Muller. CR-LIBM: a correctly rounded elementary function library. In *Advanced Signal Processing Algorithms, Architectures, and Implementations*, volume 5205, pages 458–464. International Society for Optics and Photonics, 2003.

15 Eva Darulova and Anastasia Volkova. Sound Approximation of Programs with Elementary Functions. In *Computer Aided Verification (CAV)*, 2019. doi:10.1007/978-3-030-25543-5_11.

16 Manuel Eberl. A Decision Procedure for Univariate Real Polynomials in Isabelle/HOL. In *Certified Programs and Proofs (CPP)*, page 75–83, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676724.2693166.

17 Sicun Gao, Soonho Kong, and Edmund M Clarke. dReal: An SMT solver for nonlinear theories over the reals. In *International Conference on Automated Deduction*, pages 208–214. Springer, 2013.

18 John Harrison. "floating point verification in hol light: The exponential function". In Michael Johnson, editor, *Algebraic Methodology and Software Technology*, pages 246–260, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

19 John Harrison. Verifying the accuracy of polynomial approximations in HOL. In *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 137–152. Springer, 1997.

20 John Harrison. Verifying nonlinear real formulas via sums of squares. In Klaus Schneider and Jens Brandt, editors, *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 4732 of *Lecture Notes in Computer Science*, pages 102–118, Kaiserslautern, Germany, 2007. Springer-Verlag.

21 Anastasiia Izycheva, Eva Darulova, and Helmut Seidl. Synthesizing efficient low-precision kernels. In *Automated Technology for Verification and Analysis (ATVA)*, pages 294–313. Springer, 2019.

22 Olga Kupriianova and Christoph Lauter. Metalibm: A mathematical functions code generator. In *International Congress on Mathematical Software*, pages 713–717. Springer, 2014.

23 Jay P Lim and Santosh Nagarakatte. One Polynomial Approximation to Produce Correctly Rounded Results of an Elementary Function for Multiple Representations and Rounding Modes. *Proceedings of the ACM on Programming Languages (POPL)*, 6, 2022.

24 Érik Martin-Dorel and Guillaume Melquiond. CoqInterval: A Toolbox for Proving Nonlinear Univariate Inequalities in Coq. In *Conference on Effective Analysis: Foundations, Implementations, Certification*, 2016.

25 Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning (JAR)*, 57(3):187–217, 2016.

26 R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

27 Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008.

28 Jean-Michel Muller. *Elementary Functions*. Springer, 2006.

29 César Muñoz, Anthony Narkawicz, George Hagen, Jason Upchurch, Aaron Dutle, María Consiglio, and James Chamberlain. DAIDALUS: detect and avoid alerting logic for unmanned systems. In *Digital Avionics Systems Conference (DASC)*, pages 5A1–1. IEEE, 2015.

30 Anthony Narkawicz, César Munoz, and Aaron Dutle. Formally-verified decision procedures for univariate polynomial computation based on sturm's and tarski's theorems. *Journal of Automated Reasoning (JAR)*, 54(4):285–326, 2015.

31 Anthony Narkawicz, Cesar Munoz, and Aaron Dutle. A decision procedure for univariate polynomial systems based on root counting and interval subdivision. *Journal of Formalized Reasoning*, 11(1):19, 2018.

32 Ricardo Pachón and Lloyd N Trefethen. Barycentric-Remez algorithms for best polynomial approximation in the chebfun system. *BIT Numerical Mathematics*, 49(4):721, 2009.

33 Lawrence C Paulson. Automated Theorem Proving for Special Functions: The Next Phase. In *Symposium on Symbolic-Numeric Computation*, pages 3–8, 2014.

34 Konrad Slind and Michael Norrish. A Brief Overview of HOL4. In *Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 28–32, 2008. doi:10.1007/978-3-540-71067-7_6.

35 Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming (JFP)*, 29, 2019. URL: jfp19.pdf, doi:10.1017/S0956796818000229.

36 Ping-Tak Peter Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software (TOMS)*, 15(2):144–157, 1989.