Theoretical Computer Science 412 (2011) 1523–1543

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Efficient and accurate computation of upper bounds of approximation errors

S. Chevillard [a,*], J. Harrison [b], M. Joldeş [c], Ch. Lauter [d]

[a] INRIA, LORIA, Caramel Project-Team, BP 239, 54506 Vandœuvre-lès-Nancy Cedex, France
[b] Intel Corporation, 2111 NE 25th Avenue, M/S JF1-13, Hillsboro, OR, 97124, USA
[c] École Normale Supérieure de Lyon, Université de Lyon, Arénaire, LIP (UMR 5668 CNRS - ENS Lyon - INRIA - UCBL), 46 allée d'Italie, 69364 Lyon Cedex 07, France
[d] Université Pierre et Marie Curie Paris VI (CNRS, UMR 7606, LIP6), 4 place Jussieu, 75252 Paris Cedex 05, France[1]

ARTICLE INFO

Keywords:
Supremum norm
Approximation error
Taylor models
Sum of squares
Validation
Certification
Formal proof

ABSTRACT

For purposes of actual evaluation, mathematical functions f are commonly replaced by approximation polynomials p. Examples include floating-point implementations of elementary functions, quadrature or more theoretical proof work involving transcendental functions.

Replacing f by p induces a relative error $\epsilon = p/f - 1$. In order to ensure the validity of the use of p instead of f, the maximum error, i.e. the supremum norm $\|\epsilon\|_\infty^I$ must be safely bounded above over an interval I, whose width is typically of order 1.

Numerical algorithms for supremum norms are efficient, but they cannot offer the required safety. Previous validated approaches often require tedious manual intervention. If they are automated, they have several drawbacks, such as the lack of quality guarantees.

In this article, a novel, automated supremum norm algorithm on univariate approximation errors ϵ is proposed, achieving an *a priori* quality on the result. It focuses on the validation step and paves the way for formally certified supremum norms.

Key elements are the use of intermediate approximation polynomials with bounded approximation error and a non-negativity test based on a sum-of-squares expression of polynomials.

The new algorithm was implemented in the Sollya tool. The article includes experimental results on real-life examples.

1. Introduction

Replacing functions by polynomials to ease computations is a widespread technique in mathematics. For instance, handbooks of mathematical functions [1] give not only classical properties of functions, but also convenient polynomial and rational approximations that can be used to approximately evaluate them. These tabulated polynomials have proved to be very useful in the everyday work of scientists.

Nowadays, computers are commonly used for computing numerical approximations of functions. Elementary functions, such as exp, sin, arccos, erf, etc., are usually implemented in libraries called `libms`. Such libraries are available on most

* Corresponding author.
E-mail addresses: sylvain.chevillard@ens-lyon.org (S. Chevillard), johnh@ichips.intel.com (J. Harrison), mioara.joldes@ens-lyon.fr (M. Joldeş), christoph.lauter@lip6.fr (Ch. Lauter).

[1] Ch. Lauter is now with UPMC Paris VI but was with Intel Corporation when this research work was done.

0304-3975/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2010.11.052

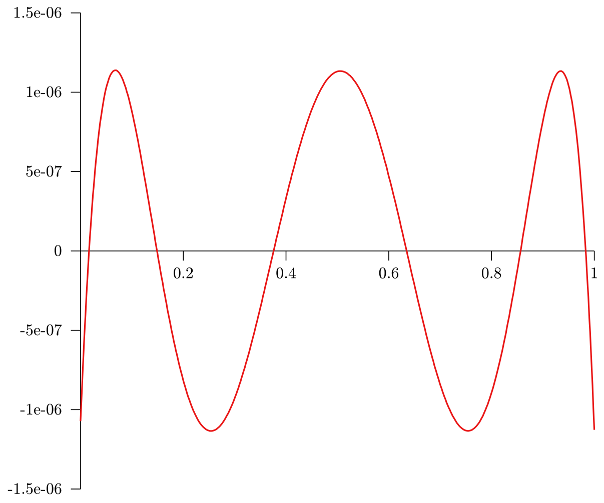

Fig. 1. Approximation error in a case that is typical for a `libm`.

systems, and many numerical programs depend on them. Examples include `CRlibm`, `glibc`, Sun[2] `libmcr` and the Intel® `libm` available with the Intel® Professional Edition Compilers and other Intel® Software Products.

When writing handbooks as well as when implementing such functions in a `libm`, it is important to rigorously bound the error between the polynomial approximation p and the function f. In particular, regarding the development of `libm`s, the IEEE 754-2008 standard [2] recommends that the functions be correctly rounded.

Currently, most `libm`s offer strong guarantees: they are made with care and pass many tests before being published. However, in the core of `libm`s, the error between polynomial approximations and functions is often only estimated with a numerical application such as Maple that supports arbitrary-precision computations. As good as this numerical estimation could be, it is not a mathematical proof. As argued by the promoters of correctly rounded transcendentals, if a library claims to provide correctly rounded results, its implementation should be mathematically proven down to the smallest details, because necessary validation cannot be achieved through mere testing [3].

Given f the function to be approximated and p the approximation polynomial used, the approximation error is given by $\varepsilon(x) = p(x)/f(x) - 1$ or $\varepsilon(x) = p(x) - f(x)$, depending on whether the relative or absolute error is considered. This function is often very regular: in Fig. 1 the approximation error is plotted in a typical case when a minimax approximation polynomial of degree 5 is used [4, Chapter 3].

A numerical algorithm tells us that the maximal absolute value of ε in this case (Fig. 1) is approximately $1.1385 \cdot 10^{-6}$. But can we guarantee that this value is actually greater than the real maximal error, i.e. an upper bound for the error due to the approximation of f by p? This is the problem we address in this article. More precisely, we present an algorithm for computing a tight interval $\boldsymbol{r} = [\ell, u]$, such that $\|\varepsilon\|_\infty^I \in \boldsymbol{r}$. Here, $\|\varepsilon\|_\infty^I$ denotes the infinity or supremum norm over the interval I, defined by $\|\varepsilon\|_\infty^I = \sup_{x \in I}\{|\varepsilon(x)|\}$. Although several previous approaches exist for bounding $\|\varepsilon\|_\infty^I$, this problem does not have a completely satisfying solution at the moment. In what follows, we give an overview of all the features needed for and achieved by our algorithm.

i. The algorithm is fully automated. In practice, a `libm` contains many functions. Each of them contains one or more approximation polynomials whose error must be bounded. It is frequent that new `libm`s are developed when new hardware features are available. So our problem should be solved as automatically as possible and should not rely on manual computations.
ii. The algorithm handles not only simple cases when f is a basic function (such as exp, arcsin, tan, etc.) but also more complicated cases when f is obtained as a composition of basic functions, such as $\exp(1 + \cos(x)^2)$, for instance. Besides the obvious interest of having an algorithm as general as possible, this is necessary even for implementing simple functions in `libm`s. Indeed, it is usual to replace the function to be implemented, f, by another one, g, in a so-called range reduction process [5, Chapter 11]. The value $f(x)$ is in fact computed from $g(x)$. So, eventually, the function approximated by a polynomial is g. This function is sometimes given as a composition of several basic functions.

   In consequence, the algorithm should accept as input any function f defined by an expression. The expression is made using basic functions such as exp or cos. The precise list of basic functions is not important for our purpose: we can consider the list of functions defined in software tools like Maple or Sollya [6], for instance. The only requirement for basic functions is that they be differentiable up to a sufficiently high order.

[2] Other names and brands may be claimed as the property of others.

iii. The algorithm should be able to automatically handle a particular difficulty that frequently arises when considering relative errors $\varepsilon(x) = p(x)/f(x) - 1$ in the process of developing functions for `libms`: the problem of removable discontinuities. If the function to be implemented, f, vanishes at a point x_0 in the interval considered, in general, the approximation polynomial is designed such that it vanishes also at the same point with a multiplicity large enough. Hence, although f vanishes, ε may be defined by continuous extension at such points x_0, called removable discontinuities. For example, if p is a polynomial of the form $x\,q(x)$, the function $p(x)/\sin(x) - 1$ has a removable discontinuity at $x_0 = 0$. Our algorithm can handle removable discontinuities in all practical cases.

iv. The accuracy obtained for the supremum norm is controlled *a priori* by the user through a simple input parameter $\overline{\eta}$. This parameter controls the relative tightness of $\boldsymbol{r} = [\ell,\ u]$: this means that the algorithm ensures that eventually $0 \leq \frac{u-\ell}{\ell} \leq \overline{\eta}$. This parameter can be chosen as small as desired by the user: if the interval $\boldsymbol{r}$ is returned, it is guaranteed to contain $\|\varepsilon\|_\infty^I$ and to satisfy the required quality. In some rare cases, roughly speaking if the function is too complicated, our algorithm will simply fail, but it never lies. In our implementation, all the computations performed are rigorous, and a multiprecision interval arithmetic library [7] is used. We have conducted many experiments challenging our algorithm, and in practice it never failed.

v. Since complicated algorithms are used, their implementation could contain some bugs. Hence, beside the numerical result, the algorithm should return also a formal proof. This proof can be automatically checked by a computer and gives a high guarantee on the result. Currently, this formal proof is not complete, but generating a complete formal proof is essentially just a matter of implementation.

1.1. *Outline of the paper*

In the next section, we explain the main ideas of previous approaches that partially fulfilled these goals and analyze their deficiencies. In Section 3, our algorithm is presented. It provides all the features presented above. As we will see, the algorithm relies on automatically computing an intermediate polynomial: Section 4 discusses several existing methods that can be used for computing this intermediate polynomial and we compare their practical results. These methods are not able to handle removable discontinuities; however, we managed to adapt one of the methods for solving this issue. Our modified method is presented in Section 4.4. In Section 5, we explain how a formal proof can be generated by the algorithm and checked by the HOL Light proof checker.[3] Finally, in Section 6, we show how our new algorithm behaves on real-life examples, and compare its results with the ones given by previous approaches.

2. Previous work

2.1. *Numerical methods for supremum norms*

First, one can consider a simple numerical computation of the supremum norm. In fact, we can reduce our problem to searching for extrema of the error function. These extrema are usually found by searching for the zeros of the derivative of the error function. Well-known numerical algorithms like bisection, Newton's algorithm or the secant algorithm can be used [8]. These techniques offer a good and fast estimation of the needed bound, and implementations are available in most numerical software tools, like Maple or Matlab.

Roughly speaking, all the numerical techniques finish by exhibiting a point x, more or less near to a point $x^\star$ where ε has a global extremum. Moreover, the convergence of these techniques is generally very fast (quadratic in the case of Newton's algorithm [8]). Improving the accuracy of x with respect to $x^\star$ directly improves the accuracy of $\varepsilon(x)$ as an approximation of the global optimum.

Moreover, it is possible to get a safe lower bound, ℓ, on $|\varepsilon(x)|$ by evaluating $|\varepsilon|$ over the point interval $[x,\ x]$ with interval arithmetic. This bound can be made arbitrarily tight by increasing the working precision. This can be achieved using multiple-precision interval arithmetic libraries, like for example the MPFI Library [7].

Hence, we assume that a numerical procedure `computeLowerBound` is available that can compute a rigorous lower bound $\ell \leq \|\varepsilon\|_\infty$ with arbitrary precision. More formally, it takes as input a parameter γ that heuristically controls the accuracy of ℓ. The internal parameters of the numerical algorithm (e.g. the number of steps in Newton's iteration, the precision used for the computations, etc.) are heuristically adjusted in order to be fairly confident that the relative error between ℓ and $\|\varepsilon\|_\infty$ is less than γ. However, one cannot verify the actual accuracy obtained. Hence such methods do not provide any mean of computing a tight *upper bound*, and are not sufficient in our case.

2.2. *Rigorous global optimization methods using interval arithmetic*

It is important to remark that obtaining a tight upper bound for $\|\varepsilon\|_\infty$ is equivalent to rigorously solving a univariate global optimization problem. This question has already been extensively studied in the literature [9–11]. These methods are based on a general interval branch-and-bound algorithm, involving an exhaustive search over the initial interval. This interval is subdivided recursively ("branching"), and those subintervals that cannot possibly contain global optima are

[3] http://www.cl.cam.ac.uk/~jrh13/hol-light/.

rejected. The rejection tests are based on using interval arithmetic for bounding the image of a function. Many variants for accelerating the rejection process have been implemented. They usually exploit information about derivatives (most commonly first and second derivatives) of the function. One example is the interval Newton method [12], used for rigorously enclosing all the zeros of a univariate function.

Dependency problem for approximation errors. While the above-mentioned methods can be successfully used in general, when trying to solve our problem, one is faced with the so-called "dependency phenomenon" [13]. Roughly speaking, it is due to the fact that multiple occurrences of the same variable are not exploited by interval arithmetic. The computations are performed "blindly", taking into account only the range of a variable independently for each occurrence. In our case, the dependency phenomenon stems from the subtraction present in the approximation error function $\varepsilon = p - f$. In short, when using interval arithmetic, the correlation between f and p is lost. This means that although f and p have very close values, interval arithmetic cannot benefit from this fact and will compute an enclosure of ε as the difference of two separate interval enclosures for f and p over the given interval. Even if we could obtain exact enclosures for both f and p over the interval considered, the interval difference would be affected by overestimation. It can be shown (see, e.g., Section 3.2.4 of [14]) that in order to obtain a small overestimation for its image, we need to evaluate ε over intervals of the size of $\|\varepsilon\|_\infty$. In some specific cases used in the process of obtaining correctly rounded functions, this is around 2^{-120}. This means that a regular global optimization algorithm would have to subdivide the initial interval into an unfeasible number of narrow intervals (for example, if the initial interval is $[0, 1]$, 2^{120} intervals have to be obtained) before the actual algorithm is able to suppress some that do not contain the global optima.

In fact, in [15], authors of the present article tried to use a similar recursive technique, but they observed that the derivative of the error, the second derivative of the error, and so on and so forth, are all prone to the same phenomenon of very high overestimation. That is why, for higher degree of p (higher than 10), the number of splittings needed to eliminate the correlation problem is still unfeasible.

In conclusion, rigorous global optimization algorithms based *only* on recursive interval subdivision and interval evaluation for the error function and its derivatives are not suitable in our case.

2.3. Methods that evade the dependency phenomenon

In order to bypass the high dependency problem present in $\varepsilon = p - f$, we have to write $p - f$ differently, so that we can avoid the decorrelation between f and p. For this, one widespread technique in the literature [16,13,17,18] consists in replacing the function f by another polynomial T that approximates it and for which a bound on the error $f - T$ is not too difficult to obtain. While choosing the new approximation polynomial T, we have several advantages.

First, we can consider particular polynomials for which the error bound is known or it is not too difficult to compute. One such polynomial approximation can be obtained by a Taylor series expansion of f, supposing that it is differentiable up to a sufficient order. If f behaves sufficiently well, this eventually gives a good enough approximation of f to be used instead of f in the supremum norm.

Second, we remove the decorrelation effect between f and p, by transforming it into a cancellation phenomenon between the coefficients of T and those of p. Increasing the precision used is sufficient for dealing with such a cancellation [18].

Approaches that follow this idea have been developed in the past 15 years [19,16,15,20,18]. In the following, we analyze them based on the above-mentioned key features for our algorithm.

Krämer needed to bound approximation errors while he was developing the FI_LIB library [19]. His method was mostly manual: bounds were computed case by case, and proofs were made on paper. As explained above, from our point of view, this is a drawback.

In [16], and more recently in [20], one of the authors of the present paper proposed approaches for validating bounds on approximation errors with a formal proof checker. This method presents an important advantage compared to other techniques concerning the safety of the results: the proofs of Krämer were made by hand, which is error prone; likewise, the implementation of an automatic algorithm could contain bugs. In contrast, a formal proof can be checked by a computer and gives a high guarantee. The methods presented in [16,20] mainly use the same techniques as Krämer: in particular, the formal proof is written by hand. This is safer than in [19], since the proof can be automatically checked, but not completely satisfying: we would like the proof to be automatically produced by the algorithm.

Recently, in [15,18], authors of the present article tried to automate the computation of an upper bound on $\|\varepsilon\|_\infty$. It did not seem obvious how to automate the technique used in [19,16,20], while correctly handling the particular difficulty of removable discontinuities. The algorithm presented in [15] was a first attempt. The idea was to enhance interval arithmetic [21], such that it could correctly handle expressions exhibiting removable discontinuities. As mentioned in Section 2.2, this algorithm does not really take into account the particular nature of ε, and has mainly the same drawbacks as the generic global optimization techniques.

The second attempt, presented in [18], was more suitable but not completely satisfying. This approach is based on Taylor forms [13,17]. Generally speaking, Taylor forms represent an automatic way of providing both a Taylor polynomial T and a bound for the error between f and T. In [18], automatic differentiation is used both for computing the coefficients of T and for obtaining an interval bound Δ for the Taylor remainder, using Lagrange's formula for the remainder. This technique for obtaining the couple (T, Δ) was available since publication of the work of Moore [21]. In [18], it is then used as a building block in a more complex algorithm for computing the supremum norm of error functions. This algorithm is able to handle complicated examples quickly and accurately. However, two limitations were spotted by the authors. One is that there is no *a priori* control of the accuracy obtained for the supremum norm. In fact, the algorithm uses a polynomial T, of a heuristically determined degree, such that the remainder bound obtained is less than a "hidden" parameter, which is heuristically set by the user. This means, in fact, that the approach is not completely automatic. Another limitation of this algorithm is that no formal proof is provided. This is mainly due to the combination of several techniques that are not currently available in formal proof checkers, like techniques for rigorously isolating the zeros of a polynomial or automatic differentiation.

Another way of computing a Taylor form automatically was introduced by Berz and his group [17] under the name of "Taylor models", and it was used for problems that seemed intractable by interval arithmetic. Although Taylor models proved to have many applications, the available software implementations are scarce. The best known, which, however, is not easily available, is COSY [22], written in FORTRAN by Berz and his group. Although highly optimized and used for rigorous global optimization problems in [23,24] and articles referenced therein, currently, for our specific problem, COSY has several major drawbacks. First, it does not provide multiple-precision arithmetic, and thus fails to solve the cancellation problem mentioned. Second, it does not provide any *a priori* control of the accuracy obtained for the global optimum. Third, it does not deal with the problem of functions that have removable discontinuities.

3. Computing a safe and guaranteed supremum norm

3.1. Computing a validated supremum norm versus validating a computed supremum norm

As we have seen, the various previously proposed algorithms for supremum norm evaluation have the following point in common: they use validated techniques for *computing* a rigorous upper bound $u \geq \|\varepsilon\|_\infty$. In contrast, our method consists in computing a *presumed* upper bound with a fast heuristic technique, and then only *validating* this upper bound.

Moreover, we saw in Section 2.1 that it is easy to implement a procedure `computeLowerBound` that allows us to compute a rigorous (and presumably accurate) lower bound $\ell \leq \|\varepsilon\|_\infty$. If one considers u as an approximate value of the exact norm $\|\varepsilon\|_\infty$, it is possible to bound the relative error between u and the exact norm by means of ℓ:

$$\left|\frac{u - \|\varepsilon\|_\infty}{\|\varepsilon\|_\infty}\right| \leq \left|\frac{u-\ell}{\ell}\right|.$$

This is usually used as an *a posteriori* check that the computed bound u satisfies given quality requirements on the supremum norm. Otherwise, the validated techniques can be repeated with changed parameters in the hope of obtaining a more tightly bounded result. As already discussed, the exact relationship between these parameters and the accuracy of the result is generally unknown [15,18]. In contrast, our method ensures *by construction* that the quantity $|(u-\ell)/\ell|$ is smaller than a bound $\overline{\eta}$ given by the user.

3.2. Scheme of the algorithm

The principle of our algorithm is the following.

1. Compute a sufficiently accurate lower bound ℓ of $\|\varepsilon\|_\infty$.
2. Consider a value u slightly greater than ℓ, so that most likely u is an upper bound of $\|\varepsilon\|_\infty$.
3. Validate that u is actually an upper bound.
   (a) Compute a very good approximation polynomial $T \simeq f$, such that the error between them is bounded by a given value $\overline{\delta}$.
   (b) Use the triangle inequality: rigorously bound the error between p and T and use this bound together with $\overline{\delta}$ to prove that u is indeed a rigorous bound for the error between p and f.

In the following, we will explain our algorithm more formally. We remark that the computational part and the validation part are completely distinct. This is particularly interesting if a formal proof is to be generated: whatever methods are used for computing ℓ and u, the only things that must be formally proved are a triangle inequality, a rigorous bound on the error between T and f, and a rigorous bound on the error between p and T.

3.3. Validating an upper bound on $\|\varepsilon\|_\infty$ for absolute error problems $\varepsilon = p - f$

Let us first consider the absolute error case. The relative error case is quite similar in nature, but slightly more intricate technically. It will be described in Section 3.5. Our method is summed up in Algorithm 3.1.

We first run `computeLowerBound` with $\gamma = \overline{\eta}/32$. Roughly speaking, this means that we compute the supremum norm with 5 guard bits. These extra bits will be used as a headroom allowing for roughness when *validating* that u is an upper bound. There is no particular need to choose the value 1/32: generally speaking, we could choose $\beta_1\,\overline{\eta}$, where $0 < \beta_1 < 1$ is an arbitrary constant. We let $m := \ell\,(1 + \overline{\eta}/32)$. If our hypothesis about the accuracy of ℓ is right, we thus have $\|\varepsilon\|_\infty \leq m$.

```
1 Algorithm: supremumNormAbsolute
  Input: $p, f, I, \overline{\eta}$
  Output: $\ell, u$ such that $\ell \leq \|p-f\|_\infty^I \leq u$ and $|(u-\ell)/\ell| \leq \overline{\eta}$
2 $\ell \leftarrow$ computeLowerBound($p - f$, $I$, $\overline{\eta}/32$);
3 $m' \leftarrow \ell\,(1+\overline{\eta}/2)$;    $u \leftarrow \ell\,(1+31\,\overline{\eta}/32)$;    $\overline{\delta} \leftarrow 15\,\ell\,\overline{\eta}/32$;
4 $T \leftarrow$ findPolyWithGivenError($f$, $I$, $\overline{\delta}$);
5 $s_1 \leftarrow m' - (p - T)$;    $s_2 \leftarrow m' - (T - p)$;
6 if showPositivity($s_1$, $I$) $\wedge$ showPositivity($s_2$, $I$) then return ($\ell$, $u$);
7 else return $\perp$ ;                    /* Numerically obtained bound $\ell$ not accurate enough */
```

Algorithm 3.1: Complete supremum norm algorithm for $\|\varepsilon\|_\infty = \|p-f\|_\infty$.

We define $u := \ell\,(1+31\,\overline{\eta}/32)$. Trivially, $(u-\ell)/\ell \leq \overline{\eta}$; we only have to validate that $\|\varepsilon\|_\infty \leq u$.

We introduce an intermediate polynomial T. The triangle inequality gives

$$\|p-f\|_\infty \leq \|p-T\|_\infty + \|T-f\|_\infty . \quad (1)$$

We can choose the polynomial T as close as we want to f. More precisely, we will describe in Section 4 a procedure `findPolyWithGivenError` that rigorously computes T such that $\|T-f\|_\infty \leq \overline{\delta}$, given f and $\overline{\delta}$.

We need a value m' such that we are fairly confident that $\|p-T\|_\infty \leq m'$. If this inequality is true, it will be easy to check it afterwards: it suffices to check that the two polynomials $s_1 = m' - (p-T)$ and $s_2 = m' - (T-p)$ are positive on the whole domain I under consideration. There is a wide literature on proving positivity of polynomials. For the certification through formal proofs, a perfectly adapted technique is to rewrite the polynomials s_i as a sum of squares. This will be explained in detail in Section 5. If we do not need a formal proof, other techniques may be more relevant. For instance, it suffices to exhibit one point where the polynomials s_i are positive and to prove that they do not vanish in I: this can be performed using traditional polynomial real roots counting techniques, such as Sturm sequences or the Descartes test [25]. In our current implementation, the user can choose between proving the positivity with a Sturm sequence, or computing a formal certificate using a decomposition as a sum of squares.

Clearly, if $\overline{\delta}$ is small enough, T is very close to f and we have $\|p-T\|_\infty \simeq \|p-f\|_\infty$. Hence m' will be chosen slightly greater than m. More formally, we use again the triangle inequality: $\|p-T\|_\infty \leq \|p-f\|_\infty + \|f-T\|_\infty$; hence we can choose $m' = m + \overline{\delta}$.

Putting this information into (1), we have $\|p-f\|_\infty \leq m' + \overline{\delta} \leq m + 2\overline{\delta}$. This allows us to quantify how small $\overline{\delta}$ must be: we can take $\overline{\delta} = \ell\,\mu\,\overline{\eta}$, where μ is a positive parameter satisfying $\beta_1 + 2\mu \leq 1$. We conveniently choose $\mu = 15/32$: hence m' simplifies to $m' = \ell\,(1+\overline{\eta}/2)$.

3.4. Case of failure of the algorithm

It is important to remark that the inequality $\|p-T\|_\infty \leq m + \overline{\delta}$ relies on the hypothesis that $\|p-f\|_\infty \leq m$. This hypothesis might actually be wrong, because the accuracy provided by the numerical procedure `computeLowerBound` is only heuristic. In such a case, the algorithm will fail to prove the positivity of s_1 or s_2.

However, our algorithm never lies, i.e. if it returns an interval $[\ell, u]$, this range is proved to contain the supremum norm and to satisfy the quality requirement $\overline{\eta}$.

There is another possible case of failure: the procedure `findPolyWithGivenError` might fail to return an approximation polynomial T satisfying the requirement $\|T-f\|_\infty \leq \overline{\delta}$. We will explain why in Section 4.

In practice, we never encountered cases of failure. Nevertheless, if such a case happens, it is always possible to bypass the problem: if the algorithm fails because of `computeLowerBound`, it suffices to run it again with $\gamma \ll \overline{\eta}/32$, in the hope of obtaining at least the five desired guard bits. If the algorithm fails because of `findPolyWithGivenError`, it suffices in general to split I into two (or more) subintervals and handle each subinterval separately.

3.5. Relative error problems $\varepsilon = p/f - 1$ without removable discontinuities

In order to ease the issue with relative approximation error functions $\varepsilon = p/f - 1$, let us start with the assumption that f does not vanish in the interval I under consideration for $\|\varepsilon\|_\infty^I$. That assumption actually implies that $p/f - 1$ does not have any removable discontinuity. We will eliminate this assumption in Section 3.6.

Since f does not vanish and is continuous on the compact domain I, $\inf_I |f| \neq 0$. In general, a simple interval evaluation of f over I gives an interval $\boldsymbol{J}$ that does not contain 0. In this case, $F = \min\{|\inf \boldsymbol{J}|, |\sup \boldsymbol{J}|\}$ is a rigorous non-trivial lower bound on $|f|$. Such a bound will prove useful in the following. In the case when $\boldsymbol{J}$ actually contains zero, it suffices to split the interval I. However, we never had to split I in our experiments. The accuracy of F with respect to the exact value $\inf_I |f|$ is not really important: it only influences the definition of $\overline{\delta}$, forcing it to be smaller than it needs to be. As a result, the degree of the intermediate polynomial T might be slightly greater than required.

```
1 Algorithm: supremumNormRelative
  Input: $p, f, I, \overline{\eta}$
  Output: $\ell, u$ such that $\ell \le \|p/f - 1\|_\infty^I \le u$ and $|(u-\ell)/\ell| \le \overline{\eta}$
2 $J \leftarrow f(I)$;   if $0 \in J$ then return $\perp$ else $F \leftarrow \min\{|\inf J|, |\sup J|\}$;
3 $\ell \leftarrow$ computeLowerBound($p/f - 1$, $I$, $\overline{\eta}/32$);
4 $m' \leftarrow \ell\,(1+\overline{\eta}/2)$;   $u \leftarrow \ell\,(1+31\,\overline{\eta}/32)$;   $\overline{\delta} \leftarrow 15\,\ell\,\overline{\eta}/32\,\left(\frac{1}{1+u}\right)\left(\frac{F}{1+15\,\overline{\eta}/32}\right)$;
5 $T \leftarrow$ findPolyWithGivenError($f$, $I$, $\overline{\delta}$);
6 $s \leftarrow \mathrm{sign}(T(\inf I))$;   $s_1 \leftarrow s\,m'\,T - (p-T)$;   $s_2 \leftarrow s\,m'\,T - (T-p)$;
7 if showPositivity($s_1$, $I$) $\wedge$ showPositivity($s_2$, $I$) then return ($\ell$, $u$);
8 else return $\perp$ ;                       /* Numerically obtained bound $\ell$ not accurate enough */
```

Algorithm 3.2: Complete supremum norm algorithm for $\|\varepsilon\|_\infty = \|p/f - 1\|_\infty$.

We work by analogy with the absolute error case. As before, we define $m := \ell\,(1+\beta_1\,\overline{\eta})$, $m' := \ell\,(1+(\beta_1+\mu)\,\overline{\eta})$ and $u := \ell\,(1+(\beta_1+2\mu)\,\overline{\eta})$. As before, we want to choose $\overline{\delta}$ in order to ensure that $\left\|\frac{p}{T} - 1\right\|_\infty$ be smaller than m'. For this purpose, we use the convenient following triangle inequality:

$$\left\|\frac{p}{T} - 1\right\|_\infty \le \left\|\frac{p}{f} - 1\right\|_\infty + \left\|\frac{p}{f}\right\|_\infty \left\|\frac{1}{T}\right\|_\infty \|T - f\|_\infty \tag{2}$$

If the hypothesis on the accuracy of ℓ is correct, $\left\|\frac{p}{f}\right\|_\infty$ is bounded by $1+u$. Moreover, T is close to f, so $\|1/T\|_\infty$ can be bounded by something a bit larger than $1/F$. This lets us define $\overline{\delta}$ as

$$\overline{\delta} := \ell\,\mu\,\overline{\eta}\left(\frac{1}{1+u}\right)\left(\frac{F}{1+\mu\,\overline{\eta}}\right). \tag{3}$$

Lemma 1. *We have* $\|1/T\|_\infty \le (1+\mu\,\overline{\eta})/F$.

Proof. We remark that $\ell/(1+u) \le 1$; hence we have $\overline{\delta} \le F\left(\frac{\mu\,\overline{\eta}}{1+\mu\,\overline{\eta}}\right)$. Now,

$$\forall x \in I, \quad |T(x)| \ge |f(x)| - |T(x) - f(x)| \ge F - \overline{\delta} \ge \frac{F}{1+\mu\,\overline{\eta}}.$$

This concludes the proof. □

Using this lemma, and provided that the hypothesis $\|p/f - 1\|_\infty \le m$ is correct, we see that Eq. (2) implies that $\|p/T - 1\|_\infty \le m'$. Our algorithm simply validates this inequality. As before, this reduces to checking the positivity of two polynomials. Indeed, the previous lemma shows, as a side effect, that T does not vanish in the interval I. Let us denote by $s \in \{-1, 1\}$ its sign on I. Thus $|p/T - 1| \le m'$ is equivalent to $|p - T| \le s\,m'\,T$. Defining $s_1 = s\,m'\,T - (p - T)$ and $s_2 = s\,m'\,T - (T - p)$, we just have to show the positivity of s_1 and s_2.

In order to conclude, it remains to show that $\|p/T - 1\|_\infty \le m'$ implies that $\|p/f - 1\|_\infty \le u$. For this purpose, we use Eq. (2), where the roles of f and T are inverted:

$$\left\|\frac{p}{f} - 1\right\|_\infty \le \left\|\frac{p}{T} - 1\right\|_\infty + \left\|\frac{p}{T}\right\|_\infty \left\|\frac{1}{f}\right\|_\infty \|f - T\|_\infty .$$

In this equation, $\|p/T - 1\|_\infty$ is bounded by m', $\|p/T\|_\infty$ is bounded by $1 + m' \le 1 + u$, $\|1/f\|_\infty$ is bounded by $1/F$, and $\|f - T\|_\infty$ is bounded by $\overline{\delta}$. Using the expression of $\overline{\delta}$ given by Eq. (3), we finally have $\|p/f - 1\|_\infty \le m' + \ell\,\mu\,\overline{\eta} \le u$.

3.6. Handling removable discontinuities

We now consider the case when f vanishes over I. Several situations are possible.

- The zeros of f are exact floating-point numbers, and the relative error $\varepsilon = p/f - 1$ can be extended and remains bounded on the whole interval I by continuity. As seen in Section 1, the matter is not purely theoretical, but is quite common in practice. This is the case that we address in the following.
- The function f vanishes at some point z that is not exactly representable, but ε remains reasonably bounded if restricted to floating-point numbers. In this case, it is reasonable to consider the closest floating-point values to z: $\underline{z} < z < \overline{z}$. It suffices to compute the supremum norm of ε separately over the intervals $[\inf I,\ \underline{z}]$ and $[\overline{z},\ \sup I]$. Hence, this case does not need to be specifically addressed by our algorithm.

- The relative error is not defined at some point z, and takes very large values in the neighborhood of z, even when restricted to floating-point numbers. This case does not appear in practice: indeed, we supposed that p was constructed for being a good approximation of f on the interval I. In consequence, we do not consider as a problem the fact that our algorithm fails in such a situation.

From now on, we concentrate on the first item: hence we presume that we are in a case when ε can be extended by continuity on the whole interval I and that the zeros of f are exact floating-point numbers. As will be shown in what follows, this presumption is not a hypothesis that is necessary for the rigor of the supremum norm result. It merely ensures that the result is meaningful and not just an enclosure with infinite bounds.

The following heuristic is quite common in manual supremum norm computations [15]: as p is a polynomial, it can have only a finite number of zeros z_i with orders k_i. In order for ε to be extended by continuity, the zeros of f must be amongst the z_i. This means that it is possible to determine a list of s presumed floating-point zeros z_i of f and to transform the relative approximation error function as $\varepsilon = q/g - 1$, where $q(x) = \frac{p(x)}{(x-z_0)^{k_0} \cdots (x-z_{s-1})^{k_{s-1}}}$ and $g(x) = \frac{f(x)}{(x-z_0)^{k_0} \cdots (x-z_{s-1})^{k_{s-1}}}$.

In this transformation, two types of rewritings are used. On the one hand, q is computed by long division using exact, rational arithmetic. The remainder being zero indicates whether the presumed zeros of f are actual zeros of p, as expected. If the remainder is not zero, the heuristic fails; otherwise, q is a polynomial. In contrast, the division defining g is performed symbolically, i.e. an expression representing g is constructed.

With the transformation performed, q is a polynomial and g is presumably a function not vanishing on I. These new functions are hence fed into Algorithm 3.2.

As a matter of course, g might actually still vanish on I as the zeros of f are only numerically determined: this does not compromise the safety of our algorithm. In the case when g does vanish, Algorithm 3.2 will fail while trying to compute F. This indicates the limits of our heuristic which, in practice, just works fine. See Section 6 for details on examples stemming from practice.

We remark that the heuristic algorithm just discussed actually only moves the problem elsewhere: into the function g for which an intermediate polynomial T must eventually be computed. This is not an issue. The expression defining f may anyway have removable discontinuities. This can even happen for supremum norms of absolute approximation error functions. An example would be $f(x) = \frac{\sin x}{\log(1+x)}$ approximated by $p(x) = 1 + x/2$ on an interval I containing 0. We will address in Section 4.4 the problem of computing an intermediate approximation polynomial T and a finite bound $\overline{\delta}$ when the expression of f contains a removable discontinuity.

4. Obtaining the intermediate polynomial T and its remainder

In what follows, we detail the procedure `findPolyWithGivenError` needed in the Algorithms 3.1 and 3.2. Given a function f, a domain I and a bound $\overline{\delta}$, it computes an intermediate polynomial T such that $\|T - f\|_\infty \leq \overline{\delta}$. We denote by n the degree of the intermediate polynomial T, and by $R = f - T$ the approximation error.

In the following, we will in fact focus on procedures `findPoly`(f, I, n) that, given n, return a polynomial T of degree n and an interval bound Δ rigorously enclosing $R(I)$: $\Delta \supseteq R(I)$. It is important to note that the polynomial T will eventually be used to prove polynomial inequalities in a formal proof checker. In such an environment, the cost of computations depends strongly on the degree of the polynomials involved. So we want the degree n to be as small as possible. A simple bisection strategy over n, as described in Algorithm 4.1, allows us to implement `findPolyWithGivenError` using the procedure `findPoly`.

Two factors influence the quality of Δ: first, how well T approximates f over I, i.e. how small $\|R\|_\infty$ is, and second, how much overestimation will be introduced in Δ when rigorously bounding R over I.

In that respect, we present several approaches to implement `findPoly`, classified as follows.

- Methods that separately compute a polynomial T and afterwards a bound Δ.
  - The *interpolation polynomial method* that computes a near-optimal approximation (i.e. tries to minimize $\|R\|_\infty$) and then computes Δ using *automatic differentiation*.
  - Methods based on Taylor expansions, where $\|R\|_\infty$ is larger, but Δ is computed with less overestimation.
- The *Taylor models method* that simultaneously computes the polynomial T along with a bound Δ.

We have tested all these methods. In our experiments, we could not find one clearly dominating the others by tightness of the obtained bound Δ. Taylor models often give relatively accurate bounds, so they might be preferred in practice. However, this is just a remark based on a few experiments, and it probably should not be considered as a universal statement.

In what follows, we present the algorithms as if all the operations could be performed exactly, without rounding errors. This is only for the sake of simplicity. In practice, it is easy to implement them using interval arithmetic in arbitrary precision: the real values are replaced by narrow intervals enclosing them. By increasing the precision, the width of these narrow intervals can be arbitrarily reduced. The resulting polynomial T is hence a polynomial with almost point-interval coefficients. Then T is converted into a polynomial with floating-point coefficients plus a rigorous (and narrow) interval bound. In particular, arbitrary-precision interval arithmetic is used throughout the implementation of our supremum norm algorithm in the Sollya tool.

```
1  Algorithm: findPolyWithGivenError
   Input: $f, I, \overline{\delta}$
   Output: $T$ such that $\|T - f\|_\infty^I \leq \overline{\delta}$ while trying to minimize the degree of $T$
2  $n \leftarrow 1$;
3  do
4      $(T, \Delta_{\mathrm{try}}) \leftarrow$ findPoly($f, I, n$);    $n \leftarrow 2n$;
5  while $\Delta_{\mathrm{try}} \not\subseteq [-\overline{\delta}; \overline{\delta}]$ ;
6  $n \leftarrow n/2$;    $n_{\min} \leftarrow n/2$;    $n_{\max} \leftarrow n$;
7  while $n_{\min} + 1 < n_{\max}$ do
8      $n \leftarrow \lfloor (n_{\min} + n_{\max})/2 \rfloor$;    $(T, \Delta_{\mathrm{try}}) \leftarrow$ findPoly($f, I, n$);
9      if $\Delta_{\mathrm{try}} \subseteq [-\overline{\delta}; \overline{\delta}]$ then $n_{\max} \leftarrow n$ else $n_{\min} \leftarrow n$ ;
10 end
11 $(T, \Delta_{\mathrm{try}}) \leftarrow$ findPoly($f, I, n_{\max}$);
12 return $T$ ;
```

Algorithm 4.1: Find a polynomial T such that $\|T - f\|_\infty \leq \overline{\delta}$ with n as small as possible.

4.1. *Computing T before bounding the error*

4.1.1. *Using an interpolation polynomial and automatic differentiation*

Interpolation polynomials are a good choice for T. First, they are easy to compute; the reader can find techniques in any book on numerical analysis. Second, it is well known (see, e.g., [4]) that, when using suitable interpolation points, a near-optimal polynomial approximation is obtained. Roughly speaking, this means that for such an approximation, $\|R\|_\infty$ is almost as small as possible. See, e.g., [4] for an effective measure of this property. Finally, we can rigorously bound R using the following formula [4]: if T interpolates f at points $x_0, \ldots, x_n \in I$, the error R satisfies

$$\forall x \in I, \ \exists\, \xi \in I, \quad R(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - x_i), \tag{4}$$

where $f^{(n+1)}$ denotes the $(n+1)$-th derivative of f.

When bounding R using (4), the only difficulty is to bound $f^{(n+1)}(\xi)$ for $\xi \in I$.

This can be achieved using a technique often called *automatic differentiation* [26,27,21],[4] *differentiation arithmetic* [27], *algorithmic differentiation* [28] or *Taylor arithmetic* [26]. It allows for evaluating the first n derivatives of f at a point x_0 without doing any symbolic differentiation.

The general idea of this technique is to replace any function g by an array $G = [g_0, \ldots, g_n]$, where $g_i = g^{(i)}(x_0)/i!$. It is easy to see that, given two arrays U and V (corresponding to two functions u and v), the array corresponding to $w = u + v$ is simply given by $\forall i,\ w_i = u_i + v_i$. Using the Leibniz formula, the array corresponding to $w = u\,v$ is given by $\forall i,\ w_i = \sum_{k=0}^{i} u_k\, v_{i-k}$. Also, there exist formulas [21,26] for computing W from U, where $w = \exp(u)$, $w = \sin(u)$, $w = \arctan(u)$, etc. Hence, given any function f represented by an expression, there exists a straightforward recursive procedure that uses the above-mentioned rules for computing the array F.

In fact, manipulating these arrays is nothing but manipulating truncated formal series. Fast algorithms exist for multiplying, composing or inverting formal series [29,30]. However, they are not mandatory for our purpose, since we intend to deal with degrees n not larger than a few hundreds.

For bounding $f^{(n+1)}(\xi)$ when $\xi \in I$, it suffices to apply the same automatic differentiation algorithm using interval arithmetic and replacing x_0 by I. Unfortunately, due to the *dependency phenomenon* (see Section 2.2), the bound obtained for $f^{(n+1)}(I)$, and hence Δ, can be highly overestimated. This implies that we may lose all the benefit of using an interpolation polynomial, although the actual bound $\|R\|_\infty$ is almost optimal.

4.1.2. *Using a Taylor polynomial*

The remark above suggests that it could be interesting to use a polynomial T with worse actual approximation quality, i.e. a larger $\|R\|_\infty$, but for which the computation of the bound Δ is less prone to overestimation. Consider a point $x_0 \in I$. We suppose that, for each x in I, $f(x)$ is the sum of its Taylor series:

[4] More precisely, we should say that it is inspired by automatic differentiation, since automatic differentiation is usually a code-transformation process, intended to deal with functions of several variables.

$$\forall x \in I, \quad f(x) = \underbrace{\left(\sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!} (x - x_0)^i \right)}_{T(x)} + \underbrace{\left(\sum_{i=n+1}^{+\infty} \frac{f^{(i)}(x_0)}{i!} (x - x_0)^i \right)}_{R(x)}. \tag{5}$$

Technically, this means that the function is *analytic* on a complex disc $\mathcal{D}$ containing I and centered on x_0 in the complex plane [31].

The functions we are concerned with are analytic on the whole complex plane, except maybe at a given list of points, their *singularities*. In practice, the singularities are often far enough from the interval I that the hypothesis holds. When singularities are too close to I, it is not possible to find a disc centered in x_0, containing I, and avoiding all the singularities. In this case, it suffices in general to split I into subintervals and reason on each subinterval separately.

Computing the Taylor polynomial T itself is easy: the coefficients can efficiently be computed using automatic differentiation. So, the main difficulty just lies in computing a bound for R.

One solution could be to use the classical Lagrange formula. Indeed, this is a special case of formula (4): hence it also requires one to bound $f^{(n+1)}(I)$, and it leads to exactly the same problem of overestimation as before, but without the advantages exhibited by the interpolation polynomials.

Another, more promising, technique to bound R starts with the observation that, in Eq. (5), R is expressed as a series. If we can find values M and d such that we can prove that

$$\forall i > n, \quad \left| \frac{f^{(i)}(x_0)}{i!} \right| \le \frac{M}{d^i} := b_i, \tag{6}$$

we can obviously bound R with

$$\forall x \in I, \quad |R(x)| = \left| \sum_{i=n+1}^{+\infty} \frac{f^{(i)}(x_0)}{i!} (x - x_0)^i \right| \le \sum_{i=n+1}^{+\infty} b_i \, |x - x_0|^i. \tag{7}$$

Here, $\sum_{i=n+1}^{+\infty} b_i \, |x - x_0|^i$ is a *majorizing series* of R. Since it is geometric, it is easy to bound:

$$\text{if } \gamma := \max_{x \in I} \frac{|x - x_0|}{d} < 1, \quad \text{it holds that } \forall x \in I, \ |R(x)| \le \frac{M\,\gamma^{n+1}}{1 - \gamma}.$$

Of course the principle of majorizing series is not limited to geometric series: b_i can take other forms than M/d^i, provided that the series $\sum_{i=n+1}^{+\infty} b_i \, |x - x_0|^i$ can easily be bounded. Neher and Eble proposed a software tool called ACETAF [32] for automatically computing suitable majorizing series. ACETAF proposes several techniques, all based on a fundamental theorem of complex analysis called *Cauchy's estimate*. We refer to [32] for the details on this subject.

The methods used in ACETAF depend on many parameters that, at least currently, need to be heuristically adjusted by hand. With the parameters well adjusted, we experimentally observed that the computed bound Δ was a fairly tight enclosure of the actual bound $R(I)$. Due to this reduced overestimation, we frequently obtained better rigorous bounds with this method compared to the one presented in Section 4.1.1, where a near-optimal polynomial approximation was used.

However, without adjustment, very poor bounds are computed by ACETAF. Hence, though promising, this method cannot straightforwardly be used for designing a completely automatic algorithm. This is a major drawback in our case.

Furthermore, these methods need the hypothesis that the function f is analytic on a given complex disc $\mathcal{D}$ to be verified beforehand. If this hypothesis is not fulfilled, they may return a completely incorrect finite bound Δ. One typical such case occurs when f has a singularity in the interior of $\mathcal{D}$, but $|f|$ remains bounded over the contour $\partial\,\mathcal{D}$ [32]. As we have already said, the hypothesis of analyticity of f can be considered as practically always true. However, in our context, it is not sufficient to *believe* that the function satisfies this hypothesis: we have to *prove* it automatically, and if possible formally. ACETAF includes an algorithm for checking the analyticity of a function over a disc, but it seems to be only heuristic, and thus is not suitable for our purpose.

As an alternative to methods based on Cauchy's estimate, Mezzarobba and Salvy show in [33] how to compute an accurate majorizing series and its bound, for the special class of D-finite functions, also called *differentially finite* or *holonomic*. A function is D-finite when it is a solution of a linear differential equation with polynomial coefficients. This class is large (it contains, e.g., exp, sin, arctan, Bessel functions, etc.), but not all commonly used functions are D-finite: for instance, tan is not. More generally, the set of D-finite functions is not closed under composition. This is a serious limitation in our case, as we aim for an automatic and more general approach.

4.2. Simultaneously computing T and Δ

Both techniques presented so far consist in two separate steps: first we compute the polynomial T, and afterwards we rigorously bound the approximation error by Δ. Simultaneous computation of both T and Δ is also possible.

Such a technique has been promoted by Berz and Makino [34,17] under the name of *Taylor models*. More precisely, given a function f over an interval I, this method simultaneously computes a polynomial T_f (usually, the Taylor polynomial of f, with approximate coefficients) and an interval bound $\boldsymbol{\Delta_f}$ such that $\forall x \in I,\ f(x) - T_f(x) \in \boldsymbol{\Delta_f}$. The method applies to any function given by an expression; there is no parameter to manually adjust.

Taylor models do not require the function to be analytic. Indeed, if the function has a singularity in the complex plane close to the interval I, Taylor models are likely to compute a very bad bound $\boldsymbol{\Delta}$. However, this bound remains rigorous in any case. In contrast, the methods described in the previous section were *based* on the fact that the series converges to f: hence, to use them, it was *necessary* to check the analyticity beforehand. We stress again: in general f *is* analytic, but automatically proving it may be challenging.

Taylor models are heavily inspired by automatic differentiation. As we have seen, automatic differentiation allows one to compute the first n derivatives of a function by applying simple rules recursively on the structure of f. Following the same idea, Taylor models compute the couple $(T,\ \boldsymbol{\Delta_f})$ by applying simple rules recursively on the structure of f. Taylor models can be added, multiplied, inverted, composed, etc., as with automatic differentiation.

Indeed, the computation of the coefficients of T_f with Taylor models is completely similar to their computation by automatic differentiation. However, the bound $\boldsymbol{\Delta_f}$ computed with Taylor models is usually much tighter than the one obtained when evaluating the Lagrange remainder with automatic differentiation.

To understand this phenomenon, let us consider the Taylor expansion of a composite function $w = h \circ u$ and, in particular, its Lagrange remainder part:

$$\forall x \in I,\ \exists\, \xi \in I, \quad (h \circ u)(x) = \left(\sum_{i=0}^{n} \frac{(h \circ u)^{(i)}(x_0)}{i!} (x - x_0)^i \right) + \underbrace{\frac{(h \circ u)^{(n+1)}(\xi)}{(n+1)!} (x - x_0)^{n+1}}_{\text{Lagrange remainder}}.$$

When bounding this remainder with automatic differentiation, an interval enclosure $\boldsymbol{J}$ of $(h \circ u)^{(n+1)}(I)/(n+1)!$ is computed. This interval, $\boldsymbol{J}$, is obtained by performing many operations involving enclosures of all the $u^{(i)}(I)/i!$. These enclosures are themselves obtained by recursive calls. Due to the dependency phenomenon, these values are already overestimated, and this overestimation increases at each step of the recursion.

In contrast, in the case of Taylor models, $(h \circ u)(x)$ is seen as the basic function h evaluated at point $u(x)$. Hence its Taylor expansion at $u(x_0)$ is

$$w(x) = h(u(x)) = \underbrace{\sum_{i=0}^{n} \frac{h^{(i)}(u(x_0))}{i!} (u(x) - u(x_0))^i}_{=: S(x)} + \frac{h^{(n+1)}(u(\xi))}{(n+1)!} (u(x) - u(x_0))^{n+1}. \tag{8}$$

In this formula, the only derivatives involved are the derivatives of h, which is a basic function (such as exp, sin, arctan, etc.): fairly simple formulas exist for the derivatives of such functions, and evaluating them by interval arithmetic does not lead to serious overestimation. In the sum $S(x)$, $(u(x) - u(x_0))^i$ is recursively replaced by a Taylor model $(T_i,\ \boldsymbol{\Delta_i})$ representing it. Then, the parts corresponding to the T_i contribute to the computation of T_w, while the parts corresponding to the $\boldsymbol{\Delta_i}$ contribute to the remainder. If the $\boldsymbol{\Delta_i}$ are not too much overestimated, the final remainder $\boldsymbol{\Delta_w}$ is not too much overestimated either. In conclusion, the overestimation does not grow too much during the recursion process.

That subtle algorithmic trick is the key in Taylor models. Otherwise, only technical details are to be settled. A wide literature exists, and we refer the reader to, for example, [17] or [35, Chapter 5.1] for proofs and details. In practice, several optimizations and variants have some impact on performance. However, even a straightforward out-of-the-box implementation of Taylor models gives satisfying results. In practice, the bound computed with Taylor models is much less overestimated than a bound computed by automatic differentiation. The techniques presented in Section 4.1.2 often give a tighter estimation of the remainder than Taylor models, but the gap is generally not large.

4.3. Practical comparison of the different methods

Table 1 shows the quality of some bounds obtained by the methods that we have presented. The function f, the interval I, and the degree n of T are given in the first column. The interpolation was performed at Chebyshev points (see [4]), and the corresponding approximation error R was bounded using automatic differentiation with interval arithmetic, as explained in Section 4.1.1. This leads to an enclosure $\boldsymbol{\Delta}$, the maximum absolute value of which is shown in the second column. The third column is a numerical estimation of $\|R\|_\infty$.

The Taylor polynomial was developed in the midpoint of I. The resulting interval bound was computed using ACETAF as presented in Section 4.1.2. The result is reported in the fourth column of the table. The sixth column of the table is a numerical estimation of $\|R\|_\infty$.

The fifth column corresponds to the bound given by Taylor models.

ACETAF actually proposes four different methods, with more or fewer parameters to be adjusted by hand. To limit the search space, we only show the results obtained with the first method in ACETAF, which actually has one parameter. This is somehow unfair, because the other methods may give more accurate bounds. In our experiments, we adjusted this parameter by hand, in order to minimize the width of $\boldsymbol{\Delta}$. As can be seen in Table 1, even this first simple method gives results that are often as accurate as Taylor models, or even better.

Table 1
Examples of bounds obtained by several methods.

$f(x), I, n$	Interpolation	Exact bound	ACETAF	TM	Exact bound
$\sin(x)$, [3, 4], 10	**1.19e−14**	1.13e−14	6.55e−11	1.22e−11	1.16e−11
$\arctan(x)$, [−0.25, 0.25], 15	**7.89e−15**	7.95e−17	1.00e−9	2.58e−10	3.24e−12
$\arctan(x)$, [−0.9, 0.9], 15	**5.10e−3**	1.76e−8	6.46	1.67e2	5.70e−3
$\exp(1/\cos(x))$, [0, 1], 14	0.11	6.10e−7	9.17e−2	**9.06e−3**	2.59e−3
$\frac{\exp(x)}{\log(2+x)\,\cos(x)}$, [0, 1], 15	0.18	2.68e−9	1.76e−3	**1.18e−3**	3.38e−5

We tried to make the examples representative for several situations. The first example is a basic function which is analytic on the whole complex plane. There is almost no overestimation in this case, whatever method we use. The second is also a basic function, but it has singularities in the complex plane. However, in this example, the interval I is relatively far from the singularities. All the methods present a relatively small overestimation. The third example is the same function, but over a wider interval, so the singularities are closer, and Taylor polynomials are not very good approximations. The fourth and fifth examples are composite functions on fairly wide intervals, which challenge the methods. The overestimation in the interpolation method becomes very large, while it stays reasonable with ACETAF and Taylor models.

In each row, the method that leads to the tightest bound is set in bold. No method is better than the others in all circumstances. However, Taylor models seem to offer a good compromise, in particular for composite functions.

4.4. The problem of removable discontinuities

In Section 3.6, we solved the problem of computing supremum norms for functions with removable discontinuities, provided that the algorithm `findPolyWithGivenError` is able to handle such functions.

Classical Taylor models cannot handle such removable discontinuities: a Taylor model for $u/v = u \cdot 1/v$ is obtained by computing a Taylor model for $1/v$. If v has a zero in the interval I, the remainder bound becomes infinite. In what follows, we propose a novel modification to Taylor models that permits us to perform the division.

Consider an expression of the form u/v. We suppose that a numerical heuristic has already found a floating-point number z_0, where u and v seem to vanish simultaneously. If there are several zeros of v in the interval, we split it into subintervals. So we will also presume that z_0 is the unique zero of v in the interval. Let us stress again that these conditions are not necessary to ensure rigor. For instance, the numerical procedure that found z_0 might have missed a zero of v; or z_0 could be very close to an actual zero of v, yet not exact. This is not a problem: in these cases, our method returns a remainder bound that is infinite, useless, but perfectly rigorous (as classical Taylor models would do, anyway).

We already explained in Section 3.6 that the presumption that a floating-point number z_0 can be found as a actual zero of u and v is fairly reasonable in the practice of floating-point code development. As a matter of course, from a theoretical point of view, it could be considered as very restrictive and as a drawback of our method. However, without more symbolic techniques, handling non-representable removable discontinuities seems hopeless.

Our method is based on the following idea: represent u and v with Taylor expansions centered on z_0 and cancel out the leading part $(x - z_0)^k$ in both expressions. For example, take $z_0 = 0$, $k = 1$, and

$$\frac{\sin x}{\log(1+x)} = \frac{x - x^3/6 + x^5/120 + \cdots}{x - x^2/2 + x^3/3 - x^4/4 + \cdots} = \frac{1 - x^2/6 + x^4/120 + \cdots}{1 - x/2 + x^2/3 - x^3/4 + \cdots}.$$

Once $(x - z_0)^k$ has been factored and simplified, the denominator of the division does not vanish anymore; thus the division can be performed using the usual division of Taylor models.

The first difficulty encountered when trying to automate this scheme is that we need to be sure that the leading coefficients of u and v are exactly zero: it is not sufficient to approximately compute them. A classical solution to this problem is to represent the coefficients by narrow enclosing intervals instead of floating-point numbers. Interval arithmetic is used throughout the computations, which ensures that the true coefficients actually lie in the corresponding narrow intervals. For basic functions, we know which coefficients are exactly zero, so we can replace them by the point-interval [0, 0]. This point interval has a nice property: it propagates well during the computations, since any interval multiplied by [0, 0] leads to [0, 0] itself. So, in practice, the property of *being exactly zero* is well propagated when composing functions: hence we can expect that, for any reasonably simple functions u and v vanishing in z_0, the leading coefficients of their Taylor expansion will be [0, 0], from which we can surely deduce that the true coefficients are exactly 0.

The second difficulty is that, in fact, we do not compute series: we use Taylor models, i.e. truncated series, together with a bound on the remainder. If we use classical Taylor models, the function u/v will be replaced by something of the form

$$\frac{T_u + \Delta_{\boldsymbol{u}}}{T_v + \Delta_{\boldsymbol{v}}}. \tag{9}$$

Hence, even if both T_u and T_v have a common factor $(x - z_0)^k$, we cannot divide both the numerator and denominator by $(x - z_0)^k$, because the division will not simplify in the bounds.

In fact, if T_u is the Taylor polynomial of degree n of a function u in z_0, we know that the remainder is of the form $R_u = (x - z_0)^{n+1}\widetilde{R_u}$. The usual Taylor models propagate an enclosure $\Delta_{\boldsymbol{u}}$ of $R_u(I)$ through the computation. Instead, we can propagate an enclosure $\Delta_{\boldsymbol{u}}$ of $\widetilde{R_u}(I)$. In this case, Eq. (10) becomes

$$\frac{T_u + (x - z_0)^{n+1}\,\Delta_{\boldsymbol{u}}}{T_v + (x - z_0)^{n+1}\,\Delta_{\boldsymbol{v}}}, \tag{10}$$

and it becomes possible to cancel out the term $(x - z_0)^k$ from the numerator and the denominator.

This leads to the following definition. We note that, in this definition, z_0 itself is replaced by a narrow interval enclosing it. This is convenient when composing our modified Taylor models. Of course, since we assume that z_0 is a floating-point number, we can replace it by the point-interval $[z_0,\ z_0]$.

Definition 1 (*Modified Taylor Models*)**.** A modified Taylor model of degree n consists of:

- an interval (usually a point-interval) $\boldsymbol{z_0}$ representing the expansion point;
- a list of (usually narrow) interval coefficients $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$;
- an interval Δ representing a bound on the (scaled) remainder.

A modified Taylor model *represents* a function f over I when

$$\forall \xi_0 \in \boldsymbol{z_0}, \exists \alpha_0 \in \boldsymbol{a_0}, \ldots, \exists \alpha_n \in \boldsymbol{a_n}, \forall x \in I, \exists \delta \in \Delta, \quad f(x) = \left(\sum_{i=0}^{n} \alpha_i\,(x - \xi_0)^i\right) + (x - \xi_0)^{n+1}\,\delta.$$

The novelty in our modified Taylor models does not merely rely on the fact that the coefficients of the polynomial are represented with narrow intervals. The main difference relies in the *relative remainder bound* in which the term $(x - z_0)^{n+1}$ is kept factored out. The remainder actually vanishes when evaluating a modified Taylor model at $x = z_0$, something that never happens for a classical Taylor model. This is exactly what permits removable discontinuities to be handled.

The arithmetical operations on Taylor models (addition, negation, multiplication) easily translate to modified Taylor models, provided that the involved models have the same degree and are developed around the same interval $\boldsymbol{z_0}$. If necessary, it is easy to convert a model of degree n into a model of lower degree n'. It is also worth mentioning that a model around an interval $\boldsymbol{z_0}$ is also a valid model around any $\boldsymbol{z_0'} \subseteq \boldsymbol{z_0}$. The composition of Taylor models of same degree is also a natural generalization of the classical composition of Taylor models, provided that straightforward compatibility conditions are satisfied by the expansion points. Only the division rule is modified, as follows. First, a numerical heuristic is applied to determine the point z_0, where u and v simultaneously vanish, along with the order k of this zero. Second, modified models of order $n + k$ of u and v in z_0 are recursively computed. After a check that the leading k coefficients really are point intervals $[0,\ 0]$, the leading terms are cancelled, and a Taylor model of degree n for u/v is obtained through inversion and multiplication the classical way. If this check fails, no cancellation is performed, which leads to a Taylor model with an infinite remainder bound.

As a matter of course, it is easy to convert a modified Taylor model to a classical one by multiplying the remainder bound Δ by $(I - z_0)^{n+1}$ using interval arithmetic. We observed that the final bound is slightly more overestimated than the one obtained with classical Taylor models. However, the difference in tightness stayed fairly small in the examples we tried.

We implemented these modified Taylor models in the Sollya software tool, and used them for handling functions with removable discontinuities. The experimental results will be discussed in Section 6.

5. Certification and formal proof

Our approach is distinguished from many others in the literature in that we aim to give a validated and guaranteed error bound, rather than merely one 'with high probability' or 'modulo rounding error'. Nevertheless, since both the underlying mathematics and the actual implementations are fairly involved, the reliability of our results, judged against the very highest standards of rigor, can still be questioned by a determined skeptic. The most satisfying way of dealing with such skepticism is to use our algorithms to generate a complete formal proof that can be verified by a highly reliable proof-checking program. This is doubly attractive because such proof checkers are now often used for verifying floating-point hardware and software [16,36–42]. In such cases, bounds on approximation errors often arise as key lemmas in a larger formal proof, so an integrated way of handling them is desirable.

There is a substantial literature on using proof checkers to verify the results of various logical and mathematical decision procedures [43]. In some cases, a direct approach seems necessary, where the algorithm is expressed logically inside the theorem prover, formally proved correct, and 'executed' in a mathematically precise way via logical inference. In many cases, however, it is possible to organize the main algorithm so that it generates some kind of 'certificate' that can be formally checked, i.e. used to generate a formal proof, without any formalization of the process that was used to generate it. This can often be both simpler and more efficient than the direct approach. (In fact, the basic observation that 'result checking' can be more productive than 'proving correctness' has been emphasized by Blum [44], and appears in many other contexts, such as computational geometry [45].) The two main phases of our approach illustrate this dichotomy.

- In order to bound the difference $|f - T|$ between the function f and its Taylor series T, there seems to be no shortcut beyond formalizing the theory underlying the Taylor models inside the theorem prover and instantiating it for the particular cases used.
- Bounding the difference between the Taylor series T and the polynomial p that we are interested in reduces to polynomial non-negativity on an interval, and this admits several potential methods of certification, with 'sum-of-squares' techniques being perhaps the most convenient.

We consider each of these in turn.

5.1. Formalizing Taylor models

Fully formalizing this part inside a theorem prover is still work in progress. For several basic functions such as sin, versions of Taylor's theorem with specific, computable bounds on the remainder have been formalized in HOL Light, and set up so that formally proven bounds for any specific interval can be obtained automatically. For example, in this interaction example from [40], the user requests a Taylor series for the cosine function such that the absolute error for $|x| \leq 2^{-2}$ is less than 2^{-35}. The theorem prover not only returns the series $1 - x^2/2 + x^4/24 - x^6/720 + x^8/40320$ but also a theorem, formally proven from basic logical axioms, that indeed the desired error bound holds: $\forall x,\ |x| \leq 2^{-2} \Rightarrow |\cos(x) - (1 - x^2/2 + x^4/24 - x^6/720 + x^8/40320)| \leq 2^{-35}$.

```
#MCLAURIN_COS_POLY_RULE 2 35;;
it : thm =
 |- ∀x. abs x <= inv (&2 pow 2)
        ⇒ abs(cos x - poly [&1; &0; --&1 / &2; &0; &1 / &24; &0;
                                  --&1 / &720; &0; &1 / &40320] x)
            <= inv(&2 pow 35)
```

However, this is limited to a small repertoire of basic functions expanded about specific points, in isolation, often with restrictions on the intervals considered. Much more work of the same kind would be needed to formalize the general Taylor models framework we have described in this paper, which can handle a wider range of functions, expanded about arbitrary points and nested in complex ways. This is certainly feasible, and related work has been reported [46,47], but much remains to be done, and performing the whole operation inside a formal checker appears to be very time-consuming.

5.2. Formalizing polynomial non-negativity

Several approaches for formally proving polynomial non-negativity have been reported, including a formalization of Sturm's theorem [16] and recursive isolation of roots of successive derivatives [40]. Many of these, as well as others that could be amenable to formalization [48], have the drawback of requiring extensive computation of cases inside the formal proof checker. Computation in the formal environment of the checker is much more expensive than computing in a standard arithmetic environment, such as an interval arithmetic library [49]. An appealing idea for avoiding this cost in the proof checker is to generate certificates involving sum-of-squares (SOS) decompositions. The computation needed for generating the certificate then stays external to the proof checker. The proof checker simply checks the certificate, which is meant to be – and is – much less computationally intense.

In order to prove that a polynomial $p(x)$ is everywhere non-negative, an SOS decomposition $p(x) = \sum_{i=1}^{k} a_i s_i(x)^2$ for rational $a_i > 0$ is an excellent certificate: it can be used to generate an almost trivial formal proof, mainly involving the verification of an algebraic identity. For the more refined assertions of non-negativity over an interval $[a, b]$, slightly more elaborate 'Positivstellensatz' certificates involving sums of squares and multiplication by $b - x$ or $x - a$ work well.

For general multivariate polynomials, Parrilo [50] pioneered the approach of generating such certificates using semidefinite programming (SDP). However, the main high-performance SDP solvers involve complicated nonlinear algorithms implemented in floating-point arithmetic. While they can invariably be used to find *approximate* SOS decompositions, it can be problematic to get *exact* rational decompositions, particularly if the original coefficients have many significant bits and the polynomial has relatively low variation. Unfortunately these are just the kinds of problems we are concerned with. But if we restrict ourselves to *univariate* polynomials, which still covers our present application, more direct methods can be based only on complex root finding, which is easier to perform in high precision. In what follows, we correct an earlier description of such an algorithm [20], and extend it to the generation of full Positivstellensatz certificates.

The basic idea is simple. Suppose that a polynomial $p(x)$ with rational coefficients is everywhere non-negative. Roots always occur in conjugate pairs, and any real roots must have even multiplicity, otherwise the polynomial would cross the x-axis instead of just touching it. Thus, if the roots are $a_j \pm ib_j$, we can imagine writing the polynomial as

$$\begin{aligned} p(x) &= l \cdot [(x - [a_1 + ib_1])(x - [a_2 + ib_2]) \cdots (x - [a_m + ib_m])] \cdot \\ &\quad [(x - [a_1 - ib_1])(x - [a_2 - ib_2]) \cdots (x - [a_m - ib_m])] \\ &= l(q(x) + ir(x))(q(x) - ir(x)) \\ &= lq(x)^2 + lr(x)^2. \end{aligned}$$

This well-known proof that any non-negative polynomial can be expressed as a sum of two squares with arbitrary real coefficients can be adapted to give an exact rational decomposition algorithm, compensating for the inevitably inexact representation of the roots $a_j \pm ib_j$. This is done by finding a small initial perturbation of the polynomial that is still non-negative. The complex roots can then be located sufficiently accurately using the excellent arbitrary-precision complex root finder in PARI/GP,[5] which implements a variant of an algorithm due to Schönhage [51].

5.2.1. Squarefree decomposition

Since the main part of the algorithm introduces inaccuracies that can be made arbitrarily small but not eliminated completely, it is problematic if the polynomial is ever *exactly* zero. However, if the polynomial touches the x-axis at $x = a$, there must be a root $x - a$ of even multiplicity, say $p(x) = (x - a)^{2k}p^*(x)$. We can factor out all such roots by a fairly standard squarefree decomposition algorithm that uses only exact rational arithmetic and does not introduce any inaccuracy. The modified polynomial $p^*(x)$ can then be used in the next stage of the algorithm and the resulting terms in the SOS decomposition multiplied appropriately by the $(x - a)^k$. So suppose, hypothetically, that the initial polynomial $p(x)$ has degree n, and splits as

$$p(x) = c \prod_k (x - a_k)^{m_k}.$$

We use the standard technique of taking the greatest common divisor of a polynomial and its own derivative to separate out the repeated roots, applying it recursively to obtain the polynomials $r_i(x)$, where $r_0(x) = p(x)$, and then $r_{i+1} = \gcd(r_i(x), r_i'(x))$ for each $0 \le i \le n - 1$, so

$$r_i(x) = c \prod_k (x - a_k)^{\max(m_k - i, 0)}.$$

Note that each $m_k \le n$, so $r_i(x) = c$ for each $i \ge n$. Now, for each $1 \le i \le n + 1$, let $l_i(x) = r_{i-1}(x)/r_i(x)$, so

$$l_i(x) = \prod_k (x - a_k)^{(\text{if } m_k \ge i \text{ then } 1 \text{ else } 0)},$$

and then similarly for each $1 \le i \le n$ let $f_i(x) = l_i(x)/l_{i+1}(x)$, so that

$$f_i(x) = \prod_k (x - a_k)^{(\text{if } m_k = i \text{ then } 1 \text{ else } 0)}.$$

We have now separated the polynomial into the components $f_i(x)$, where the basic factors $(x - a_k)$ appear with multiplicity i, and we can then extract a maximal 'squarable' factor by

$$s(x) = \prod_{1 \le i \le n} f_i(x)^{\lfloor i/2 \rfloor}.$$

We can then obtain a new polynomial $p^*(x) = p(x)/s(x)^2$ without repeated roots, for the next step, and subsequently multiply each term inside the SOS decomposition by $s(x)$.

5.2.2. Perturbation

From now on, thanks to the previous step, we can assume that our polynomial is strictly positive definite, i.e. $\forall x \in \mathbb{R},\ p(x) > 0$. Since all polynomials of odd degree have a real root, the degree of the polynomial (and the original polynomial before the removal of squared part) must be even, say $\deg(p) = n = 2m$, and the leading coefficient of $p(x) = \sum_{i=0}^{n} a_i x_i$ must also be positive, $a_n > 0$. Since $p(x)$ is *strictly* positive, there must be an $\varepsilon > 0$ such that the perturbed polynomial $p_\varepsilon(x) = p(x) - \varepsilon(1 + x^2 + \cdots + x^{2m})$ is also (strictly) positive. Provided that $\varepsilon < a_n$, this is certainly positive for sufficiently large x, say $|x| > R$, since the highest term of the difference $p(x) - \varepsilon(1 + x^2 + \cdots + x^{2m})$ will eventually dominate. And on the compact set $|x| \le R$, we can just also choose $\varepsilon < \inf_{|x| \le R} p(x) / \sup_{|x| \le R}(1 + x^2 + \cdots + x^{2m})$.

To find such an ε algorithmically, we just need to test if a polynomial has real roots, which we can easily do in PARI/GP using Sturm's method; we can then search for a suitable ε by choosing a convenient starting value and repeatedly dividing by 2 until our goal is reached; we actually divide by 2 again to leave a little margin of safety. (Of course, there are more efficient ways of doing this.) We have been tacitly assuming that the initial polynomial *is* indeed non-negative, but if it is not, that fact can be detected at this stage by checking the $\varepsilon = 0$ case, ensuring that $p(x)$ has no roots and that $p(c) > 0$ for any convenient value like $c = 0$.

5.2.3. Approximate SOS of perturbed polynomial

We now use the basic 'sum of two real squares' idea to obtain an approximate SOS decomposition of the perturbed polynomial $p_\varepsilon(x)$, just by using approximations of the roots. Recall from the discussion above that, with exact knowledge of

5 http://pari.math.u-bordeaux.fr/.

the roots $a_j \pm ib_j$ of $p_\varepsilon(x)$, we could obtain an SOS decomposition with two terms. Assuming that l is the leading coefficient of $p_\varepsilon(x)$, we would have $p_\varepsilon(x) = ls(x)^2 + lt(x)^2$. Using only approximate knowledge of the roots as obtained by PARI/GP, we obtain instead $p_\varepsilon(x) = ls(x)^2 + lt(x)^2 + u(x)$, where the coefficients of the remainder $u(x)$ can be made as small as we wish. We determine how small this needs to be in order to make the next step below work correctly, and select the accuracy of the root finding accordingly.

5.2.4. Absorption of remainder term

We now have $p(x) = ls(x)^2 + lt(x)^2 + \varepsilon(1+x^2+\cdots+x^{2m}) + u(x)$, so it will suffice to express $\varepsilon(1+x^2+\cdots+x^{2m}) + u(x)$ as a sum of squares. Note that the degree of u is $< 2m$ by construction (though the procedure to be outlined would work with minor variations even if it were exactly $2m$). Let us say that $u(x) = a_0 + a_1x + \ldots + a_{2m-1}x^{2m-1}$. Note that $x = (x + 1/2)^2 - (x^2 + 1/4)$ and $-x = (x - 1/2)^2 - (x^2 + 1/4)$, and so, for any $c \geq 0$,

$$\begin{aligned} cx^{2k+1} &= c(x^{k+1} + x^k/2)^2 - c(x^{2k+2} + x^{2k}/4), \\ -cx^{2k+1} &= c(x^{k+1} - x^k/2)^2 - c(x^{2k+2} + x^{2k}/4). \end{aligned}$$

Consequently, we can rewrite the odd-degree terms of u as

$$a_{2k+1}x^{2k+1} = |a_{2k+1}|(x^{k+1} + \operatorname{sgn}(a_{2k+1})\, x^k/2)^2 - |a_{2k+1}|(x^{2k+2} + x^{2k}/4),$$

and so

$$\varepsilon(1+x^2+\cdots+x^{2m}) + u = \sum_{k=0}^{m-1} |a_{2k+1}|(x^{k+1} + \operatorname{sgn}(a_{2k+1})\, x^k/2)^2 + \sum_{k=0}^{m} (\varepsilon + a_{2k} - |a_{2k-1}| - |a_{2k+1}|/4)\, x^{2k},$$

where, by convention, $a_{-1} = a_{2m+1} = 0$. This already gives us the required SOS representation, provided that $\varepsilon \geq |a_{2k+1}|/4 - a_{2k} + |a_{2k-1}|$ for each k, and we can ensure this by computing the approximate SOS sufficiently accurately.

5.2.5. Finding Positivstellensatz certificates

By a well-known trick, we can reduce a problem of the form $\forall x \in [a,\ b],\ 0 \leq p(x)$, where $p(x)$ is a univariate polynomial, to the unrestricted polynomial non-negativity problem $\forall y \in \mathbb{R},\ 0 \leq q(y)$ by the change of variable $x = \frac{a+by^2}{1+y^2}$ and clearing denominators:

$$q(y) = (1+y^2)^{\deg(p)} p\left(\frac{a+by^2}{1+y^2}\right).$$

To see that this change of variables works, note that, as y ranges over the whole real line, y^2 ranges over the non-negative reals, and so $x = (a+by^2)/(1+y^2)$ ranges over $a \leq x < b$, and although we do not attain the upper limit b, the two problems $\forall x,\ a \leq x \leq b \Rightarrow 0 \leq p(x)$ and $\forall x,\ a \leq x < b \Rightarrow 0 \leq p(x)$ are equivalent, since $p(x)$ is a continuous function.

If we now use the algorithm from the previous subsections to obtain an SOS decomposition $q(y) = \sum_i c_i s_i(y)^2$ for non-negative rational numbers c_i, it is useful to be able to transform back to a Positivstellensatz certificate [50] for the non-negativity on $[a, b]$ of the original polynomial $p(x)$. So, suppose that we have

$$q(y) = (1+y^2)^{\deg(p)} p\left(\frac{a+by^2}{1+y^2}\right) = \sum_i c_i s_i(y)^2.$$

Let us separate each $s_i(y)$ into the terms of even and odd degree $s_i(y) = r_i(y^2) + yt_i(y^2)$, giving us the decomposition

$$q(y) = \sum_i c_i(r_i(y^2)^2 + y^2t_i(y^2)^2 + 2yr_i(y^2)t_i(y^2)).$$

However, note that, by construction, $q(y)$ is an even polynomial, and so, by comparing the odd terms on both sides, we see that $\sum_i yr_i(y^2)t_i(y^2) = 0$. By using this, we obtain the simpler decomposition arising by removing all those terms:

$$q(y) = \sum_i c_i\, r_i(y^2)^2 + c_i\, y^2\, t_i(y^2)^2.$$

Inverting the change of variable, we get $y^2 = \frac{x-a}{b-x}$ and $1 + y^2 = \frac{b-a}{b-x}$. Therefore, we have, writing $d = \deg(p)$,

$$\left(\frac{b-a}{b-x}\right)^d p(x) = \sum_i c_i r_i \left(\frac{x-a}{b-x}\right)^2 + c_i \frac{x-a}{b-x} t_i \left(\frac{x-a}{b-x}\right)^2,$$

and so

$$p(x) = \sum_i \frac{c_i}{(b-a)^d}(b-x)^d r_i \left(\frac{x-a}{b-x}\right)^2 + \frac{c_i}{(b-a)^d}(x-a)(b-x)^{d-1} t_i \left(\frac{x-a}{b-x}\right)^2 .$$

We can now absorb the additional powers of $b-x$ into the squared terms to clear their denominators and turn them into polynomials. We distinguish two cases, according to whether $d = \deg(p)$ is even or odd. If d is even, we have

$$p(x) = \sum_i \frac{c_i}{(b-a)^d}\left[(b-x)^{\frac{d}{2}} r_i \left(\frac{x-a}{b-x}\right)\right]^2 + (x-a)(b-x)\sum_i \frac{c_i}{(b-a)^d}\left[(b-x)^{\frac{d}{2}-1} t_i \left(\frac{x-a}{b-x}\right)\right]^2 ,$$

while, if d is odd, we have

$$p(x) = (b-x)\sum_i \frac{c_i}{(b-a)^d}\left[(b-x)^{\frac{d-1}{2}} r_i \left(\frac{x-a}{b-x}\right)\right]^2 + (x-a)\sum_i \frac{c_i}{(b-a)^d}\left[(b-x)^{\frac{d-1}{2}} t_i \left(\frac{x-a}{b-x}\right)\right]^2 .$$

In either case, this gives a certificate that makes clear the non-negativity of $p(x)$ on the interval $[a, b]$, since it constructs $p(x)$ via sums and products from squared polynomials, non-negative constants, and the expressions $x-a$ and $b-x$ in a simple and uniform way.

6. Experimental results

We have implemented our novel algorithm for validated supremum norms in the Sollya software tool.[6] The sum-of-squares decomposition necessary for the certification step has been implemented using the PARI/GP software tool.[7] The formal certification step has been performed using the HOL Light theorem prover.[8]

During the computation step before formal verification, the positivity of difference polynomials s_1 and s_2 (see Section 3) is shown using an interval arithmetic based implementation of the Sturm sequence algorithm [52]. The implementation has a fall-back to rational arithmetic if interval arithmetic fails to give an unambiguous answer because the enclosure is not sufficiently tight [15]. In the examples presented, this fall-back has never been invoked. However, beside this method, other well-known techniques exist [25].

The intermediate polynomial T has been computed using Taylor models. Our implementation supports both absolute and relative remainder bounds. Relative remainder bounds are used by the algorithm only when strictly necessary, i.e. when a removable discontinuity is detected (see Section 4.4). Our implementation of Taylor models also contains some optimizations for computing tighter remainder bounds found in the literature [46]. We did not explicitly discuss those optimizations in this article, for the sake of brevity.

We have compared the implementation of our novel supremum norm algorithm on 10 examples with implementations of the following existing algorithms discussed in Section 2.

- A pure numerical algorithm for supremum norms available in the Sollya tool through the command `dirtyinfnorm` [6]. The algorithm mainly samples the zeros of the derivative of the approximation error function ε and refines them with a Newton iteration. We will refer to this algorithm as `Ref1`. As a matter of course, this algorithm does not offer the guarantees we address in this article. Its purpose is only to give reference timings corresponding to the kind of algorithms commonly used to compute supremum norms.
- A rigorous interval-arithmetic-based supremum norm available through the Sollya command `infnorm` [6]. The algorithm performs a trivial bisection until interval arithmetic shows that the derivative of the approximation error function ε no longer contains any zero, or some threshold is reached. The algorithm is published in [15]. We will refer to this algorithm as `Ref2`. We were not able to obtain a result in reasonable time (less than 10 minutes) using this algorithm for some instances. The cases are marked "N/A" below.
- A rigorous supremum norm algorithm based on automatic differentiation and rigorous bounds of the zeros of a polynomial. The algorithm is published in [18]. It gives an *a posteriori* error. We will refer to this algorithm as `Ref3`.

We will refer to the implementation of our new supremum norm algorithm as `Supnorm`. We made sure all algorithms computed a result with comparable final accuracy. This required choosing suitable parameters by hand for algorithms `Ref2` and `Ref3`. The time required for this manual adaptation is not accounted for but, of course, it exceeds the computation time by a huge factor. Our novel algorithm achieves this automatically by its *a priori* accuracy control.

We used the example instances for supremum norm computations published in [18]. They are summed up in Table 2. In this table, the "mode" indicates whether the absolute or relative error between p and f was considered. In the "quality" column, we compute $-\log_2(\overline{\eta})$, which gives a measure of the number of correct bits obtained for $\|\varepsilon\|_\infty$.

[6] http://sollya.gforge.inria.fr/.
[7] http://pari.math.u-bordeaux.fr/.
[8] http://www.cl.cam.ac.uk/~jrh13/hol-light/.

Table 2
Definition of our examples.

Example number	f	$[a, b]$	$\deg(p)$	Mode	Quality $-\log_2 \overline{\eta}$
1	$\exp(x) - 1$	$[-0.25, 0.25]$	5	rel.	37.6
2	$\log_2(1+x)$	$[-2^{-9}, 2^{-9}]$	7	rel.	83.3
3[a]	$\arcsin(x+m)$	$[a_3, b_3]$	22	rel.	15.9
4	$\cos(x)$	$[-0.5, 0.25]$	15	rel.	19.5
5	$\exp(x)$	$[-0.125, 0.125]$	25	rel.	42.3
6	$\sin(x)$	$[-0.5, 0.5]$	9	abs.	21.5
7	$\exp(\cos^2 x + 1)$	$[1, 2]$	15	rel.	25.5
8	$\tan(x)$	$[0.25, 0.5]$	10	rel.	26.0
9	$x^{2.5}$	$[1, 2]$	7	rel.	15.5
10	$\sin(x)/(\exp(x) - 1)$	$[-2^{-3}, 2^{-3}]$	15	abs.	15.5

[a] Values for example 3: $m = 770422123864867 \cdot 2^{-50}$, $a_3 = -205674681606191 \cdot 2^{-53}$, $b_3 = 205674681606835 \cdot 2^{-53}$.

Table 3
Degree of the intermediate polynomial T chosen by `Supnorm`, and computed enclosure of $\|\varepsilon\|_\infty$.

Example number	$\deg(p)$	$\deg(T)$	$\|\varepsilon\|_\infty \in [\ell, u]$
1	5	13	0.98349131972[2−7]e−7
2	7	17	0.215060633232252001406277 0[4−7]e−21
3	22	32	0.25592[3−8]e−34
4	15	22	0.23083[7−9]e−24
5	25	34	0.244473007268[5−7]e−57
6	9	17	0.118837[0−2]e−13
7	15	44	0.30893006[2−9]e−13
8	10	22	0.35428[6−8]e−13
9	7	20	0.2182[5−7]e−8
10	15	27	0.6908[7−9]e−20

More precisely, we can classify the examples as follows.

- The two first examples are somehow "toy" examples, also presented in [15].
- The third example is a polynomial taken from the source code of `CRlibm`. It is the typical problem that developers of `libms` address. The degree of p is 22, which is quite high in this domain.
- In examples 4 through 10, p is the minimax polynomial, i.e. the polynomial p of a given degree that minimizes the supremum norm of the error. These examples involve more or less complicated functions over intervals of various width. Examples 7 and 9 should be considered as quite hard for our algorithm, since the interval $[a, b]$ has width 1: this is wide when using Taylor polynomials, and it requires a high degree. Example 10 shows that our algorithm is also able to manage removable discontinuities inside the function f.

The complete definition of the examples as well as our implementation of the algorithms is available with the research report version of this article at http://prunel.ccsd.cnrs.fr/ensl-00445343/.

The implementation of both the novel supremum norm algorithm and the three reference algorithms is based on the Sollya tool. That tool was compiled using `gcc` version 4.3.2. The timings were performed on an Intel® Core™ i7-975-based system clocked at 3.33 GHz running Redhat[9] Fedora 10 × 64 Linux 2.6.27.21. Table 3 presents the results of our algorithm. In this table, the interval $[\ell, u]$ computed by our algorithm is represented with the first common leading digits to ℓ and u, followed by brackets that give the actual enclosure. For instance, 1.234[5−7] actually represents the interval $[1.2345, 1.2347]$. Table 4 gives the timings that we obtained.

The implementation of our novel algorithm, compared with the other validated supremum algorithms `Ref2` and `Ref3`, exhibits the best performance. As a matter of course, counterexamples can be constructed but are hard to find.

We note that, with our new supremum norm algorithm, the overhead of using a validated technique for supremum norms of approximation error functions with respect to an unsafe, numerical technique `Ref1` drops to a factor around 3–5. This positive effect is reinforced by the fact that the absolute execution times for our supremum norm algorithm are less than 1 s in most cases. Hence supremum norm validation needs no longer be a one-time–one-shot overnight task, as previous work suggests [15].

Even certification in a formal proof checker comes into reach with our supremum norm algorithm. In the last column of Table 4, we give the execution times for the post-computational rewriting of the difference polynomials s_i as a sum of squares.

[9] Other names and brands may be claimed as the property of others.

Table 4
Timing of several algorithms.

Example number	`Ref1` time (ms)	`Ref2` time (ms)	`Ref3` time (ms)	`Supnorm` time (ms)	`SOS` time (ms)
	Not rigorous	Rigorous			
1	14	2190	121	42	1 631
2	41	N/A	913	103	11 436
3	270	N/A	1803	364	42 735
4	93	N/A	1009	139	8 631
5	337	N/A	2887	443	155 265
6	13	3657	140	39	2 600
7	180	N/A	3220	747	81 527
8	47	66565	362	94	5 919
9	27	5109	315	73	3 839
10	43	N/A	N/A	168	8 061

If even though the execution times for sum-of-squares decomposition may seem high compared to the actual supremum norm computation times, they are quite reasonable, since our implementation is still not at all optimized. Moreover, most of the time, the final certification step, requiring the computation of a sum-of-squares decomposition, is run only once per supremum norm instance in practice. Hence the time needed for computing this certificate is not critical.

7. Conclusion

Each time a transcendental function f is approximated using a polynomial p, there is a need to determine the maximum error induced by this approximation. Several domains where such a bound for the error is needed are floating-point implementation of elementary functions and some cases of validated quadrature, as well as in more theoretical proof work, involving transcendental functions.

Computing a rigorous upper bound on the supremum norm of an approximation error function ε has long been considered a difficult task. While fast numerical algorithms exist, there has been a lack of a validated algorithm. Expecting certified results was out of sight. Several previous proposals in the literature had many drawbacks. The computational time was too high, hence not permitting one to tackle complicated cases involving composite functions or high-degree approximation polynomials. Moreover, the quality of the supremum norm's output was difficult to control. This was either due to the unknown influence of parameters or simply because the techniques required too much manual work.

The supremum norm algorithm proposed in this article solves most of the problems. It is able to compute, in a validated way, a rigorous upper bound for the supremum norm of an approximation error function – in both absolute and relative error cases – with an *a priori* quality. The execution time, measured on real-life examples, is more than encouraging. There is no longer an important overhead in computing a validated supremum norm instead of a mere floating-point approximation without any bound on its error. In fact, the overhead factor is between 3 and 5 only, and the absolute execution time is often less than 1 second on a current machine.

The algorithm presented is based on two important validation steps: the computation of an intermediate polynomial T with a validated bound for the remainder and the proof that some polynomials s_i are non-negative. In this article, several ways of automatically computing an intermediate polynomial with a remainder bound were revised. Special attention was given to how non-negativity of a polynomial could be shown rewriting it as a sum of squares. This technique already permits us not only to validate a non-negativity result but actually to certify it by formally proving it in a formal proof checker.

One point in certification is still outstanding: certification of the intermediate polynomial's remainder bound in a formal proof checker. The algorithms for Taylor models, revised in this article and implemented in a validated way, will have to be "ported" to the environment of a formal proof checker. Previous works like [46] are encouraging and the task does not seem to be technically difficult, the algorithms being well understood. The challenge is in the sheer number of base remainder bounds to be formally proved for all basic functions. Even given general "collateral" proofs in the field of analysis, there would be extensive case-by-case work for each of the basic functions considered. However, we will continue to work on this point in the future.

Another small issue in the proposed validated supremum norm algorithm also needs to be addressed and understood. As detailed in Section 3, the algorithm consists of two steps: first, a numerical computation of a potential upper bound and second, a validation of this upper bound. A detailed timing analysis shows that the first step often takes more than half of the execution time. On the one hand, this observation is encouraging, as it means that computing a validated result for a supremum norm is not much more expensive than computing a numerical approximation. On the other hand, this means that our hypothesis that a lower bound for a supremum norm could be found in negligible time has to be reconsidered. Future work should address that point, finding a way to start with a very rough and quickly available lower bound approximation that gets refined in the course of alternating computation and validation.

Acknowledgements

The authors would like to thank Nicolas Brisebarre, Martin Berz, Guillaume Hanrot, Joris van der Hoeven, Kyoko Makino, Frédéric Messine, Ned Nedialkov, and Markus Neher for their precious advice and help. They also wish to thank the anonymous referees, who made many very accurate remarks and gave fruitful advice for improving this article.

References

[1] M. Abramowitz, I.A. Stegun, Handbook of Mathematical Functions, Dover, 1965.
[2] IEEE Computer Society, IEEE Standard for Floating-Point Arithmetic, IEEE Std 754^{TM}-2008.
[3] F. de Dinechin, Ch. Lauter, G. Melquiond, Assisted verification of elementary functions using Gappa, in: Proceedings of the 21st Annual ACM Symposium on Applied Computing - MCMS Track, vol. 2, ACM, 2006, pp. 1318–1322.
[4] E.W. Cheney, Introduction to Approximation Theory, McGraw-Hill, 1966.
[5] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torrès, Handbook of Floating-Point Arithmetic, Birkhäuser, Boston, 2009.
[6] S. Chevillard, Ch. Lauter, M. Joldeş, Users' manual for the Sollya tool, Release 2.0, https://gforge.inria.fr/frs/download.php/26860/sollya.pdf, April 2010.
[7] N. Revol, F. Rouillier, The MPFI library, http://gforge.inria.fr/projects/mpfi/.
[8] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical recipes in C, in: The Art of Scientific Computing, 2nd edition, Cambridge University Press, 1992.
[9] F. Messine, Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution de problèmes avec contraintes, Ph.D. thesis, INP de Toulouse, 1997.
[10] R.B. Kearfott, Rigorous Global Search: Continuous Problems, Kluwer, Dordrecht, Netherlands, 1996.
[11] E. Hansen, Global Optimization using Interval Analysis, Marcel Dekker, 1992.
[12] N. Revol, Interval Newton iteration in multiple precision for the univariate case, Numerical Algorithms 34 (2) (2003) 417–426.
[13] A. Neumaier, Taylor forms — use and limits, Reliable Computing 9 (1) (2003) 43–79.
[14] S. Chevillard, Évaluation efficace de fonctions numériques. Outils et exemples, Ph.D. thesis, École Normale Supérieure de Lyon, Lyon, France, 2009.
[15] S. Chevillard, Ch. Lauter, A certified infinite norm for the implementation of elementary functions, in: Proc. of the 7th International Conference on Quality Software, 2007, pp. 153–160.
[16] J. Harrison, Floating point verification in HOL light: the exponential function, Technical Report 428, University of Cambridge Computer Laboratory, http://www.cl.cam.ac.uk/users/jrh/papers/tang.ps.gz, 1997.
[17] K. Makino, M. Berz, Taylor models and other validated functional inclusion methods, International Journal of Pure and Applied Mathematics 4 (4) (2003) 379–456. http://bt.pa.msu.edu/pub/papers/TMIJPAM03/TMIJPAM03.pdf.
[18] S. Chevillard, M. Joldeş, Ch. Lauter, Certified and fast computation of supremum norms of approximation errors, in: 19th IEEE SYMPOSIUM on Computer Arithmetic, 2009, pp. 169–176.
[19] W. Krämer, Sichere und genaue Abschätzung des Approximationsfehlers bei rationalen Approximationen, Tech. Rep. 3/1996, Institut für angewandte Mathematik, Universität Karlsruhe, Mar. 1996.
[20] J. Harrison, Verifying nonlinear real formulas via sums of squares, in: Proc. of the 20th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2007, Springer-Verlag, 2007, pp. 102–118.
[21] R.E. Moore, Methods and Applications of Interval Analysis, Society for Industrial and Applied Mathematics, 1979.
[22] M. Berz, K. Makino, COSY INFINITY Version 9.0, http://cosyinfinity.org.
[23] M. Berz, K. Makino, Rigorous global search using taylor models, in: SNC'09: Proceedings of the 2009 Conference on Symbolic Numeric Computation, ACM, New York, NY, USA, 2009, pp. 11–20.
[24] M. Berz, K. Makino, Y.-K. Kim, Long-term stability of the Tevatron by verified global optimization, in: Proceedings of the 8th International Computational Accelerator Physics Conference — ICAP 2004, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment 558 (1) (2006) 1–10.
[25] F. Rouillier, P. Zimmermann, Efficient isolation of polynomial's real roots, Journal of Computational and Applied Mathematics 162 (1) (2004) 33–50.
[26] C. Bendsten, O. Stauning, TADIFF, a Flexible C++ Package for Automatic Differentiation Using Taylor Series, Tech. Rep. IMM-REP-1997-07, Technical University of Denmark, April 1997.
[27] L.B. Rall, The arithmetic of differentiation, Mathematics Magazine 59 (5) (1986) 275–282.
[28] A. Griewank, Evaluating derivatives: principles and techniques of algorithmic differentiation, in: Frontiers in Appl. Math., no. 19, SIAM, Philadelphia, PA, 2000.
[29] R.P. Brent, H.T. Kung, $\mathcal{O}\left((n \log n)^{3/2}\right)$ algorithms for composition and reversion of power series, in: J.F. Traub (Ed.), Analytic Computational Complexity, Academic Press, New York, 1975, pp. 217–225.
[30] R.P. Brent, H.T. Kung, Fast algorithms for manipulating formal power series, Journal of the ACM 25 (4) (1978) 581–595.
[31] L.V. Ahlfors, Complex analysis, in: An Introduction to the Theory of Analytic Functions of One Complex Variable, 3rd Edition, McGraw-Hill, New York, 1979.
[32] I. Eble, M. Neher, ACETAF: A software package for computing validated bounds for Taylor coefficients of analytic functions, ACM Transactions on Mathematical Software 29 (3) (2003) 263–286.
[33] M. Mezzarobba, B. Salvy, Effective bounds for P-recursive sequences, Tech. Rep., arXiv, 2009.
[34] K. Makino, Rigorous analysis of nonlinear motion in particle accelerators, Ph.D. thesis, Michigan State University, East Lansing, Michigan, USA, 1998.
[35] R. Zumkeller, Global optimization in type theory, Ph.D. thesis, École Polytechnique, 2008.
[36] J.S. Moore, T. Lynch, M. Kaufmann, A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division program, IEEE Transactions on Computers 47 (1998) 913–926.
[37] D. Russinoff, A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions, LMS Journal of Computation and Mathematics 1 (1998) 148–200. Available on the web at http://www.russinoff.com/papers/k7-div-sqrt.html.
[38] J. O'Leary, X. Zhao, R. Gerth, C.-J.H. Seger, Formally verifying IEEE compliance of floating-point hardware, Intel Technology Journal 1999-Q1 (1999) 1–14. Available on the web at http://download.intel.com/technology/itj/q11999/pdf/floating_point.pdf.
[39] R. Kaivola, M.D. Aagaard, Divider circuit verification with model checking and theorem proving, in: M. Aagaard, J. Harrison (Eds.), Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000, in: Lecture Notes in Computer Science, vol. 1869, Springer-Verlag, 2000, pp. 338–355.
[40] J. Harrison, Formal verification of floating point trigonometric functions, in: W.A. Hunt, S.D. Johnson (Eds.), Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000, in: Lecture Notes in Computer Science, vol. 1954, Springer-Verlag, 2000, pp. 217–233.
[41] C. Jacobi, Formal verification of a fully IEEE compliant floating point unit, Ph.D. thesis, University of the Saarland, available on the web at http://engr.smu.edu/~seidel/research/diss-jacobi.ps.gz, 2002.
[42] S. Boldo, Preuves formelles en arithmétiques à virgule flottante, Ph.D. thesis, ENS Lyon, available on the web at http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2004/PhD2004-05.pdf, 2004.

[43] R.J. Boulton, Efficiency in a fully-expansive theorem prover, Technical Report 337, University of Cambridge Computer Laboratory, UK (Ph.D. thesis), 1994.
[44] M. Blum, Program result checking: a new approach to making programs more reliable, in: Automata, Languages and Programming, 20th International Colloquium, ICALP93, Proceedings, in: Lecture Notes in Computer Science, vol. 700, Springer-Verlag, 1993, pp. 1–14.
[45] K. Mehlhorn, S. Neher, M. Seel, R. Seidel, T. Schilz, S. Schirra, C. Uhrig, Checking geometric programs or verification of geometric structures, in: Proceedings of the 12th Annual Symposium on Computational Geometry, FCRC'96, Association for Computing Machinery, Philadelphia, 1996, pp. 159–165.
[46] R. Zumkeller, Formal global optimization with Taylor models, in: Proc. of the 4th International Joint Conference on Automated Reasoning, 2008, pp. 408–422.
[47] F. Cháves, M. Daumas, A library of Taylor models for PVS automatic proof checker, CoRR abs/cs/0602005.
[48] H. Ehlich, K. Zeller, Schwankung von Polynomen zwischen Gitterpunkten, Mathematische Zeitschrift 86 (1) (1964) 41–44.
[49] G. Melquiond, Floating-point arithmetic in the Coq system, in: Proc. of the 8th Conference on Real Numbers and Computers, 2008, pp. 93–102.
[50] P.A. Parrilo, Semidefinite programming relaxations for semialgebraic problems, Mathematical Programming, Series B 96 (2003) 293–320.
[51] X. Gourdon, Combinatoire, algorithmique et géometrie des polynômes, Ph.D. thesis, École Polytechnique, Paris, France, 1996.
[52] M.-F. Roy, Basic algorithms in real algebraic geometry and their complexity: from Sturm's theorem to the existential theory of reals, in: F. Broglia (Ed.), in: Expositions in Mathematics, vol. 23, de Gruyter, 1996, Lectures in Real Geometry.