Rigorous Polynomial Approximations and Applications

Mioara Maria Joldes

N° d'ordre :
N° attribué par la bibliothèque :

ÉCOLE NORMALE SUPÉRIEURE DE LYON
Laboratoire de l'Informatique du Parallélisme

THÈSE

présentée et soutenue publiquement le 26 Septembre 2011 par

Mioara JOLDEȘ

pour l'obtention du grade de

Docteur de l'École Normale Supérieure de Lyon

spécialité : Informatique

au titre de l'École Doctorale de Mathématiques et d'Informatique Fondamentale de Lyon

Approximations polynomiales rigoureuses et applications

Directeurs de thèse : Nicolas BRISEBARRE
Jean-Michel MULLER

Après avis de : Didier HENRION
Warwick TUCKER

Devant la commission d'examen formée de :

Frédéric BENHAMOU
Nicolas BRISEBARRE
Didier HENRION
Jean-Michel MULLER
Warwick TUCKER

For Soho, where I belong.

Acknowledgements

I cite G. Cantor, for saying that *the art of proposing a question must be held of higher value than solving it* and express my deepest gratitude and appreciation for my advisors. Nicolas found me [*] during an exam in a Master 2 course at ENS, and proposed the right question for me. Although the answer was not given during my master, as initially thought, although there were moments when I wanted different questions, he encouraged me every day, he believed in me, he was always available when I needed his advice in scientific and non-scientific matters. His altruism when sharing his knowledge with me, his desire for perfection when carefully reviewing all my work, were always astonishing and motivating for me. This thesis, and the nice time I had, owes a lot to his devotion, ideas, suggestions [†] and so good sense of humor (not the Romanians' related jokes, though!). Jean-Michel always kept an experienced and kind eye over my work. I felt honored to have as supervisor the best expert in Computer Arithmetic: he answered all my related questions and he always made so good and often mediating suggestions [‡]. It was a great chance for me to have had advisors who guided my path, but let me make my own choices and find my own values and interests in research. I thank them for all that, and for having made *une docteur* out of *une chipie*. Merci beaucoup, Nicolas et Jean-Michel.

I want to thank Frederic Benhamou, Didier Henrion and Warwick Tucker for having accepted to participate in my jury, for their pertinent questions and useful suggestions. I would like to express my gratitude to the reviewers of this manuscript. I thank them for their hard work, for their understanding and availability regarding the schedule of this work. It seems that in what follows I will have the chance and the honor to work more with them. I think that they brought tremendous opportunities in my young researcher life and I thank them for that also.

Moreover, I would like to thank my collaborators: *supnorm* and *Sollya* (and *big brothers*) Sylvain and Christoph, for having helped me with their hard work, bright ideas and (endless, since I don't give in either) debates; Bogdan and Florent, who let me keep in touch with my old love, the FPGAs; Alexandre and Marc for having shared with me the magic of *D-finite* functions; Érik, Ioana and Micaela for their formalization efforts in our *Coqapprox* meetings; Serge for his unbelievable kindness.

I would also like to thank the members of the Arenaire team for having made my stay there an unforgetable experience: my office mates Andy, Guillaume, Ivan and Xavier for having listened to all my usual bla-bla and for having accepted my bike in the office; Claude-Pierre (thanks a lot for your work for the *Projet Région* that financed my scholarship!), Damien (thanks for the beautiful Euclidean Lattices course that led me to Arenaire!), Gilles (thanks for our short but extremely meaningful discussions!), Guillaume (many thanks for your advices and help!), Nathalie (thanks for often pointing me interesting conferences!), Nicolas L. (thanks for the fun I had teaching and joking with you!), Vincent (thanks for the 17 obsession!), and our assistants Damien and Séverine for their great help with *les ordres de mission*. I also thank Adrien, Christophe, Diep, Fabien, Eleonora, Nicolas Brunie, Jingyan, Adeline, Philippe, David and Laurent-Stéphane for their help

*. *une pioupiou*

†. *Pas de calendriers et de répétitions, quand même !*

‡. *Some of us would be still debating today, otherwise.*

in my Arenaire day-by-day life.

Then, I am very grateful to my external collaborators who helped and inspired me: Marius Cornea, John Harrison, Alexander Goldsztejn, Marcus Neher, and my new team CAPA at Uppsala.

I would like to say a "pure" *Mulțam fain!* to my ENS Romanian mafia for being there for me, for our coffees, lunches (sorry I always had to leave earlier!) and *ieșit la scări*. I adored our non-scientific and not politically correct discussions and jokes.

My thesis would not exist without the basis of knowledge I accumulated during my studies in Romania. I will always be grateful to my teachers there. Especially, I would like to thank Octavian Creț not only for having sent the e-mail announcing the scholarships at ENS, but also for all his hard and fair work in difficult and unfair times. I thank also my professors Alin Suciu, Ioan Gavrea and Dumitru Mircea Ivan who impressed my student mind with their unique intelligence, modesty and humor.

I would like to give my special thanks to Bogdan (we made it, we're doctors!) and my family (parents, grand-parents, dear sister Oana and Dia). Their total support, care and love enabled me to complete this work. I thank also my cousins Călin and Claudia for their help and support and long trip to France.

Last but not least I thank the ENS janitor who brought me the hot chocolate the night before my defense and my father for having taught me the beauty of mathematics. Their simple kindness keeps me going on.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Rigorous computing (sometimes called *validated computing* as well) is the field that uses numerical computations, yet is able to provide rigorous mathematical statements about the obtained result. This area of research deals with problems that cannot or are difficult and costly in time to be solved by traditional mathematical formal methods, like problems that have a large search space, problems for which closed forms given by symbolic computations are not available or too difficult to obtain, or problems that have a non-linear ingredient (the output is not proportional to the input). Examples include problems from global optimization, ordinary differential equations (ODE) solving or integration. While such hard problems could be well-studied by numerical computations, with the advent of computers in scientific research, however, one lacks mathematical formal statements about their solutions. Consider a simple definite integral example, taken from Section "Numerical Mathematics in Mathematica" in The Mathematica Book [170].

Example 1.0.1. *Compute* $\int_0^1 \sin(\sin x)\mathrm{d}x$.

In this case, there is no symbolic "closed-formula" for the result, so the answer is not known exactly. Numerical computations come in handy, and we can obtain a numerical value for this integral: $0.430606103120690604912377\ldots$. However, quoting from the same book, "an important point to realize is that when Mathematica does a numerical integral, the only information it has about your integrand is a sequence of numerical values for it.[...] If you give a sufficiently pathological integrand, these assumptions may not be valid, and as a result, Mathematica may simply give you the wrong answer for the integral." How can we then be sure of the number of digits in the answer that are correct? How can we validate from a mathematical point of view what we have computed? Maybe checking against other software could be useful. Indeed, in this case Maple15 [110], Pari/GP [156] or Chebfun developed in Matlab [160] give us the same, say, 10 digits. So, maybe this could be enough for some to conclude that the first 10 digits are correct.

But consider another example that is a little bit trickier:

Example 1.0.2. $\int_0^3 \sin\left(10^{-3} + (1-x)^2\right)^{-3/2}\mathrm{d}x$.

Maple15 returns * 10 significant digits: 0.7499743685, but fails to answer if we ask for more †,

*. the code used is: `evalf(int(sin((10^(-3) + (1-x)^2)^(-3/2)), x=0..3)).`
†. the code used is: `evalf(int(sin((10^(-3) + (1-x)^2)^(-3/2)), x=0..3),15).`

Pari/GP [156] gives $0.79277309714790807555009783 54$, Mathematica * and Chebfun † fail to answer. Moreover, the article [32] where this example was taken from claims that the first 10 digits are 0.7578918118. What is the correct answer, then ?

Another interesting example that we were faced with comes directly from a problem, discussed in detail in Chapter 3, that developers of mathematical libraries address. Roughly speaking, the problem is the following: we are given a function f over an interval $[a, b]$ and a polynomial p that approximates f over $[a, b]$. Given some accuracy parameter $\bar{\eta}$, we want to compute bounds $u, \ell \geqslant 0$ such that $\ell \leq \sup_{a \leqslant x \leqslant b} |f(x) - p(x)| \leq u$ and that $\left|\frac{u-\ell}{\ell}\right| \leqslant \bar{\eta}$. We can also consider that we have obtained u, ℓ by some numerical means and we want simply to check the above inequality. Roughly speaking, $-\log_{10} \bar{\eta}$ quantifies the number of correct significant digits obtained for $\sup_{a \leqslant x \leqslant b} |f(x) - p(x)|$. A typical numerical example, which we slightly simplified for expository purposes, is the following:

Example 1.0.3. *Let* $[a, b] = [-205674681606191 \cdot 2^{-53}; 205674681606835 \cdot 2^{-53}]$,

$$f(x) = \operatorname{asin}(x + 770422123864867 \cdot 2^{-50}),$$

$p(x) = 1565177036271399720760797210697274 5 \cdot 2^{-114} + 3476698806776688943652103662933 \cdot 2^{-101}x + 1789497250031118708226980770517 1 \cdot 2^{-104}x^2 + 1269766072964410252693451531025 91 \cdot 2^{-106}x^3 + 2491073788955624134951519440427 99 \cdot 2^{-106}x^4 + 1390539516497963047681499952255 89 \cdot 2^{-104}x^5 + 1654286641682512495018879218888 47 \cdot 2^{-103}x^6 + 2061676018738841632810986186311 59 \cdot 2^{-102}x^7 + 6638661126013334729551039065309 9 \cdot 2^{-99}x^8 + 2433556521489987 \cdot 2^{-43}x^9 + 409716955440671 \cdot 2^{-39}x^{10} + 2242518346998655 \cdot 2^{40}x^{11} + 3108616106416871 \cdot 2^{-39}x^{12} + 4356285307071455 \cdot 2^{-38}x^{13} + 6161286268548935 \cdot 2^{-37}x^{14} + 8783550111623067 \cdot 2^{-36}x^{15} + 788026560267325 \cdot 2^{-31}x^{16} + 1138037795125313 \cdot 2^{-30}x^{17} + 3304615966282565 \cdot 2^{30}x^{18} + 602367826671283 \cdot 2^{-26}x^{19} + 1765006192104851 \cdot 2^{-26}x^{20} + 1337636086941861 \cdot 2^{-24}x^{21} + 986777691264547 \cdot 2^{-22}x^{22}$.

Compute $\max_{x \in [a,b]} |f(x) - p(x)|$ *with* 6 *correct significant digits.*

We will see in Chapter 3 that this kind of problem can be reduced for instance to the following:

Prove that $0.194491 \cdot 10^{-34} \leqslant \max_{x \in [a,b]} |f(x) - p(x)| \leqslant 0.1944913 \cdot 10^{-34}$.

Authors of very efficient numerical software warn us [159]: "It is well known that the problem of determining the sign of a difference of real numbers with guaranteed accuracy poses difficulties. However, the chebfun system makes no claim to overcome these difficulties", or "A similar problem arises when you try to find a numerical approximation to the minimum of a function. Mathematica samples only a finite number of values, [...] and you may get the wrong answer for the minimum." [170, Chap. 3.9.2].

This problem of proving inequalities occurs in another famous example from the area of computer assisted proofs: the project of formally proving the Kepler's conjecture [75]. One example of such an inequality necessary in this proof is the following:

Example 1.0.4. *Prove the inequality:*

$2\pi - 2x \operatorname{asin}((\cos 0.797) \sin(\pi/x)) > 0.591 - 0.0331x + 1.506$, *where* $3 < x < 64$.

Finally we quote [170]: "In many calculations, it is therefore worthwhile to go as far as you can symbolically, and then resort to numerical methods only at the very end. This gives you the best chance of avoiding the problems that can arise in purely numerical computations."

*. the code used is: `NIntegrate[Sin[(10^(-3)+(1-x)^2)^(-3/2)],x,0,3, Method->Oscillatory]`.

†. the code used is: `F = @(t) sin((10^(-3) + (1-t).^2).^(-3/2)); f = chebfun(F,[0,3]);`.

In fact, with rigorous computing we aim at combining efficiently symbolic and numeric computations in order to benefit from the speed of numerical computations, but to guarantee in the end mathematical statements about the results. In this way, rigorous computing bridges the gap between scientific computing and pure mathematics, between speed and reliability.

The most courageous dream would be to be able to rigorously solve any problem that can be solved numerically by efficient existing software, and even more. However, we will see throughout this work that adaptation of numerical algorithms to rigorous computing is not straightforward in general. So our reasonable purpose is two-fold:

On the one hand we explain and exemplify how existing rigorous computing tools can be used to solve practical problems that we encountered, with a major emphasis on the field of Computer Arithmetic. This is mainly due to the fact that this thesis was developed in Arenaire project [*] which aims at elaborating and consolidating knowledge in the field of Computer Arithmetic. Improvements are sought in terms of reliability, accuracy, and speed for the available arithmetic, at the hardware level as well as at the software and algorithmic levels, on computers, processors, dedicated or embedded chips.

On the other hand, when these tools fail, we try to improve them, design new ones and apply them to a larger spectrum of practical problems. The examples we gave above can be solved today with our tools. Some more complicated ones may be not. It is one of our future projects to extend these tools to broader ranges of problems.

With this in mind, we restrict our presentation to the one-dimensional setting. This is due mainly to the fact that the results presented in this work deal only with univariate functions.

When using validated computations, the aim is not only to compute *approximations of the solution*, but also and more importantly, *enclosures* of the same. The width of such an enclosure gives a direct quality measurement of the computation, and can be used to adaptively improve the calculations at run-time. A major field for computing approximate solutions is to use polynomial approximations. We consider appropriate at the beginning of this work to briefly recall some essential theoretical results in the subject of polynomial approximation and to give a flavor of why and how *rigorous computing* articulates with polynomial approximations to give *rigorous polynomial approximations* - the subject of this thesis.

1.1 Introduction to rigorous polynomial approximations - outline of the thesis

It is very useful to be able to replace any given function by a simpler function, such as a polynomial, chosen to have values not identical with but very close to those of the given function, since such an *approximation* may be more *compact* to represent and store but also more *efficient to evaluate and manipulate*. Usually, an *approximation problem* consists of three components:

- A function f to be approximated which usually belongs to a *function class* Ω. Generally, the functions we will work with are real functions of one real variable (unless specifically stated otherwise). In order to isolate certain properties of these functions and the norms we use, we consider several function classes, such as: continuous functions on $[a, b]$ denoted by $\mathcal{C}[a, b]$; bounded functions on $[a, b]$ denoted by $\mathcal{L}_\infty[a, b]$; square-integrable functions on $[a, b]$ denoted by $\mathcal{L}_2[a, b]$; functions that are solutions of linear differential equations with polynomial coefficients called D-finite functions, etc. For theoretical purposes it is usually desirable to choose the function class Ω to be a vector space (or linear space).
- A type of approximation, which in this work is *polynomial* [†]. We consider a family of polyno-

*. http://www.ens-lyon.fr/LIP/Arenaire/

†. rational fractions could also be considered - this is well-known under the name of rational approximations

mials $\mathcal{P}$ and we search for approximations $p \in \mathcal{P}$. Usually $\mathcal{P}$ is a subspace of Ω. For example, given $n \in \mathbb{N}$ we can consider the family $\mathcal{P}_n = \{p(x) \in \mathbb{R}[x], \deg p \leqslant n\}$ of polynomials with real coefficients of degree at most n.

– A norm (of the approximation error), in terms of which the problem may be formally posed. Denoted by $||\cdot||$, the norm serves to compare the function f with p, and gives a single scalar measure of the closeness of p to f, namely: $||f - p||$.

Definition 1.1.1. *A norm $||\cdot||$ is defined as any real scalar measure of elements of a vector space that satisfies the axioms:*

1. $||u|| \geqslant 0$, *with equality if and only if* $u \equiv 0$;
2. $||u + v|| \leqslant ||u|| + ||v||$ *(the triangle inequality);*
3. $||\alpha u|| = |\alpha|\, ||u||$, *for any scalar* α.

In this work, standard choices of norms for function spaces are the following:

– $\mathcal{L}_\infty$ norm (or uniform norm, minimax norm, or Chebyshev norm): $\|f\|_\infty = \sup\limits_{a \leqslant x \leqslant b} |f(x)|$;

– $\mathcal{L}_2$ norm (or least-squares norm, or Euclidean norm): $||f||_2 = \sqrt{\int\limits_a^b |f(x)|^2 w(x) \mathrm{d}x}$, where w is a given continuous and non-negative weight function.

Once the norm is chosen, we can quantify the quality of the approximation p to f. In the sequel we use [102, Definition 3.2.]:

Definition 1.1.2. *Good, best, near-best approximations*

– *An approximation $p \in \mathcal{P}$ is said to be* good *if $||f - p|| \leqslant \varepsilon$, where ε is a prescribed absolute accuracy.*

– *An approximation $p^* \in \mathcal{P}$ is said to be* best *if, for any other approximation $p \in \mathcal{P}, ||f - p^*|| \leqslant ||f - p||$. This best approximation is not necessarily unique.*

– *An approximation $p \in \mathcal{P}$ is said to be* near-best *within a relative distance ρ if, $||f - p|| \leqslant (1 + \rho)||f - p^*||$, where p^* is a best approximation to f in $\mathcal{P}$.*

We remark that in the definition above we used the absolute error: $\varepsilon_{\text{abs}} = f - p$. However, in some contexts in the subsequent chapters we will use the relative error criterion $\varepsilon_{\text{rel}} = 1 - p/f$.

Usually polynomials $p \in \mathcal{P}$ are represented by their coefficients in a certain polynomial basis. For example, the most common representation for a polynomial is in monomial basis $p(x) = \sum\limits_{i=0}^{n} p_i x^i$. Several other bases are available (Chebyshev, Newton, Bernstein) and will be discussed in this work.

However, whatever basis we chose, when implementing approximation schemes in machine, the real-valued coefficients of the obtained approximation polynomial will not in general be exactly representable. Hence, we also need as a key element the *machine representable* format $\mathcal{F}$ of the coefficients. Several formats are available and we will discuss them in the sequel. Examples are: basic hardware formats available in almost all processors today like floating-point numbers; custom precision (constrained) hardware formats used in custom architectures; more intricate formats that require intermediate micro-code or software routines like multiple-precision floating-point numbers or multiple-precision intervals; even broader formats provided by existent Computer Algebra Systems (CAS) like Maple, equipped with symbolic computation formats.

Moreover, when implementing approximation routines with only a finite *machine representable* format $\mathcal{F}$ for underlying computations, instead of real numbers, rounding and truncation errors occur. The purpose of this thesis is to provide algorithms, theoretical proofs and software tools for computing both a polynomial approximation with coefficients in the format $\mathcal{F}$ (for a given representation basis) and a measure of the quality of approximation that is a *rigorous* "good" upper-bound B for the approximation error (absolute or relative): $||\varepsilon_{\text{abs/rel}}|| \leqslant B$. We note that the defini-

tion of "good" upper-bound B is that either the absolute or relative distance between $||\varepsilon_{\text{abs/rel}}||$ and B is within some prescribed accuracy level that depends on the kind of application we consider.

More formally, the general rigorous approximation problem dealt with in this thesis is:

Problem 1. *Rigorous polynomial approximation (RPA)*

Let f be a function belonging to some specified function class Ω over a given interval $[a, b]$ and let $\mathcal{P}$ a specified family of polynomials with coefficients (in a given basis) exactly representable in some specified format $\mathcal{F}$. Find the coefficients of a polynomial approximation $p \in \mathcal{P}$ together with a "good" bound B such that $||f - p|| \leqslant B$.

Usually we will give the answer to this problem in the form of a couple (p, B) that we call *rigorous polynomial approximation*. The purpose of giving this general name and definition is two-fold:

- On the one hand, there are already in literature several works that gave answers to particular instances of Problem 1 depending both on the specified parameters: function class Ω, family of polynomials $\mathcal{P}$, coefficient format $\mathcal{F}$ and on the mathematical approximation method or idea used to obtain p and B. Examples include: Taylor Models and Taylor Forms where the name suggests that Taylor approximations are used in the process; Ultra-Arithmetic where there is an idea of an analogy between numbers and function spaces arithmetic; Higher Order Inclusions where a polynomial of degree higher than 1 is used with the purpose of computing an enclosure of the image of the function, etc.
  To that extent, our purpose is to describe, *disambiguate*, implement and compare these tools.
- On the other hand, our main contributions are to solve this problem using other polynomial approximations different from Taylor series. We analyze the possibility of using other such approximations, we design a new tool based on Chebyshev series approximations.

We give below examples representative for each of the following chapters:

- In Chapter 2 we deal with RPAs based roughly on Taylor series. The RPAs we are able to obtain using this method will be given in Item (1) of the following examples.
- In Chapter 3 we deal with RPAs based on best polynomial approximations. This method is more computationally expensive, in particular it uses an intermediary RPA obtained in Chapter 2 and will be treated in Item (2). Example 1.0.3 was such an approximation.
- In Chapter 4 we deal with RPAs based on near-best polynomial approximations in Chebyshev basis and will be treated in Item (3) of the following examples. The first three chapters consider functions given explicitly by an expression.
- In Chapter 5 we deal with RPAs based on near-best polynomial approximations in Chebyshev basis for functions that are given as solutions of ordinary differential equations with polynomial coefficients (D-finite functions). This case is presented in Example 1.1.6.

For all examples in this section, the approximation error $\varepsilon = f - p$ is plotted numerically for each approximation. We show in Example 5.7.4 how to obtain rigorous plots for functions. The norm considered is the supremum norm.

Let us start with the most trivial example:

Example 1.1.3. *Let $f = \exp$, over $[0, 1]$.*

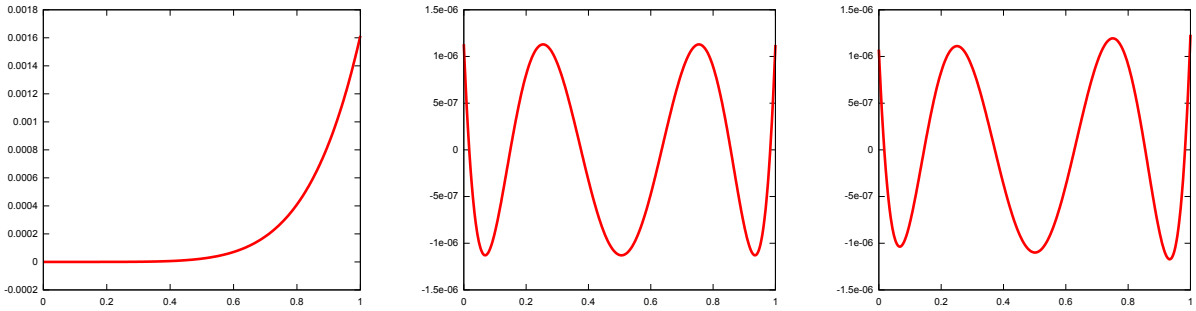
(0) We consider first polynomials of degree 5, with real coefficients, expressed in monomial basis. Then the common Taylor approximation $p = \sum_{i=0}^{5} \frac{1}{i!} x^i$ together with the bound $B = \frac{\exp(1)}{6!}$ is a rigorous polynomial approximation of f.

(1) Of course, instead of real coefficients, one usually requires coefficients exactly representable in some machine format. In the item above, we change the required format to coefficients exactly representable in `binary32` *format $\mathcal{F}_{24} = \{2^E \cdot m | E \in \mathbb{Z}, m \in \mathbb{Z}, 2^{23} \leq |m| \leq 2^{24} - 1\} \cup \{0\}$, with the required bounds on E (see Table 1.1, Definition 1.2.2). Based simply on Taylor approximation, truncation of*

the real coefficients and bounding of the truncation errors, one obtains $p = 1+x+2^{-1}\cdot x^2+5592405\cdot 2^{-25}\cdot x^3+5592405\cdot 2^{-27}\cdot x^4+1118481\cdot 2^{-27}x^5$ *together with the bound* $B = 1.6155438\cdot 10^{-3}$.

(2) *In Chapter 3, we will see how to obtain rigorous best polynomial approximations. In this case, maintaining the requirements given above, one obtains:* $p = 16777197\cdot 2^{-24}+8389275\cdot 2^{-23}x+4186719\cdot 2^{-23}x^2+1429441\cdot 2^{-23}\cdot x^3+9341529\cdot 2^{-28}x^4+1866159\cdot 2^{-27}x^5$ *together with the bound* $B = 1.13248836\cdot 10^{-6}$.

(3) *Changing the representation basis is also possible. In Chapter 4, we will see how to obtain near-best rigorous polynomial approximations in Chebyshev basis. Here, for the sake of simplicity and just to show the near-minimax character, we give the polynomial transformed back to monomial basis. Still using* `binary32` *coefficients, we are able to find:* $p = 8388599\cdot 2^{-23}+8389241\cdot 2^{-23}x+16747961\cdot 2^{-25}x^2+11429449\cdot 2^{-26}x^3+2342335\cdot 2^{-26}x^4+14884743\cdot 2^{-30}x^5$, $B = 1.245354\cdot 10^{-6}$.



(a) RPA-Taylor (b) RPA-Minimax (c) RPA-Chebyshev

Figure 1.1: Approximation error $\varepsilon = f - p$ for each RPA given in Example 1.1.3.

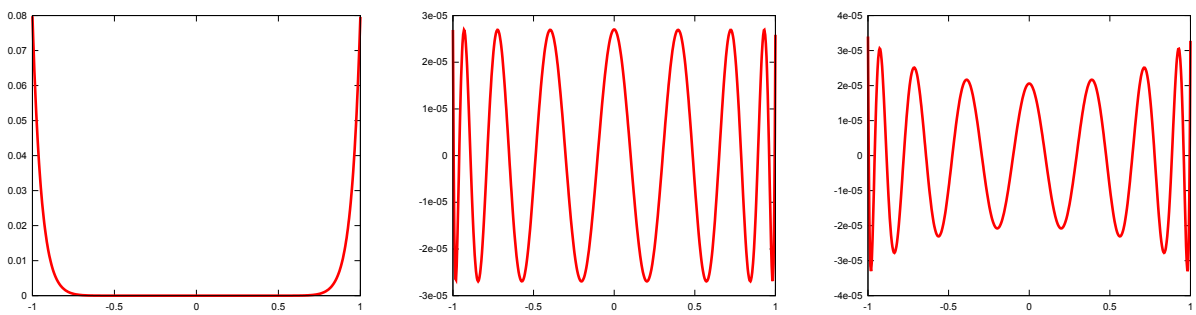
A more challenging example could be for a function that is given by a more complicated expression:

Example 1.1.4. $f = \exp(1/\cos(x))$, *over* $[-1, 1]$, *considering approximation polynomials of degree* 15, *with* `binary32` *coefficients. For simplicity, we refrain from giving the actual polynomial coefficients in each of the situations below and just plot the error* ε *in Figure 1.2.*

(1) *The rigorous error bound obtained with Taylor approximations is* $B = 0.2326799$.

(2) *In the case of best approximations,* $B = 2.702385\cdot 10^{-5}$.

(3) *In the case of near-best approximations in Chebyshev basis,* $B = 5.343836\cdot 10^{-5}$.



(a) RPA-Taylor (b) RPA-Minimax (c) RPA-Chebyshev

Figure 1.2: Approximation error $\varepsilon = f - p$ for each RPA given in Example 1.1.4.

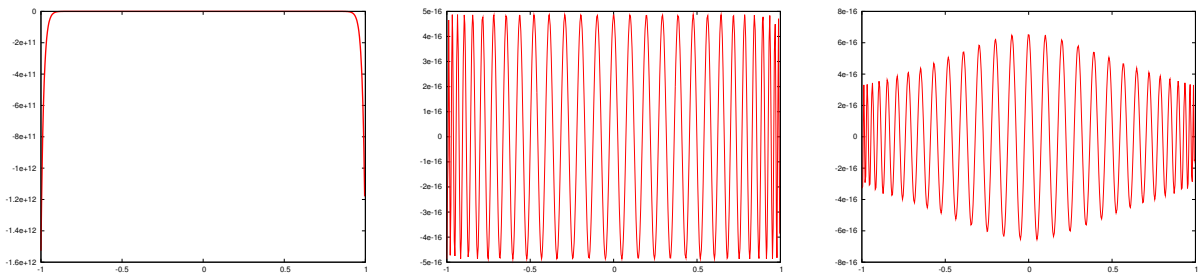
We can now consider even more complicated functions, like for example a function which is infinitely differentiable on the real interval considered, but not analytic in the whole complex disc containing this interval (see Section 4.2.3 for more details).

Example 1.1.5. *$f = \exp\left(\frac{1}{1+2x^2}\right)$, over $[-1, 1]$, considering approximation polynomials of degree 60, with 165 bits multiple precision floating-point coefficients. Like in the above example, we plot the error ε in Figure 1.3 for each of the following cases:*

(1) Taylor approximations are not convergent over $[-1, 1]$. Using our method, the bound obtained is $B = +\infty$.

(2) In the case of best approximations, the rigorous bound can be obtained using algorithms that resort to interval subdivisions and to intermediary approximations computed with the method of Chapters 2 or 4 and it is: $B = 0.486981 \cdot 10^{-15}$.

(3) In the case of near-best approximations in Chebyshev basis, using methods in Chapter 4, we directly obtain some finite bound $B = 2.0269917 \cdot 10^{-3}$ over the whole interval, but it is still highly over-estimated.



(a) RPA-Taylor (b) RPA-Minimax (c) RPA-Chebyshev

Figure 1.3: Approximation error $\varepsilon = f - p$ for each RPA given in Example 1.1.5.

Finally, we can consider classes of functions that are not defined explicitly. For example, we address the problem of computing near-best rigorous polynomial approximations for D-finite functions:

Example 1.1.6. *Let f be the solution of the following differential equation $4xy(x) + (1 + 4x^2 + 4x^4)y'(x)$, $y(0) = \exp(1)$ and $[-1, 1]$ the interval considered. In Chapter 5 we will see that we can compute near-best rigorous polynomial approximations in Chebyshev basis for such D-finite functions. We note that the exact solution would be in this case $f(x) = \exp(1/(1 + 2x^2))$. For example, we plotted in Figure 1.4 the error between a polynomial of degree 60 whose coefficients are rational numbers and f over $[-1, 1]$. The rigorous bound obtained is: $B = 0.145 \cdot 10^{-14}$.*

Sometimes, when p is already given for example by some numerical routine, we name by *certifying the approximation*, the process of computing B. We note that *certified results* is a common term used in conjunction with formal proof assistants [167, 11]. This work is not centered on formally proving such approximations, although in Chapter 3 we prove formally some parts of our algorithms and we intend to do so for the remaining parts. As we will see in the following chapters, the algorithms we use are sufficiently simple for allowing their formal proof. It is essentially a matter of implementation and one of our current goals in the TaMaDi project [116] is to have formally proven polynomial approximations in COQ [43] using the algorithms presented in this work.

Complexity model. We have seen in the previous examples that the format of the coefficients is variable. Similarly, the computations involved in algorithms given in subsequent chapters are

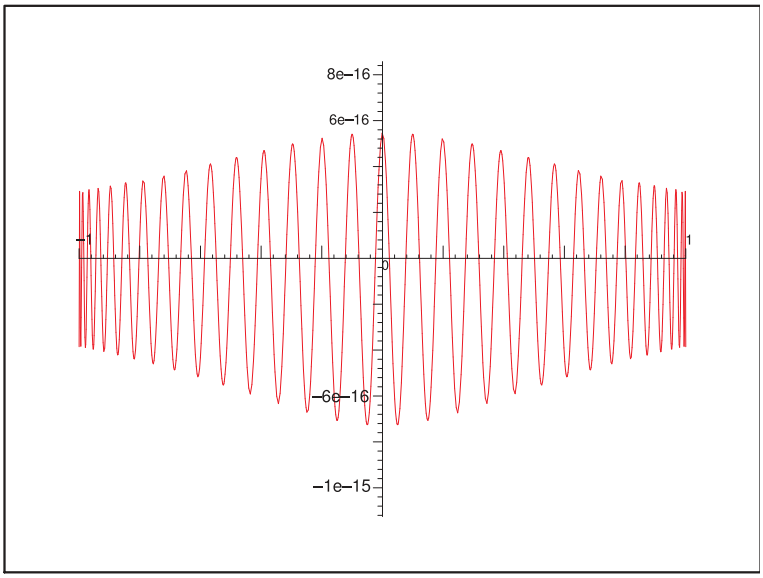


Figure 1.4: Approximation error $\varepsilon = f - p$ for the RPA given in Example 1.1.6.

also carried out using different numerical formats. To account for this variability in the underlying arithmetic, we assess the complexity of the algorithms in the *arithmetic model*. In other words, we only count basic operations in $\mathbb{Q}$, while neglecting both the size of their operands and the cost of accessory control operations.

Let us now turn to some theoretical fundaments on polynomial approximation to briefly see why, when and how *good* polynomial approximations can be obtained.

In approximating $f \in \mathcal{C}[a,b]$ by polynomials on $[a,b]$, it is always possible to obtain a *good* approximation by taking the degree high enough. This is the conclusion of the well-known theorem of Weierstraß:

Theorem 1.1.7. *Weierstraß's Theorem*

For any given f in $\mathcal{C}[a,b]$ and for any given $\varepsilon > 0$, there exists a polynomial p_n of degree n for some sufficiently large n such that $\|f - p_n\|_\infty < \varepsilon$.

A constructive proof of this theorem was given by Bernstein who constructed for a given $f \in \mathcal{C}[0,1]$ a sequence of polynomials $(B_n f)_{n\geqslant 1}$ (now called Bernstein polynomials):

$$B_n f = \sum_{k=0}^{n} f\left(\frac{k}{n}\right)\binom{n}{k} x^k (1-x)^{n-k},$$

that is proven to converge to f in the sense of the uniform norm. A detailed proof of this theorem can be found in [33, Chap.3]. We note that another constructive sequence of polynomials that are proven to be uniformly convergent towards $f \in \mathcal{C}[-1,1]$ on $[-1,1]$ are the Cesaro sums of its Chebyshev series expansion (see [102, Theorem 5.8]).

From the point of view of efficiency we would most of the time like to keep polynomials degree as low as possible, so we will focus in what follows on best or near-best approximations.

Best uniform polynomial approximations Let $f \in \mathcal{C}[a,b]$ and $n \in \mathbb{N}$ given. The best uniform polynomial approximation $p^* \in \mathcal{P}_n$ as defined in Definition 1.1.2 can be viewed as the solution of the following optimization problem:

$$\text{Find } p \in \mathcal{P}_n \text{which minimizes } \|f - p\|_\infty . \tag{1.1}$$

It was theoretically proven probably for the first time by Kirchberger [86] that we can always find such a polynomial p^* and that it is unique. We refrain from giving proofs and refer the interested reader to [Chap. 7][133]. The polynomial p^* is also called *minimax* polynomial. The name *minimax* comes form the rewriting of Equation (1.1) as:

$$\text{Find } p \in \mathcal{P}_n \text{which minimizes } \max_{a \leqslant x \leqslant b} |f(x) - p(x)|. \tag{1.2}$$

The following constructive powerful theorem characterizes minimax approximations:

Theorem 1.1.8. *Alternation theorem*
For any f in $\mathcal{C}[a,b]$ a unique minimax polynomial approximation $p_n^* \in \mathcal{P}_n$ exists, and is uniquely characterized by the alternating property *(or* equioscillation property*) that there are at least $n+2$ points in $[a,b]$ at which $f - p_n^*$ attains its maximum absolute value (namely $\|f - p_n^*\|_\infty$) with alternating signs.*

This theorem, due to Borel (1905), asserts that, for p_n^* to be the best approximation, it is both *necessary and sufficient* that the alternating property should hold, that only one polynomial has this property, and that there is only one best approximation.

Based on this property, the Soviet mathematician Remez designed in 1934 an iterative algorithm for computing the minimax polynomial $p_n^* \in \mathcal{P}_n$. This algorithm stands out as a nontrivial one conceived for solving a challenging computational problem and it was developed before the advent of computers. For implementations we refer the reader to [126, 35].

Example 1.1.9. *In Figures 1.1(b), 1.2(b), 1.3(b), the error between the approximated function and the minimax polynomial obtained with Sollya software [37] was plotted.*

We recall that in the above examples we said that we should be able to compute both the coefficients of p_n^* and a rigorous upper-bound B for $\|f - p_n^*\|_\infty$, while most numerical routines are concerned only with the fast computation of the coefficients. So, in most of the cases, we can not follow just the well-known ways of classical numerical algorithms. For Remez algorithm the convergence is proved to be quadratical (under some reasonable conditions) [165]. But what about bounding $\|f - p_n^*\|_\infty$?

A theoretical result proved by Bernstein [9] provides a closed error-bound formula:

Theorem 1.1.10. *Let $n \in \mathbb{N}$ fixed and a function $f \in \mathcal{C}^{(n+1)}[-1,1]$. Let p_n^* be the best uniform approximation polynomial to f over $[-1,1]$, of degree at most n. Then there exists $\xi \in (-1,1)$ such that*

$$\|f - p_n^*\|_\infty = \frac{\left|f^{(n+1)}(\xi)\right|}{2^n\,(n+1)!}. \tag{1.3}$$

This bound is not sufficient in general, since *near-best* uniform polynomial approximations also satisfy such a bound (see Equation 1.3, Lemma 4.3.4). Hence, having only this bound, even if we could find a way of having the "true" coefficients of p_n^*, it would not worth paying the effort of computing them, since easier ways exists for computing *near-best* approximations.

Certifying best. However, in some applications, one would like to take advantage of the improvement in the approximation error provided by the *best* approximation. Such an application led us to conceive an algorithm that automatically provides a rigorous and as accurate as desired enclosure of $\left\|\varepsilon_{abs/rel}\right\|_\infty$, where $\varepsilon_{abs/rel}$ is the absolute or relative approximation error between

a "sufficiently smooth" function f and a very good polynomial approximation of f. For example, such an approximation can be the polynomial obtained by truncating the coefficients of the real best approximation p_n^* to a machine-representable format. This work is discussed in detail in Chapter 3.

Using worse to certify best. One basic idea behind this work is to make use of a higher degree *worse* quality polynomial approximation, say T, which has the advantage of being easier to compute and certify. Then, with the help of the triangular inequality

$$||f - p_n^*|| \leqslant ||f - T|| + ||T - p_n^*||,$$

we can obtain a bound for $||f - p_n^*||$ as the sum of the bounds $||f - T||$ and $||T - p_n^*||$. The second norm is the norm of a polynomial $T - p^*$ which is in general easy to compute.

So, we were led to consider *easily certifiable worse* quality polynomial approximations. An already existing well-known rigorous computing tool based on Taylor approximations is: *Taylor models*. This tool was made popular by Makino and Berz and the ideas behind it are explained in a series of articles of several authors [97, 124, 98]. However, we encountered several difficulties in using it in an "off-the-shelf" manner. Among these, the implementations are scarce or not freely available, no multiple precision support for computations was available, we could not deal with a class of functions sufficiently large for our purposes, etc. More detailed arguments are given in Section 2.3. Hence, we proceeded to implementing Taylor Models-like algorithms in our software tool Sollya. Our implementation deals only with univariate functions, but several improvements or remarks were integrated, several data structure choices are different. We chose to explain in detail in Chapter 2 the algorithms for Taylor Models, including the specificities of our implementation. To our knowledge, there is no such detailed account in the literature regarding detailed algorithms for Taylor Models. This will also allow us to explain the "philosophy" of Taylor models that will be used in subsequent chapters. One other potential use of these algorithms is their on-going formalization in a formal proof checker. We use then, this tool to deal with the practical application of certifying the minimax error presented in Chapter 3.

Between worse and best there lay the near-best. But other well-known polynomial approximations of better quality than Taylor exist. So how can we take advantage of them in a rigorous computing tool instead of Taylor approximations? This is the question we answer in detail in Chapter 4. Here we mention just that we use two approximations proved to be *near-best* in the sense of the uniform norm:

- Truncated Chebyshev series (TCS), i.e., the polynomial obtained by truncating the Chebyshev series of a function f.
- Chebyshev interpolants (CI), i.e., polynomials which interpolate the function at special points called Chebyshev nodes. They are also called "approximate truncated Chebyshev series" in literature.

For example, Powell [132] showed that a Chebyshev interpolation polynomial will be almost as effective as the best approximation polynomial with the same degree. In fact, he showed that the ratio between the Chebyshev interpolation error and the best approximation error is bounded by:

$$1 < \frac{\varepsilon_{CI}}{\varepsilon_{best}} \leq 1 + \frac{1}{n+1} \sum_{i=0}^{n} \tan\left(\frac{(i+1/2)\pi)}{2(n+1)}\right) \leqslant 2 + \frac{2}{\pi}\log(n+1).$$

This implies that for example, if a polynomial approximation of degree 1000 is used, it may be guaranteed that the resulting Chebyshev interpolation approximation error does not exceed the

minimax error by more than at most a factor of six. A similar result [102, Chap. 5.5] exists for truncated Chebyshev series. Some of their advantages are that compared to Taylor approximations, these approximations have better convergence domain (see Section 4.2.3) and also, we should obtain bounds for the approximations that are scaled down by a factor of 2^{n-1} for a polynomial of degree n considered over $[-1, 1]$ (cf. (4.47) for example). CI and TCS approximations are tightly connected and we developed a rigorous computing tool analogous to Taylor Models based on both of them. We will see in Chapter 4 that the challenge of so-called "Chebyshev models" is to match the Taylor Model complexity of operations such addition, multiplication, composition, while offering a rigorous error bound close to optimality.

While the first 4 chapters deal with RPAs for functions defined explicitly by closed formulas, in Chapter 5 we will show how we can rigorously compute near-best uniform polynomial approximations for solutions of linear differential equations with polynomial coefficients (also known as D-finite functions). In this case, one key ingredient of our method is that coefficients of the Chebyshev series expansions of the solutions obey linear recurrence relations with polynomial coefficients. However, these do not lend themselves to a direct recursive computation of the coefficients, owing chiefly to the lack of initial conditions. Hence we will use a combination of a classical numerical method going back to Clenshaw, revisited in the light of properties of the recurrence relations we consider, and rigorous enclosure method for ordinary differential equation.

Finally, in Chapter 6 we present a practical application of RPAs to the synthesis of elementary functions in hardware. The main motivation of this work is to facilitate the implementation of a full hardware mathematical library (libm) in FloPoCo *, a core generator for high-performance computing on Field Programmable Gate Arrays (FPGAs). We present an architecture generator that inputs the specification of a function and outputs a synthesizable description of an architecture evaluating this function with guaranteed accuracy. We give a complete implementation in the context of modern FPGAs features, however, most of the methodology is independent of the FPGA target and could apply to other hardware targets such as ASIC circuits.

But, we can not start presenting all these works without giving a brief overview on computer arithmetic, just to remember in a glimpse on how modern computers compute today, whether we can trust them, or we should agree with A. Householder on the fact that "It makes me nervous to fly on airplanes, since I know they are designed using floating-point arithmetic."

1.2 Computer arithmetic

We need to represent and manipulate real numbers in any computer. Since the early days of electronic computing, many ways of approximating the infinite set of real numbers with a finite set of "machine numbers" have been developed. Several examples are: floating-point arithmetic, fixed-point arithmetic, logarithmic number systems, continued fractions, rational numbers, 2-adic numbers, etc. For this task, many constraints like speed, accuracy, dynamic range, ease of use and implementation, memory cost, or power consumption have to be taken into account. A good compromise among these factors is achieved by floating-point arithmetic which is nowadays the most widely used way of representing real numbers in computers. We give in what follows a brief overview of floating-point arithmetic, with the main purpose of setting up some notations and key concepts for the following chapters and refer the interested reader to [118], from which this small recap is heavily inspired.

Definition 1.2.1 (Floating-point number)**.** *A floating-point number in radix-β ($\beta \in \mathbb{N}$, $\beta \geq 2$),*

*. www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

Name	binary16	binary32 (single precision)	binary64 (double precision)	binary128 (quad precision)
p	11	24	53	113
$e_{\max}$	+15	+127	+1023	+16383
$e_{\min}$	−14	−126	−1022	−16382

Table 1.1: Main parameters of the binary interchange formats of size up to 128 bits specified by the 754-2008 standard [81].

precision-p ($p \in \mathbb{N}$, $p \geq 2$), is a number of the form

$$x = (-1)^s \cdot m \cdot \beta^e,$$

where:

- *e, called the* exponent, *is an integer such that $e_{min} \leq e \leq e_{max}$, where the extremal exponents $e_{min} < 0 < e_{max}$ are given;*
- *$m = |M| \cdot \beta^{1-p}$, called* normal significand *(or sometimes* mantissa*). M is an integer, called the* integral significand, *represented in radix β, $|M| \leqslant \beta^p - 1$. We note that m has one digit before the radix point, and at most $p-1$ digits after (notice that $0 \leq m < \beta$); and*
- *$s \in \{0, 1\}$ is the sign bit of x.*

In order to have a unique representation of a floating-point number, we *normalize* the finite nonzero floating-point numbers by choosing the representation for which the exponent is minimum (yet larger than or equal to e_{min}). This gives two kinds of floating-point numbers:

- *normal* numbers: $1 \leq |m| < \beta$, or, equivalently, $\beta^{p-1} \leq |M| < \beta^p$.
- *subnormal* numbers: $e = e_{min}$ and $|m| < 1$ or, equivalently, $|M| \leq \beta^{p-1} - 1$. Zero is a special case (see Chapter 3 of [118] for more details).

In radix 2, the first digit of the significand of a normal number is a 1, and the first digit of the significand of a subnormal number is a 0. The availability of subnormal numbers allows for *gradual underflow*. This significantly eases in general the writing of stable numerical software (see Chapter 2 of [118] and references therein).

Since most computers are based on two-state logic, radix 2 (and, more generally, radices that are a power of 2) are most common. However, radix 10 is also used, since it is what most humans use, and what has been extensively used in financial calculations and in pocket calculators. The computer algebra system Maple, also uses radix 10 for its internal representation of numbers.

In 1985, the IEEE754-1985 Standard for Binary Floating-Point Arithmetic was released [3]. This standard specifies various formats, the behavior of the basic operations ($+, -, \times, \div$ and $\sqrt{}$), conversions, and exceptional conditions. Nowadays, most systems of commercial significance offer compatibility with IEEE 754-1985. This has resulted in significant improvements in terms of accuracy, reliability, and portability of numerical software. A revision of this standard, called IEEE 754-2008 (which includes decimal floating-point arithmetic also), was adopted in June 2008 [81]. For example, the main parameters of the binary formats of size up to 128 bits defined by the new revision of the standard are given in Table 1.1.

Definition 1.2.2. *We define the set of binary floating-point numbers of precision p: $\mathcal{F}_p = \left\{2^E \cdot m | E \in \mathbb{Z}, m \in \mathbb{Z}, 2^{p-1} \leq |m| \leq 2^p - 1\right\} \cup \{0\}$.*

For example, with the required bounds on E, $\mathcal{F}_{24}$ is the single precision (binary32) format.

One of the most interesting ideas brought out by IEEE 754-1985 is the concept of *rounding mode*. In general, the result of an operation (or function) on floating-point numbers is not exactly

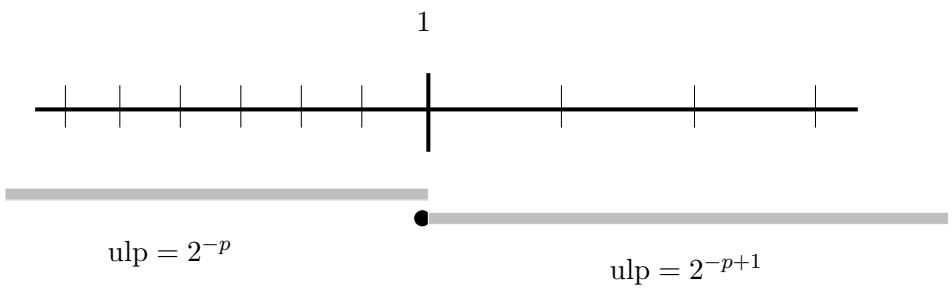


Figure 1.5: *Values of* $\mathrm{ulp}(x)$ *around* 1*, assuming radix* 2 *and precision* p*.*

representable in the floating-point system being used, so it has to be *rounded*. The four rounding modes that appear in the IEEE 754-2008 standard are:

- round toward $-\infty$: $\mathrm{RD}(x)$ is the largest value that is either a floating-point number or $-\infty$ less than or equal to x;
- round toward $+\infty$: $\mathrm{RU}(x)$ is the smallest value that is either a floating-point number or $+\infty$ greater than or equal to x;
- round toward zero: $\mathrm{RZ}(x)$ is the closest floating-point number to x that is no greater in magnitude than x (it is equal to $\mathrm{RD}(x)$ if $x \geq 0$, and to $\mathrm{RU}(x)$ if $x \leq 0$);
- round to nearest: $\mathrm{RN}(x)$ is the floating-point number that is the closest to x. A tie-breaking rule must be chosen when x falls exactly halfway between two consecutive floating-point numbers. A frequently chosen tie-breaking rule is called *round to nearest even*: x is rounded to the only one of these two consecutive floating-point numbers whose integral significand is even. This is the default mode in the IEEE 754-2008 Standard.

One convenient way to measure rounding errors is to express them in terms of what we would intuitively define as the "weight of the last bit of the significand" or *unit in the last place* (ulp). We refer the reader to [118, Chapter 2.6.] for a detailed account. Roughly speaking, the $\mathrm{ulp}(x)$ is the distance between two consecutive FP numbers around x. Since this is ambiguous at the neighborhood of powers of radix 2, there are several slightly different definitions due for example to W. Kahan, or J. Harrison or D. Goldberg. Here we use the last one. Accordingly, Figure 1.5 shows the values of ulp near 1.

Definition 1.2.3 (Goldberg's ulp)**.** *If* $x \in [2^e, 2^{e+1})$ *then* $\mathrm{ulp}(x) = 2^{e-p+1}$.

When the exact result of a function is rounded according to a given rounding mode (as if the result were computed with infinite precision and unlimited range, then rounded), one says that the function is *correctly rounded*. According to the IEEE754-1985 Standard, the basic operations ($+, -, \times, \div$ and $\sqrt{}$) have to produce correctly rounded results.

However, most of real applications use also functions such as $\exp$, $\log$, $\sin$, $\arccos$, or some compositions of them. These functions are usually implemented in mathematical libraries called `libm`. Such libraries are available on most systems and many numerical programs depend on them. But until recently, there was no such requirement for these functions. The main impediment for this was the Table Maker's Dilemma (TMD) [118, Chapter 12], named in reference to the early builders of logarithm tables. This problem can be stated as follows: consider a function f and a floating-point number x. Since floating-point numbers are rational numbers, in many cases, the image $y = f(x)$ is not a rational number, and therefore, cannot be represented exactly as a floating-point number. The correctly rounded result will be the floating-point number that

is closest to this mathematical value. Using a finite precision environment (on a computer), only an approximation $\hat{y}$ to the real number y can be computed. If the accuracy used for computation is not enough, it is impossible to decide the correct rounding of $\hat{y}$. A technique published by Ziv [174, 118] is to improve the accuracy of the approximation until the correctly rounded value can be decided. A first practical improvement over Ziv's approach derives from the availability of tight bounds on the worst case accuracy required to compute many elementary functions, computed by Lefevre and Muller [94, 118] using ad-hoc algorithms. For some functions, these worst cases are completely covered ($\exp, \log_2, \log$, hyperbolic sine, cosine and their inverses, for the binary32 and binary64 formats). If the worst case accuracy required to round correctly a function is known [94], then only two steps are needed. This makes it easier to optimize and prove each step. This improvement allowed for the possibility of writing a libm where the functions are correctly rounded and this is obtained at known and modest additional costs. This is one of the main purposes of the Arenaire team that develops the CRlibm project [135].

In the new revision IEEE754-2008, correct rounding for some function* like: $\exp$, $\ln$, $\log_2$, $\sin$, $\cos$, $\tan$, $\arctan$ is recommended. The participation of leading microprocessor manufacturers like Intel or AMD for this standard revision proves that the purpose of CRlibm was achieved: the requirement of correct rounding for elementary functions is compatible with industrial requirements and can be done for a modest additional cost compared to a classical libm.

However, beside the TMD, the development and implementation of correctly rounded elementary functions is a complex process. A general scheme for this would include:

Step 1. Use the above mentioned methods of Muller and Lefevre [94] to obtain the necessary precision p in the worst cases. This subject is still an active research field, inside the TaMaDi [116] project, since the methods mentioned do not scale for the format binary128, for example, specified in the IEEE-754-2008 standard.

Step 2. Argument reduction for the function f to be considered: this involves the reduction of the problem to evaluating a function g over a tight interval $[a, b]$. For this, different ad-hoc methods are used on a case by case basis for each function.

Step 3. Find a polynomial approximation p_1 for g such that the maximum relative error between p_1 and g is small enough to allow for correct rounding in the general case. Find a polynomial approximation p_2 for g such that the maximum relative error between p_2 and g is small enough (less then 2^{-p}) to allow for correct rounding in the worst case. Practical examples for the implementation of binary64 correctly rounded standard functions in the CRlibm are: in the general case, we need $\|(g - p_1)/g\|_\infty < 2^{-65}$, this value insures that in practice for 99.9% of values f can be correctly rounded using the evaluation of p_1; in the worst cases, we need between 120 and 160 correct bits: $\|(g - p_2)/g\|_\infty < 2^{-p}$.

Polynomial approximation is preferred since polynomials can be evaluated completely based only on multiplications and additions, operations that are commonly available and highly optimized in current hardware. An important remark is that the coefficients of the approximation polynomials have to be exactly representable in an available machine format. There are several works [28, 26, 35] done within the Arenaire Team regarding ways of obtaining very good or best polynomial approximations with *machine efficient* coefficients.

Step 4. Once the *machine-efficient* polynomial approximations have been numerically found, one has to certify the maximum approximation error commited, i.e. to find a safe upper bound for $\|(g - p_1)/g\|_\infty$ and $\|(g - p_2)/g\|_\infty$. We note that in order to ensure the validity of the use of p_1 and p_2 instead of g, this bound has to be rigorous. Although numerical algorithms for supremum norms are efficient, they cannot offer the required safety. This was one of the

*. The exhaustive list can be found in Chap. 12.1 of [118]

initial motivations of this work and we will give a practical application of RPAs concerning this step in Chapter 3.

Step 5. Write the code for evaluating p_1 and p_2 with the required accuracy. In this step round-off errors have to be taken into account for each multiplication and addition such that the total error stays below the required threshold. In Chapter 6 we will see an example of how this step is implemented efficiently for a specific target architecture.

But, sometimes, even with a correctly implemented floating-point arithmetic, the result of a computation is far from what could be expected. We take here an example designed by Siegfried Rump in 1988 [147].

Example 1.2.4 (Rump's example)**.**

$$f(a,b) = 333.75b^6 + a^2\left(11a^2b^2 - b^6 - 121b^4 - 2\right) + 5.5b^8 + \frac{a}{2b},$$

and we want to compute $f(a,b)$ for $a = 77617.0$ and $b = 33096.0$. The results obtained by Rump on an IBM 370 computer were:

- 1.172603 *in single precision;*
- 1.1726039400531 *in double precision; and*
- 1.172603940053178 *in extended precision.*

From these computations we get the impression that the single precision result is certainly very accurate. And yet, the exact result is $-0.8273960599\cdots$. On more recent systems, we do not see the same behavior exactly. For instance, according to [118], on a Pentium4-based workstation, using GCC and the Linux system, the C program (Program 1.1) which uses double-precision computations, will return $5.960604\cdot 10^{20}$, whereas its single-precision equivalent will return $2.0317\cdot 10^{29}$ and its double-extended precision equivalent will return $-9.38724\cdot 10^{-323}$.

```
#include <stdio.h>
int main(void)
{
  double a = 77617.0;
  double b = 33096.0;
  double b2,b4,b6,b8,a2,firstexpr,f;
  b2 = b*b;
  b4 = b2*b2;
  b6 = b4*b2;
  b8 = b4*b4;
  a2 = a*a;
  firstexpr = 11*a2*b2-b6-121*b4-2;
  f = 333.75*b6 + a2 * firstexpr + 5.5*b8 + (a/(2.0*b));
  printf("Double precision result: $ %1.17e \n",f);
}
```

Program 1.1: Rump's example.

What happens if we increase the precision used? We continue this example, and implement it in C (see Program 1.2), using a *multiple precision* package: MPFR [63]. We mention here that MPFR implements arbitrary precision floating-point arithmetic compliant with the IEEE-754-2008 standard and we discuss more about this library in Section 1.3. The results obtained on an Intel Core2Duo-based workstation, using GCC and a Linux system, with MPFR version 3.0.1 for precision varying from 23 to 122 bits are given in Table 1.2.

Precision	Result
23	1.171875
24	$-6.338253001141147007483516027e29$
53	$-1.180591620717411303424000000e21$
54	1.172603940053178583902138143
55	1.172603940053178639413289374
56	1.172603940053178639413289374
57	1.172603940053178625535501566
58	$3.689348814741910323200000000e19$
59	$-1.844674407370955161600000000e19$
60	$-1.844674407370955161600000000e19$
62	1.172603940053178632040714601
63	$5.764607523034234891875000000e17$
65	1.172603940053178631878084275
70	1.172603940053178631859449551
80	1.172603940053178631858834128
85	$2.748779069451726039400531789e11$
90	$1.717986918517260394005317863e10$
100	1.172603940053178631858834904
110	1.172603940053178631858834904
120	1.172603940053178631858834904
122	$-8.273960599468213681411650955e-1$

Table 1.2: Results obtained executing Program 1.2 implementing Example 1.2.4, with precision varying from 23 to 122 bits.

We clearly see that gradually increasing the precision until the result seems to be stable is not a secure approach. What is the "safe" precision to use such that no flagrant computation error occurs? Here it seems to be 122. But how can we be sure in general that we have the correct mathematical result, or at least some of the correct digits of the result? How can we determine the accuracy of this computation ?

```
#include <stdio.h>
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>
int main (int argc, char*argv[]){
  unsigned int prec;
  mpfr_t f, a,b,sqra,sqrb,b4,b8,b6,t;
  prec=atoi(argv[1]);
  mpfr_init2 (f, prec);
  mpfr_set_d(f,333.75,GMP_RNDN);
  mpfr_init2 (b, prec);
  mpfr_set_str(b,"33096",10,GMP_RNDN);
  mpfr_init2 (b6, prec);
  mpfr_pow_ui(b6,b,6,GMP_RNDN);
  mpfr_mul(f,f,b6,GMP_RNDN);
  mpfr_init2 (sqrb, prec);
  mpfr_pow_ui(sqrb,b,2,GMP_RNDN);
  mpfr_init2 (a, prec);
  mpfr_set_str(a,"77617",10,GMP_RNDN);
  mpfr_init2 (sqra, prec);
  mpfr_pow_ui(sqra,a,2,GMP_RNDN);
  mpfr_init2 (t, prec);
  mpfr_mul(t,sqra,sqrb,GMP_RNDN);
  mpfr_mul_ui(t,t,11,GMP_RNDN);
  mpfr_sub(t,t,b6,GMP_RNDN);
  mpfr_init2(b4,prec);
  mpfr_pow_ui(b4,b,4,GMP_RNDN);
  mpfr_mul_ui(b4,b4,121,GMP_RNDN);
  mpfr_sub(t,t,b4,GMP_RNDN);
  mpfr_sub_ui(t,t,2,GMP_RNDN);
  mpfr_mul(t,sqra,t,GMP_RNDN);
  mpfr_add(f,f,t,GMP_RNDN);
  mpfr_init2(b8,prec);
  mpfr_pow_ui(b8,b,8,GMP_RNDN);
  mpfr_mul_ui(b8,b8,55,GMP_RNDN);
  mpfr_div_ui(b8,b8,10,GMP_RNDN);
  mpfr_add(f,f,b8,GMP_RNDN);
  mpfr_div(t,a,b,GMP_RNDN);
  mpfr_div_ui(t,t,2,GMP_RNDN);
  mpfr_add(f,f,t,GMP_RNDN);
  printf ("Result is ");
  mpfr_out_str (stdout, 10, 17, f, GMP_RNDD);
  return 0;
}
```

Program 1.2: Rump's example - C implementation using MPFR.

We present in the following section the state-of-the-art tool that "never lies", the basic brick in most rigorous computations: Interval Arithmetic.

1.3 Interval Arithmetic

In this chapter we briefly describe the fundamentals of interval arithmetic, following mainly [115, 162, 74]. Interval arithmetic is of use when dealing with inequalities, approximate numbers or error bounds in computations. We use an interval $\boldsymbol{x}$ as the formalization of the intuitive notion of an unknown number x known to lie in $\boldsymbol{x}$. In interval analysis we do not say that the value of a variable is a certain number, but we say that a value of a variable is in an interval of possible values. For example, when dealing with numbers that cannot be represented exactly like π or $\sqrt{2}$, we usually say that π is approximately equal to 3.14. Instead, in interval analysis we say that π is exactly in the interval $[3.14, 3.15]$. For an operation where we have errors in the inputs we can give an approximate result:

$$-\pi \cdot \sqrt{2} \approx -3.14 \cdot 1.41 = -4.4274.$$

On the other hand, when we do an operation in interval arithmetic, we no longer use approximations, but enclosures. For example for the above multiplication, whenever we have 2 values in the input intervals then their product is a value in the result interval:

$$[-3.15, -3.14] \cdot [1.41, 1.42] = [-4.473, -4.4274].$$

So regardless of the imprecision in the input data, we can always be sure that the result will be inside the computed bounds. With interval analysis we cannot be wrong because of rounding errors or method errors, we can only be imprecise by giving a very wide interval enclosure for the expected value.

Brief and subjective history. The birthdate of interval arithmetic is not certain: it it widely accepted that the *father* of interval arithmetic is Ramon Moore who mentioned it for the first time in 1962 [112] and completely defined it in 1966 [113], but earlier references were found, for example in 1931 [171], or 1958 [154]. The site `http://www.cs.utep.edu/interval-comp/`, section *Early papers* gives a complete list of references.

In the '80s, there was a major development of interval arithmetic, especially in Germany, under the impulse of U. Kulisch in Karlsruhe. A specific processor, followed by an instruction set and a compiler were developed by IBM [80] after his suggestions. In the same time, the floating-point arithmetic underwent a major turning point, with the adoption of the IEEE 754 Standard, who specified in particular the rounding modes and facilitated this way the implementation of interval arithmetic. On the other hand, interval arithmetic failed to convince many users - it was probably because there was an *over-rated* tendency to pretend that by replacing all floating-point computations with interval computations, one obtains a result that would give a tight enclosure of the rounding errors. This is hardly the case, as we will see throughout this and the following chapters. This *failure* was detrimental for the beginnings of interval arithmetic.

However, interval arithmetic continues to develop with several different objectives, and it seems to be an indispensable tool for rigorous global optimization [137, 97, 14, 36, 15], for rigorous ODE solving [44, 161, 123, 100], for rigorous quadrature [13].

We note that standardization of interval arithmetic is currently undertaken by the IEEE-1788 working group [140].

Notations

Definition 1.3.1 (Real Interval). *Let $\underline{x}, \overline{x} \in \mathbb{R}$, $\underline{x} \leq \overline{x}$. We define the interval $\boldsymbol{x}$ (denoted in bold letters) by $\boldsymbol{x} = [\underline{x}, \overline{x}] := \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$. We call $\underline{x}$ the minimum of $\boldsymbol{x}$ and $\overline{x}$ its maximum. We denote the set of all real intervals by $\mathbb{IR}$.*

Remark 1.3.2. *Intervals are closed, bounded, connected and nonempty subsets of* $\mathbb{R}$.

Definition 1.3.3 (Interval Width, Center, Radius)**.** *Let* $\boldsymbol{x} \in \mathbb{IR}$. *We denote the width of* $\boldsymbol{x}$ *by* $w(\boldsymbol{x}) = \overline{x} - \underline{x}$. *The center* $mid(\boldsymbol{x})$ *and the radius* $rad(\boldsymbol{x})$ *are defined by* $mid(\boldsymbol{x}) = (\underline{x} + \overline{x})/2$ *and* $rad(\boldsymbol{x}) = (\overline{x} - \underline{x})/2 = w(\boldsymbol{x})/2$.

Remark 1.3.4. *We note that there exist alternative characterizations of intervals which are based roughly on* $(mid(\boldsymbol{x}), rad(\boldsymbol{x}))$ *pair, for example in [120, 164], but in this work we do not detail further this representation.*

Definition 1.3.5 (Degenerate Interval)**.** *Let* $x \in \mathbb{R}$ *be a real number. A* point (degenerate) interval $[x] = [x, x]$ *is a usual numeric object that confounds with the interval of zero width and contains only the value* x.

Remark 1.3.6 (Machine representable point intervals)**.** *When* x *can not be exactly represented in the underlying machine format, we will still denote by* $[x]$ *the tightest machine representable interval containing* x. *In order to remove any ambiguity, in what follows the convention would be the following: when we are sure that* x *is exactly representable, we denote by* $[x, x]$ *the point interval, e.g.* $[0, 0]$, $[1, 1]$, *when there is no such assumption we denote simply by* $[x]$, *e.g.* $[\pi] = [\mathrm{RD}(\pi), \mathrm{RU}(\pi)]$.

Operations. We define operations on intervals. The key point in these definitions is that *computing with intervals is computing with sets*. For example, when we add two intervals, the resulting interval is a set containing the sums of all pairs of numbers, one from each of the two initial sets. So, addition will be described as follows (for simplicity, we use the same symbol $+$ for both addition of intervals and of real numbers):

$$\boldsymbol{x} + \boldsymbol{y} := \{x + y \mid x \in \boldsymbol{x}, y \in \boldsymbol{y}\}.$$

We can summarize a similar definition for all the four common operations.

Definition 1.3.7 (Arithmetic operations)**.** *Let* $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR}$. *Let* $\odot \in \{+, -, \cdot, /\}$. *We denote by:*

$$\boldsymbol{x} \odot \boldsymbol{y} := \{x \odot y \mid x \in \boldsymbol{x}, y \in \boldsymbol{y}\}.$$

with the mention that for division, $0 \notin \boldsymbol{y}$.

Proposition 1.3.8. *Let* $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{IR}$, $\boldsymbol{x} = [\underline{x}, \overline{x}]$, $\boldsymbol{y} = [\underline{y}, \overline{y}]$. *We can compute* $\boldsymbol{x} \odot \boldsymbol{y}$, *for* $\odot \in \{+, -, \cdot, /\}$ *in the following way:*

$$\begin{aligned}
[\underline{x}, \overline{x}] + [\underline{y}, \overline{y}] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
[\underline{x}, \overline{x}] - [\underline{y}, \overline{y}] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
[\underline{x}, \overline{x}] \cdot [\underline{y}, \overline{y}] &= [\min(\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y}), \max(\underline{x} \cdot \underline{y}, \underline{x} \cdot \overline{y}, \overline{x} \cdot \underline{y}, \overline{x} \cdot \overline{y})] \\
1/[\underline{y}, \overline{y}] &= [\min(1/\underline{y}, 1/\overline{y}), \max(1/\underline{y}, 1/\overline{y})] \text{ if } 0 \notin [\underline{y}, \overline{y}] \\
[\underline{x}, \overline{x}] / [\underline{y}, \overline{y}] &= [\underline{x}, \overline{x}] \cdot (1/[\underline{y}, \overline{y}]) \text{ if } 0 \notin [\underline{y}, \overline{y}]
\end{aligned}$$

Proof. We obtain these formulas using the monotonicity of these operations. □

Example 1.3.9. *Let* $\boldsymbol{x} = [1, 2]$ *and* $\boldsymbol{y} = [-1, 1]$.
Then $\boldsymbol{x} + \boldsymbol{y} = [0, 3]$; $\boldsymbol{y} - \boldsymbol{x} = [-3, 0]$; $\boldsymbol{x} \cdot \boldsymbol{y} = [-2, 2]$; $1/\boldsymbol{x} = [0.5, 1]$.

We have seen above that division by an interval containing zero is not defined under the basic interval arithmetic. However, one can define the extended division.

Remark 1.3.10 (Extended division)**.** *For division by an interval* $[\underline{y}, \overline{y}]$ *including zero,* $\underline{y} < 0 < \overline{y}$, *one defines* $1/[\underline{y}, 0] = [-\infty, 1/\underline{y}]$ *and* $1/[0, \overline{y}] = [1/\overline{y}, \infty]$. *Then the result of the division is the union of two intervals:* $1/[\underline{y}, \overline{y}] = [-\infty, 1/\underline{y}] \cup [1/\overline{y}, \infty]$.

Algebraic Properties. We can observe that the interval operations defined above do not have the properties of their real numbers analogues.

Proposition 1.3.11. *The following properties hold in interval arithmetic:*
(i) Subtraction is not the reciprocal of addition;
(ii) Division is not the reciprocal of multiplication;
(iii) Multiplication of an interval by itself is not equivalent to squaring the interval;
(iv) Multiplication is not distributive with addition;
(v) Multiplication is under-distributive with addition i.e. $\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{IR}$, $\boldsymbol{x} \cdot (\boldsymbol{y} + \boldsymbol{z}) \subseteq \boldsymbol{x} \cdot \boldsymbol{y} + \boldsymbol{x} \cdot \boldsymbol{z}$;
(vi) Let $\boldsymbol{x} \in \mathbb{IR}$. *Then* $[0] + \boldsymbol{x} = \boldsymbol{x}$ *and* $[0] \cdot \boldsymbol{x} = [0]$.

Proof. For (i)-(iv) it suffices to give a counter-example:
(i) Let $\boldsymbol{x} = [2, 3]$, $\boldsymbol{x} - \boldsymbol{x} = [2, 3] - [2, 3] = [-1, 1] \neq \{0\}$ even if, it obviously contains it. Moreover,

$$\boldsymbol{x} - \boldsymbol{x} = \{x - y \mid x \in \boldsymbol{x},\ y \in \boldsymbol{x}\} \supseteq \{x - x \mid x \in \boldsymbol{x}\} = \{0\}$$

and the inclusion is strict.
(ii) Let $\boldsymbol{x} = [2, 3]$, the interval $\boldsymbol{x}/\boldsymbol{x} = [2, 3]/[2, 3] = [2/3, 3/2] \neq 1$.
(iii) Let $\boldsymbol{x} = [-3, 2]$,

$$\boldsymbol{x} \cdot \boldsymbol{x} = [-3, 2] \cdot [-3, 2] = [-6, 9]$$

while

$$\boldsymbol{x}^2 = \{x^2 \mid x \in \boldsymbol{x}\} = [0, 9]\,.$$

(iv) Let $\boldsymbol{x} = [-2, 3]$, $\boldsymbol{y} = [1, 4]$ and $\boldsymbol{z} = [-2, 1]$,

$$\begin{aligned}
\boldsymbol{x} \cdot (\boldsymbol{y} + \boldsymbol{z}) &= [-2, 3] \cdot ([1, 4] + [-2, 1]) \\
&= [-2, 3] \cdot [-1, 5] \\
&= [-10, 15] \\
\boldsymbol{x} \cdot \boldsymbol{y} + \boldsymbol{x} \cdot \boldsymbol{z} &= [-2, 3] \cdot [1, 4] + [-2, 3] \cdot [-2, 1] \\
&= [-8, 12] + [-6, 4] \\
&= [-14, 16]
\end{aligned}$$

(v) We prove that

$$\boldsymbol{x} \cdot (\boldsymbol{y} + \boldsymbol{z}) \subseteq \boldsymbol{x} \cdot \boldsymbol{y} + \boldsymbol{x} \cdot \boldsymbol{z}.$$

We have $\{x \cdot (y + z) \,|\, x \in \boldsymbol{x},\ y \in \boldsymbol{y},\ z \in \boldsymbol{z}\} = \{x \cdot y + x \cdot z \,|\, x \in \boldsymbol{x},\ y \in \boldsymbol{y},\ z \in \boldsymbol{z}\} \subseteq \{x \cdot y + x' \cdot z \,|\, x \in \boldsymbol{x},\ x' \in \boldsymbol{x},\ y \in \boldsymbol{y},\ z \in \boldsymbol{z}\}$.

□

Interval extension of functions. We could in fact go further and define functions of interval variables by treating these, in a similar fashion, as "unary operations".

Definition 1.3.12 (Function Range (Image)). *Let* $\boldsymbol{x} \in \mathbb{IR}$ *and* $f : \boldsymbol{x} \to \mathbb{R}$. *We denote the image (range) of* f *over* $\boldsymbol{x}$ *by:* $f(\boldsymbol{x}) = \{f(x) \mid x \in \boldsymbol{x}\}$.

Finding the image of a function (usually multivariate) and in particular a value for which a function attains its minimum (maximum) forms is in itself a wide field of study in mathematics and computer science, called "Optimization Theory". Except for trivial cases, we do not have standard easy ways to describe the exact image of such functions over a specific domain. We will see, that in some cases, interval arithmetic is helpful in this matter.

For example, we give below simple formulas for computing the image of functions such as exponential and square root or squaring.

Example 1.3.13. *Let $\boldsymbol{x} \in \mathbb{IR}$, $\boldsymbol{x} = [\underline{x}, \overline{x}]$. Using monotonicity properties, we can compute:*

$$\begin{array}{lcl} [\underline{x}, \overline{x}]^2 & = & \textit{if } 0 \notin [\underline{x}, \overline{x}] \textit{ then } \left[\min(\underline{x}^2, \overline{x}^2), \max(\underline{x}^2, \overline{x}^2)\right], \textit{ else } \left[0, \max(\underline{x}^2, \overline{x}^2)\right]; \\ \sqrt{[\underline{x}, \overline{x}]} & = & \textit{if } 0 \leq \underline{x} \textit{ then } \left[\sqrt{\underline{x}}, \sqrt{\overline{x}}\right] \textit{ else FAIL;} \\ \exp([\underline{x}, \overline{x}]) & = & [\exp \underline{x}, \exp \overline{x}]. \end{array}$$

For functions (ex. $\exp$) which are monotonic, we can easily deduce simple formulas which allow us to compute their range over a given interval. For periodic functions (ex. $\sin, \tan, \cos$), one has to be more careful, but it is possible to establish algorithms for computing their exact range as soon as we can compute these functions over reals.

In this way, we can extend functions to *interval functions*.

Definition 1.3.14 (Interval extension)**.** *Let $\boldsymbol{x} \in \mathbb{IR}$ and $f : \boldsymbol{x} \to \mathbb{R}$, then $\boldsymbol{f} : \mathbb{IR} \to \mathbb{IR}$ is called an interval extension of f over $\boldsymbol{x}$ if for all $\boldsymbol{y} \subseteq \boldsymbol{x}$, $\boldsymbol{f}(\boldsymbol{y}) \supseteq f(\boldsymbol{y})$.*

This definition implies that several interval extensions are possible for the same function. For example, for $f = \exp$, both $\boldsymbol{f}([\underline{x}, \overline{x}]) = [\exp \underline{x}, \exp \overline{x}]$ and $\boldsymbol{f}([\underline{x}, \overline{x}]) = [-\infty, \infty]$ are allowable interval extensions.

We are of course interested in obtaining tightest possible results. For future reference, we define the set of *basic* (standard) functions for which we can compute using simple formulas their exact range over a given interval. We usually say that these functions have a *sharp interval extension*.

Definition 1.3.15 (Basic (standard) functions)**.** *We denote the set of* basic *(standard) functions $\mathfrak{S} = \{\sin, \exp, \tan, \log, \sqrt{\ldots}\}$ for which we can compute using such simple formulas their exact range over a given interval.*

This class of functions is too small for most practical applications, so we will use it as *basic bricks* for more complicated functions in what follows.

Definition 1.3.16 (Elementary function)**.** *Any real-valued function expressed only by a finite number of arithmetic operations and compositions with basic functions and constants is called an elementary function (see also [29, Definition 5.1.4]).*

Given an explicit representation of an elementary function f, it follows that once we have already defined interval extensions for basic functions and arithmetic operations, we can define an interval extension of f by replacing all occurrences of x by the interval $\boldsymbol{x}$ and replacing (*overloading*) all operations with interval operations. This is usually called the *natural interval extension* of f. In general, we can not hope for a sharp extension if the variable x occurs more than once in the explicit representation of f.

It should also be pointed out that an elementary function has infinitely many interval extensions. If $\boldsymbol{F}$ is an extension of f, then so is $\boldsymbol{F}(\boldsymbol{x}) + \boldsymbol{x} - \boldsymbol{x}$.

Elementary functions do not, in general, admit sharp interval extensions. However, it is possible to prove that the extensions, when well-defined, are *inclusion isotonic*:

Definition 1.3.17. *Let $\boldsymbol{x} \in \mathbb{IR}$. An interval-valued function $F : \boldsymbol{x} \cap \mathbb{IR} \to \mathbb{IR}$ is inclusion isotonic if, for all $\boldsymbol{z} \subseteq \boldsymbol{z'} \subseteq \boldsymbol{x}$, we have $F(\boldsymbol{z}) \subseteq F(\boldsymbol{z'})$.*

Theorem 1.3.18 (Fundamental Theorem of Interval Analysis)**.** *Given an elementary function f, and a natural interval extension F such that $F(\boldsymbol{x})$ is well-defined for some $\boldsymbol{x} \in \mathbb{IR}$, we have:*

- *$\boldsymbol{z} \subseteq \boldsymbol{z'} \subseteq \boldsymbol{x} \implies F(\boldsymbol{z}) \subseteq F(\boldsymbol{z'})$, (inclusion isotonicity)*
- *$f(\boldsymbol{x}) \subseteq F(\boldsymbol{x})$. (range enclosure)*

Proof. Given in [162]. □

Also, with these interval extensions we obtain *valid interval bounds* for such functions.

Definition 1.3.19 (Valid interval bound)**.** *Let $\boldsymbol{x} \in \mathbb{IR}$ and $f : \boldsymbol{x} \to \mathbb{R}$. An interval $\boldsymbol{B(f)}$ is called a valid interval bound for f when*

$$\forall x \in \boldsymbol{x}, f(x) \in \boldsymbol{B(f)}.$$

Remark 1.3.20 (At least Natural Interval Extensions are available.)**.** *In the algorithms given in all subsequent chapters we suppose that at least a natural interval extension is available. More specifically, we suppose that we have an algorithm* `eval`$(f, \boldsymbol{x})$ *that for a function f and an interval $\boldsymbol{x}$ returns a valid interval bound for f over $\boldsymbol{x}$, i.e. an interval $\boldsymbol{y}$ such that $\forall \xi \in \boldsymbol{x}, f(\xi) \in \boldsymbol{y}$. For* basic *functions this algorithm returns a* sharp *interval enclosure. For elementary functions, we do not impose any constraints on the size of the enclosure obtained, besides the fact that it should be a valid enclosure (obtained for example by natural interval extension). We also assume that the algorithm is able to handle functions of several variables varying in a vector of intervals.*

Some examples are given in the following:

Example 1.3.21. *Let $\boldsymbol{x} = [-5, 2]$, $g(x) = 2x^2 + x - 3$. The* natural interval extension *of g is $G(\boldsymbol{x}) = 2\boldsymbol{x}^2 + \boldsymbol{x} - 3$. We can evaluate it using basic interval arithmetic operations.*

$$\begin{aligned} & 2\,[-5,2]^2 + [-5,2] - 3 \\ = \;& 2\,[0,25] + [-5,2] - 3 \\ = \;& [0,50] + [-5,2] - 3 \\ = \;& [-8,49]\,. \end{aligned}$$

We have $g(\boldsymbol{x}) \subseteq G(\boldsymbol{x})$. We note that the interval bound obtained is not the sharpest *possible. Actually, $g(\boldsymbol{x}) = [-3.125, 42]$.*

We also note that although the representation of g is immaterial when computing over $\mathbb{R}$, it does make a big difference in $\mathbb{IR}$.

Example 1.3.22. *We write $g(x) = x(2x + 1) - 3$. By evaluating in interval arithmetic over $\boldsymbol{x} = [-5, 2]$, we get: $[-5, 2] \cdot (2 \cdot [-5, 2] + [1, 1]) - [3, 3] = [-28, 42]$.*

Remark 1.3.23 (Overestimation)**.** *Previous examples show that the interval bounds obtained by replacing all operations with their interval analogue are not the sharpest possible. This phenomenon is called* overestimation *and implies an* impreciseness *in the given result, since sometimes a very wide interval enclosure is given for the expected results.*

As mentioned above, this *failure* was detrimental for the beginnings of interval arithmetic. In what follows we analyze in more detail some sources of overestimation in interval arithmetic, as presented in [124]:

Dependency phenomenon 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. One can identify two special cases of "dependence":

- Wrapping, relates to overestimation due to the depth of the computational graph, caused by long sequences of nested operations depending on a limited number of variables only, which also magnifies bounds on rounding errors and hence can give wide meaningless results even for problems with exact data.

– Cancellation relates to overestimation due to expressions containing at least one addition or subtraction where, in floating point arithmetic, the result has much smaller magnitude than the arguments; in interval arithmetic, the width is additive instead of cancelling, leading to large overestimation in such cases.

In some cases, by using more clever techniques, rewriting expressions or taking into account the properties of the functions considered, one can reduce this phenomenon.

We mention also the following theorem taken from [162] which asserts that using interval subdivision, overestimation in valid bounds computations using isotonic interval extensions can be made as small as desired using interval subdivisions. Let us recall that a function $f : \boldsymbol{x} \to \mathbb{R}$ is Lipschitz continuous if there exists $K \in \mathbb{R},\ K > 0$ such that for all $x, y \in \boldsymbol{x}$, we have $|f(x) - f(x)| \leqslant K|x - y|$.

Theorem 1.3.24. *Let $\boldsymbol{x} \in \mathbb{IR}$ and let $f : \boldsymbol{x} \to \mathbb{R}$ be an elementary function such that any subexpression of f is Lipschitz continuous over $\boldsymbol{x}$. Let an inclusion isotonic interval extension $\boldsymbol{F}$ of f such that $\boldsymbol{F}(\boldsymbol{x})$ is well-defined for $\boldsymbol{x}$. Then there exists $K \in \mathbb{R},\ K > 0$ depending on $\boldsymbol{F}$ and $\boldsymbol{x}$ such that, if $\boldsymbol{x} = \bigcup_{i=1}^{k} \boldsymbol{x}_i$, then:*

$$f(\boldsymbol{x}) \subseteq \bigcup_{i=1}^{k} \boldsymbol{F}(\boldsymbol{x}_i) \subseteq \boldsymbol{F}(\boldsymbol{x}),$$

and

$$rad\left(\bigcup_{i=1}^{k} \boldsymbol{F}(\boldsymbol{x}_i)\right) \leqslant rad\left(f(\boldsymbol{x})\right) + K \max_{i=1,\dots k} rad\left(\boldsymbol{x}_i\right).$$

Proof. The detailed proof is given in [162, Chapter 3]. The idea is that the first statement is obtained directly from the isotonic property of interval extension, while the second statement is proven inductively based on the Lipschitz property of any subexpression of f. □

Implementations of Interval Arithmetic When implementing interval arithmetic on a computer, we no longer work over the reals $\mathbb{R}$, but on a finite set of *machine representable* numbers. As discussed in Section 1.2, one of the most commonly used formats are the floating-point numbers $\mathcal{F}_p$, where p is the precision used (see Definition 1.2.2). So we consider only intervals whose endpoints are in $\mathcal{F}_p$.

Definition 1.3.25 (Floating point Interval)**.** *The set of all floating point intervals is*

$$\mathbb{IF}_p = \{[\underline{x}, \overline{x}]\,|\underline{x}, \overline{x} \in \mathcal{F}_p \text{ and } \underline{x} \leq \overline{x}\}.$$

Since $\mathcal{F}_p$ is not arithmetically closed, when performing arithmetic operations in $\mathbb{IF}_p$ we must round the resulting interval *outwards* to guarantee the inclusion of the true result. By this we mean that the lower bound is rounded down and the upper bound is rounded up. Using this, we can define the four arithmetic operations in a similar manner to what described in Proposition 1.3.8.

Multiple precision Interval Arithmetic - Need and existing packages. Multiple precision arithmetic is a floating-point arithmetic, where the number of digits of the mantissa (i.e. the precision p for $\mathcal{F}_p$) can be any fixed or variable value. It offers the possibility to set precision to an arbitrary value as needed in the computations; this can be done either statically or dynamically, *i.e.* during the computations.

We note quoting [118] that "one should never forget that with 50 bits, one can express the distance from the Earth to the Moon with an error less than the thickness of a bacterium" and that the most accurate physical measurements used in quantum mechanics or general relativity have

a relative accuracy close to 10^{-15}. This of course means that the current formats implemented in almost all current platforms should suffice in general. And yet, one must not forget that sometimes we must carry out computations that must end up with a relative error less than or equal to 10^{-15}, which is much more difficult. So multiple precision is usually applied to problems where it is important to have a high accuracy (e.g., many digits of π), which in turn is needed especially in problems where cancellation (defined in 1.3.23) is present. Such an application will be detailed in Chapter 3.

We mention here several interval packages that are based on multiple precision packages, a detailed comparison is available in [74]:

- IntLab [145, 146] is an interval arithmetic package for MatLab;
- the MPFI [74] package based on MPFR *Multiple Precision Floating-point Interval arithmetic library* *;
- GMP-XSC † based on GMP multiple precision arithmetic;
- `intpakX` ‡ based on Maple arithmetic;
- the XSC language § offers a "staggered" arithmetic, which is a multiple, fixed, precision.

In what follows we chose to use MPFI, a portable library written in C, its source code and documentation can be freely downloaded ¶. It is based on MPFR (*Multiple Precision Floating-point Reliable arithmetic library*), library for arbitrary precision floating-point arithmetic compliant with the IEEE-754-2008 standard [63]. The arithmetic operations are implemented and all functions provided by MPFR are included as well (trigonometric and hyperbolic trigonometric functions and their inverses). The unique feature of MPFI is that for basic elementary functions (ex. $\sin, \cos, \exp$), their evaluation over a given interval returns the tightest interval (with floating-point endpoints) enclosing the image. Conversions to and from usual and GMP data types are available as well as rudimentary input/output functions. The largest achievable computing precision is determined by MPFR and depends in practice on the computer memory. We note that according to tests provided in [74], MPFI is slightly faster than GMP-XSC for basic operations (addition, multiplication), but for basic functions, due mainly to the "correct rounding" feature of MPFR, which makes it slower than GMP, GMP-XSC is faster. However, in our implementations we want to benefit from this feature of MPFR.

Moreover, in [74], experiments showed that `intpakX` is 50 times slower than MPFI for standard operations (addition, multiplication) with standard 15 digits numbers. However, we used `intpakX` for prototyping several algorithms due to the convenience of using features of symbolic computing, or support for various routines, provided directly by Maple. We mention that with MPFR and thus MPFI, exact directed rounding is done, i.e. the resulting intervals are the tightest guaranteed enclosures of the exact results. In `intpakX`, the resulting intervals are rounded outwardly by 1 ulp, yielding an interval with a width of 2 ulps in a single calculation.

1.4 From interval arithmetic to rigorous polynomial approximations

We have seen above that *natural interval extensions* do not yield in general a *sharp* interval bound for elementary functions. This overestimation can be reduced by using different formulas which lead to interval extensions providing sharper enclosures. For that, one could use centered forms as well as some tricks related to monotonicity or convexity test for the function considered, see for example Chap. 6.4. of [115].

*. http://www.mpfr.org/
†. http://www-public.tu-bs.de:8080/~petras/software.html
‡. http://www.math.uni-wuppertal.de/wrswt/software/intpakX/
§. http://www2.math.uni-wuppertal.de/~xsc/
¶. https://gforge.inria.fr/projects/mpfi/

But a simple idea available since Moore [114] would be to replace the considered function f by its truncated Taylor series expansion, supposing that it is differentiable up to a sufficient order. If f behaves sufficiently well, this eventually gives a good enough approximation polynomial T to be used instead of f. Furthermore, for a validated approach, a bound for the error $R = f - T$ is not too difficult to obtain using the following lemma.

Lemma 1.4.1 (Taylor-Lagrange Formula)**.** *If f is $n+1$ times continuously differentiable on a domain $\boldsymbol{I}$, then we can expand f in its Taylor series around any point $x_0 \in \boldsymbol{I}$ and we have according to Lagrange formula:*

$$\forall x \in \boldsymbol{I},\ \exists \xi \text{ between } x_0 \text{ and } x, \text{ s.t. } f(x) = \underbrace{\left(\sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!}(x-x_0)^i\right)}_{T(x)} + \underbrace{\frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}}_{\Delta_n(x,\xi)}. \quad (1.4)$$

In this way, obtaining an interval bound for f reduces to obtaining an interval bound $\boldsymbol{\Delta}$ for $\Delta_n(x,\xi)$ and for T. It is obvious that $(T, \boldsymbol{\Delta})$ represents a *rigorous polynomial approximation* for f where the key ingredient is that Taylor approximations are used for obtaining T. This is classical in literature and usually found under the name of Taylor forms [124]. 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.

1.4.1 Computing the approximation polynomial before bounding the error

We give below a classical procedure that we used in [38], which combines automatic differentiation and interval arithmetic for computing both coefficients of T and for obtaining an interval enclosure $\boldsymbol{\Delta}$ for $\Delta_n(x,\xi)$.

Automatic Differentiation (AD) (also known as *algorithmic differentiation* or *Taylor arithmetic*) is a well-known technique for computing interval enclosures of higher order derivatives for functions. We give some ideas of what AD consists of for univariate functions. We refer the reader to [136, 73, 6].

Automatic Differentiation

Automatic differentiation (AD) allows for evaluating the first n derivatives of f in a point $x_0 \in \mathbb{R}$ in an *efficient* way. The expressions for successive derivatives of practical functions f typically become very involved with increasing n. Fortunately, it is not necessary to generate these expressions for obtaining values of $\{f^{(i)}(x_0), i = 0, \ldots, n\}$.

The general idea is to replace any function g by the array $G = [g_0, \ldots, g_n]$ where $g_i = g^{(i)}(x)/i!$.

There exist formulas for computing successive derivatives for all basic functions, specifically for computing W where w is $\exp$, $\sin$, $\arctan$, etc. These formulas are available since Moore for most basic functions, see [114, 6], where they are written in a case by case manner. In fact, the general idea is to find some recurrence relations between successive derivatives of w. Also, for many basic functions simple closed formulas for the Taylor coefficients exist.

D-finite functions. We note that there exists a very interesting algorithmic approach for finding recurrence relations between the Taylor coefficients for a large class of functions that are solutions of linear ordinary differential equations (LODE) with polynomial coefficients, usually called *D-finite functions* or *holonomic functions*. It is known that the Taylor coefficients of these functions satisfy a linear recurrence with polynomial coefficients [153]. The vast majority of basic functions are *D-finite functions*: it is estimated that about 60% of the functions described in Abramowitz &

Stegun's handbook [1] fall into this category, with a simple counter-example being $\tan$. For all D-finite functions it is possible to generate these recurrence relations directly from the differential equations defining the function, see for example Gfun module in Maple [150]. Once the recurrence relation is found, the computation of the first n coefficients is done in linear time.

Example 1.4.2. *Consider $f = \exp$, it verifies the LODE $f' = f$, $f(0) = 1$, which gives the following recurrence for the Taylor coefficients $(c_n)_{n\in\mathbb{N}}$:*

$$(n+1)c_{n+1} - c_n = 0,\ c_0 = 1,$$

satisfied by $c_n = \frac{1}{n!}$.

We remark that the D-finiteness property opens the way to including in the class of *basic functions* all the D-finite functions, in the sense of the easiness and efficiency of computing Taylor series coefficients.

In what concerns the remaining functions, 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 $\forall i, w_i = u_i + v_i$. Moreover, it is also easy to see (using Leibniz formula) that the array corresponding to $w = uv$ is given by

$$\forall i, w_i = \sum_{k=0}^{i} u_k\, v_{i-k}.$$

More generally, if one knows the array U formed by the values $u^{(i)}(x)/i!$ and the array V formed by the values $v^{(i)}(y)/i!$ where $y = u(x)$, it is possible to compute from U and V the array corresponding to $w = v \circ u$ in x. Hence, given any expression `expr` representing a function f, a straightforward recursive procedure computes the array F:

- if `expr` is a constant c, return $[c,\ 0,\ 0,\ \dots]$;
- if `expr` is the variable, return $[x,\ 1,\ 0,\ \dots]$;
- if `expr` is of the form `expr1 + expr2`, compute recursively the arrays U and V corresponding to `expr1` and `expr2` and returns $[u_0 + v_0,\ \dots,\ u_n + v_n]$;
- etc.

For bounding $f^{(n)}(\boldsymbol{x})$, $\boldsymbol{x} \in \mathbb{IR}$, it suffices to apply the same algorithm using interval arithmetic, replacing x_0 by $\boldsymbol{x_0}$.

Example 1.4.3. *Given $f(x) = \sin(x)\cos(x)$, compute $f^{(4)}(0)$ and $f^{(4)}([0,1])$.*
We compute the successive derivatives for $\sin$ and $\cos$ in $x_0 = 0$:

$$u = [\sin(0), \cos(0), -\sin(0), -\cos(0), \sin(0)],$$

$$v = [\cos(0), -\sin(0), -\cos(0), \sin(0), \cos(0)]$$

and then we apply for example Leibniz formula described above: $[u_0\, v_0, u_0\, v_1 + u_1\, v_0, \dots, u_0\, v_4 + u_1\, v_3 + u_2\, v_2 + u_3\, v_1 + u_4\, v_0]$ and obtain $f^{(4)}(0) = 0$. When we replace the same computations with interval arithmetic we obtain: $f^{(4)}([0,1]) \in [0, 13.5]$. We remark that overestimation is present when applying AD with interval arithmetic also, since the image of f over $[0,1]$ is $[0,8]$.

Indeed, manipulating these arrays is nothing but manipulating truncated formal series. There exist fast algorithms for multiplying, composing or inverting formal series [24, 25]. For instance, computing the first n terms of the product of two series can be performed in $\mathcal{O}(n\,\log(n))$ operations only (instead of the $\mathcal{O}(n^2)$ operations required when using Leibniz formula). We did not implement these techniques for our applications presented in Chapter 3.

By combining automatic differentiation and interval arithmetic, we obtain enclosures of both the coefficients of T, which we denote $\boldsymbol{c_i}, i \in \{0, \dots, n\}$ and the $n+1$-th derivative $\frac{f^{(n+1)}(\boldsymbol{I})}{(n+1)!} \subseteq$

$\boldsymbol{F^{(n+1)}}(\boldsymbol{I})$. Hence $\Delta_n(\boldsymbol{I},\boldsymbol{I}) \subseteq \boldsymbol{F^{(n+1)}}(\boldsymbol{I})(\boldsymbol{I}-x_0)^{n+1}$. Consequently, we can obtain an enclosure of f based on:

(1). a polynomial part $\sum_{i=0}^{n} \boldsymbol{c_i}(x-x_0)^i$;

(2). an interval-valued remainder term $\boldsymbol{\Delta} = \boldsymbol{F^{(n+1)}}(\boldsymbol{I})(\boldsymbol{I}-x_0)^{n+1}$.

We then have the following inclusion:

$$\forall x \in \boldsymbol{I}, f(x) \in \sum_{i=0}^{n} \boldsymbol{c_i}(x-x_0)^i + \boldsymbol{\Delta}, \tag{1.5}$$

which also provides an enclosure of the range of f over $\boldsymbol{I}$:

$$f(\boldsymbol{I}) \subseteq \sum_{i=0}^{n} \boldsymbol{c_i}(\boldsymbol{I}-x_0)^i + \boldsymbol{\Delta}. \tag{1.6}$$

Remark 1.4.4. *We remark that taking $x_0 = mid(\boldsymbol{x})$, $r = rad(\boldsymbol{x})$, the interval widths of $\boldsymbol{\Delta}$ scale like r^{n+1}.*

While this technique is simple, unfortunately, due to the *dependency phenomenon*, the bound obtained for $f^{(n+1)}(\boldsymbol{I})$, and hence $\boldsymbol{\Delta}$, can be highly overestimated.

Example 1.4.5. *Let $f(x) = e^{1/\cos x}$, over $[0,1]$. We consider a Taylor approximation polynomial of degree $n = 14$, expanded around $x_0 = 0.5$. It can be shown (see for example Chapter 3) that the supremum norm of the error between f and T over $[0,1]$ is $\|f-T\|_\infty \leqslant 2.60 \cdot 10^{-3}$. Now, using automatic differentiation and Lagrange formula, one obtains the enclosure:*

$\boldsymbol{\Delta} = [-2.73 \cdot 10^2, 1.78 \cdot 10^3]$. Obviously, the overestimation is too big for the obtained bound to be useful.

We can alleviate this problem using several ideas from literature that we briefly mention here:

Using majorizing series

Besides Lagrange formula, another, more promising, technique to bound R starts with the observation that if f is *analytic* on a complex disc $\mathcal{D}$ containing $\boldsymbol{I}$ and centered on x_0 in the complex plane [2], then, in equation (1.4), $R = f - T$ can be expressed as a series: $R(x) = \sum_{i=n+1}^{+\infty} \frac{f^{(i)}(x_0)}{i!}(x-x_0)^i$. A basic theorem in complex analysis known under the name of *Cauchy's estimate* (see Theorem 4.2.16) and its variants (we refer to [55] for details) allows us to find values M and d such that

$$\forall i > n, \left|\frac{f^{(i)}(x_0)}{i!}\right| \leq \frac{M}{d^i} := b_i. \tag{1.7}$$

We can obviously bound R with

$$\forall x \in \boldsymbol{I}, \quad |R(x)| = \left|\sum_{i=n+1}^{+\infty} \frac{f^{(i)}(x_0)}{i!}(x-x_0)^i\right| \leq \sum_{i=n+1}^{+\infty} b_i\,|x-x_0|^i. \tag{1.8}$$

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:

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

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 [55] for automatically computing suitable majorizing series. ACETAF proposes several techniques, all based on *Cauchy's estimate*.

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 $\mathbf{\Delta}$ was a fairly tight enclosure of the actual range $R(\boldsymbol{I})$. For the previous example, we were able to compute the bound $9.17 \cdot 10^{-2}$, for example.

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 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 $\mathbf{\Delta}$. One typical such case occurs when f has a singularity in the interior of $\mathcal{D}$, but $|f|$ remains bounded over the contour $\partial\mathcal{D}$ [55]. As we 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, but we have to *prove* it automatically, and if possible formally. ACETAF includes a rigorous algorithm for checking the analyticity of a function over a disc based on complex interval arithmetic, but it is not formally proven.

As an alternative to methods based on Cauchy's estimate, Mezzarobba and Salvy show in [108] how to compute an accurate majorizing series and its bound, for the special class of D-finite functions. We mentioned above that most, 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. So this approach is limited to the class of D-finite functions.

Using an interpolation polynomial and automatic differentiation.

Instead of Taylor approximations, interpolation polynomials could be a good choice: first they are easy to compute; the reader can find techniques in any book on numerical analysis, we briefly discuss some in Section 4.3.1. Second, it is well-known (see Section 4.3.1 for details) that when using suitable interpolation points, a near-best polynomial approximation is obtained. Finally, we can rigorously bound the approximation error using the following formula (see Equation (4.23)): if P interpolates f at points $y_0, \ldots, y_n \in \boldsymbol{I}$, the error $R = f - P$ satisfies

$$\forall x \in \boldsymbol{I},\ \exists\, \xi \in \boldsymbol{I},\ R(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!} \prod_{i=0}^{n} (x - y_i), \tag{1.9}$$

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

When bounding R using (1.9), the only difficulty is to bound $f^{(n+1)}(\xi)$ for $\xi \in \boldsymbol{I}$. For that, we apply the same automatic differentiation algorithm as in the case of Taylor approximations explained before. Hence it leads to exactly the same problem of overestimation, with the advantage that roughly speaking, $\prod_{i=0}^{n}(x - y_i)$ can be taken to be 2^n times smaller than in the case of Taylor approximations (see for example Lemma 4.2.2, details are given in Section 4.2). However, in general, due to the overestimation in $\mathbf{\Delta}$, we may lose all the benefit of using an interpolation polynomial, although the actual bound $\|R\|_\infty$ is almost optimal. If we consider the above example, the bound for $\|R\|_\infty$ obtained using AD would be 0.11, while $\|R\|_\infty \simeq 6.10 \cdot 10^{-7}$.

The above analysis 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 $\mathbf{\Delta}$ is less prone to overestimation.

1.4.2 Simultaneously computing the approximation polynomial and the error

Both techniques presented so far consist in two separate steps: first we compute the approximation polynomial and afterwards we rigorously bound the approximation error by $\boldsymbol{\Delta}$. Simultaneous computation of both the polynomial and the error bound is also possible.

Such a technique has been promoted by Berz and Makino [97, 98] under the name of *Taylor models*. More precisely, given a function f over an interval $\boldsymbol{I}$, 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 $\forall x \in \boldsymbol{I},\ f(x) - T_f(x) \in \boldsymbol{\Delta_f}$. 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 $\boldsymbol{I}$, Taylor models are likely to compute a very bad bound $\boldsymbol{\Delta}$. However, this bound remains rigorous in any case.

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:

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

When bounding this remainder with automatic differentiation, an interval enclosure $\boldsymbol{J}$ of $(h \circ u)^{(n+1)}(\boldsymbol{I})/(n+1)!$ is computed. This interval $\boldsymbol{J}$ is obtained by performing many operations involving enclosures of all the $u^{(i)}(\boldsymbol{I})/i!$. 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

$$w(x) = h(u(x)) = \underbrace{\sum_{i=0}^{n} \frac{h^{(i)}(u(x_0))}{i!} (u(x) - u(x_0))^i}_{=:\, S(x)} + \frac{h^{(n+1)}(u(\xi))}{(n+1)!} (u(x) - u(x_0))^{n+1}. \tag{1.10}$$

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.

Let us now see some numerical examples.

1.4.3 Practical comparison of the different methods

Table 1.3 shows the quality of some bounds obtained by the methods that we have presented. The function f, the interval $\boldsymbol{I}$ and the degree n of T are given in the first column. The interpolation was performed at Chebyshev points of first kind (see Equation (4.2)) and the corresponding approximation error R was bounded using automatic differentiation with interval arithmetic as explained in Section 1.4.1. 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 $\boldsymbol{I}$. The resulting interval bound was computed using ACETAF as presented in Section 1.4.1. 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 1.3, even this first simple method gives results often as accurate as Taylor models, or even better.

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 $\boldsymbol{I}$ 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 example are composite functions on fairly wide intervals, which challenges 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. We observe that for simple functions, the bound obtained with interpolation is better, while for composite functions, Taylor models, although based on worse-quality approximations, offer a tighter bound. We explain in detail the "philosophy" of Taylor models in Chapter 2. This allows us to:

– have a detailed account that we could not find elsewhere in literature and that we intend to use as reference document for one of our on-going projects of their formalization in a formal proof checker.

– give a refinement of Taylor Models in the sense that we offer a solution to the practical difficulty of having useful Taylor Models for functions that have removable discontinuities. For example, the classical way of computing Taylor Models would return an infinite error bound for the function $\varepsilon(x) = \dfrac{x - \frac{1}{6}x^3}{\sin(x)} - 1$, on a tight interval around zero, say $\boldsymbol{I} = [-\pi/64; \pi/64]$. This kind of functions are common in our applications: this is a relative error function which quantifies the approximation error between $\sin$ and the approximation polynomial $p(x) = x - \frac{1}{6}x^3$, which is a quite common approximation for $\sin$. Both p and $\sin$ vanish at $z_0 = 0$, but the approximation error stays bounded.

– pursue the symbolic-numeric approach for creating an algebra with RPAs that will allow us to use in an efficient way the near-minimax character of interpolation polynomials for example, in order to reduce the gap between the validated and estimated error bounds presented in columns 2 and 3 of Table 1.3. This will lead us to obtaining new efficient RPAs based on better approximations in Chapters 4 and 5.

In order to facilitate the explanations and development of an algebra with such models, we

$f(x)$, $\boldsymbol{I}$, 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

Table 1.3: Examples of bounds obtained by several methods

give a brief explanation of how we chose to represent *rigorous polynomial approximations* in what follows.

1.5 Data structures for rigorous polynomial approximations

We already noticed above that we computed interval enclosures of both the coefficients of the Taylor polynomial T and the error bound R. Hence we obtained "a couple" (T, Δ), but where in fact T was a polynomial with interval coefficients. In the following we chose to represent the polynomial T with *tight interval coefficients*. We note that we do not impose yet any representation basis for the polynomial. Above, the monomial basis was used, but this is not a constrained choice.

Definition 1.5.1 (Rigorous Polynomial Approximation Structure)**.**
Let $\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}$ and $\boldsymbol{\Delta}$ be $n+2$ intervals. We call $M = (\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}, \boldsymbol{\Delta})$ a rigorous polynomial approximation structure.

Our choice may seem strange, since usually, one represents T with floating-point coefficients: it saves both memory (half the memory is needed with floating-point numbers in comparison with interval coefficients) and time (interval arithmetic is at least twice slower than floating-point arithmetic [74]). When performing floating-point operations on the coefficients, roundings occur: these rounding errors are usually added to the remainder $\boldsymbol{\Delta}$ [97]. In contrast, using interval coefficients makes programming and prototyping algorithms easier, and more importantly, it ensures that the "true" real coefficients of the approximation polynomial we consider (Taylor, Chebyshev, etc.) are contained in the corresponding intervals. This property will be essential for handling removable discontinuities of functions in Section 2.3.

CHAPTER 2

Taylor Models

În redactare nu are atâta preț poleirea frazelor, cât organizarea ideilor.
Ion Barbu

Taylor models [97, 124, 98] are a well-known tool for obtaining rigorous polynomial approximations based on Taylor approximations. However, we encountered several difficulties in using it in an "off-the-shelf" manner. Among these, the implementations are scarce or not freely available, no multiple precision support for computations was available, we could not deal with a class of functions sufficiently large for our purposes, etc.

Hence, we proceeded to implementing Taylor Models-like algorithms in our software tool Sollya. Our implementation deals only with univariate functions, but several refinements were integrated, several data structure choices are different. The most important refinement is the possibility of handling the practical difficulty of functions that have removable discontinuities. For example, the classical way of computing Taylor Models would return an infinite error bound for the function $\varepsilon(x) = \dfrac{x - \frac{1}{6}\,x^3}{\sin(x)} - 1$, on a tight interval around zero, say $\boldsymbol{I} = \left[-\frac{\pi}{64}; \frac{\pi}{64}\right]$. This kind of functions are common in our applications: this is a relative error function which quantifies the approximation error between $\sin$ and the approximation polynomial $p(x) = x - \frac{1}{6}\,x^3$, which is a quite common approximation for $\sin$. Both p and $\sin$ vanish at $z_0 = 0$, but the approximation error stays bounded.

Furthermore, we chose to explain in detail in this chapter the algorithms for Taylor Models, because, to our knowledge, there is no such detailed account in the literature. This will also allow us to explain the "philosophy" of Taylor models that will be used in subsequent chapters for obtaining new efficient RPAs based on better approximations. One other potential use is as reference document for one of our on-going projects of their formalization in a formal proof checker.

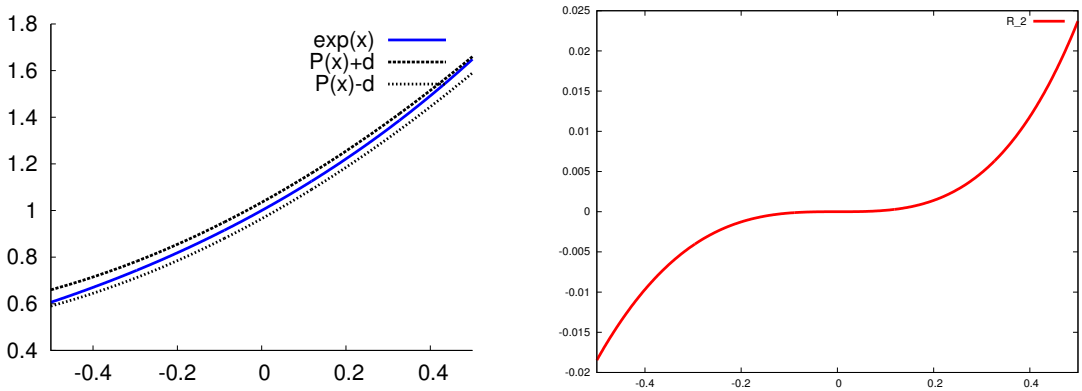
The algorithms given in this chapter are used inside the process of certifying supremum norms of approximation errors presented in Chapter 3. This is a joint work with S. Chevillard, J. Harrison and C. Lauter.

2.1 Basic principles of Taylor Models

Taylor models [97, 98, 14, 15], introduced by Berz and his group, provide another way to rigorously manipulate and evaluate functions using floating-point arithmetic. They have been widely used for validated computing: in global optimization and range bounding [97, 14, 36, 15], for validated solutions of ODEs [123], rigorous quadrature [13], etc.

A Taylor model (TM) of order n for a function f which is supposed to be $n+1$ times continuously differentiable over an interval $[a,b]$, is a rigorous polynomial approximation (RPA, see Problem 1, page 17) of f. Specifically, it is a couple $(P, \mathbf{\Delta})$ formed by a polynomial P of degree n, and an interval part $\mathbf{\Delta}$, such that $f(x) - P(x) \in \mathbf{\Delta}, \forall x \in [a,b]$. Roughly speaking, as their name suggests, the polynomial can be seen as a Taylor expansion of the function at a given point. The interval $\mathbf{\Delta}$ (also called interval *remainder*) provides the validation of the approximation, meaning that it provides an enclosure of all the approximation errors encountered (truncation, roundings).

Example 2.1.1. *Let $\boldsymbol{I} = [-\frac{1}{2}, \frac{1}{2}]$, $f(x) = \exp(x)$. Its Taylor series around $x_0 = 0$ is: $f(x) = \sum_{i=0}^{\infty} \frac{1}{i!} x^i$. Consider for example $n = 2$, $P(x) = 1 + x + 0.5x^2$. We will see in the following (Example 2.2.3) that it can be proved that $|f(x) - T(x)| \leqslant d$, where $d = 0.035$. Hence, we can say that $(1 + x + 0.5x^2, [-d, d])$ is a TM of order 2, for f, over $\boldsymbol{I}$. The visual interpretation of a TM is* a tube around the function f. *In Figure 2.1(a), f is plotted over $\boldsymbol{I}$. We observe that since $|f(x) - P(x)| \leqslant d$, we have $P(x) - d \leqslant f(x) \leqslant P(x) + d$, for all $x \in \boldsymbol{I}$.*



(a) (b)

Figure 2.1: A TM $(P, [-d, d])$ of order 2 for $\exp(x)$, over $\boldsymbol{I} = [-\frac{1}{2}, \frac{1}{2}]$. $P(x) = 1 + x + 0.5x^2$ and $d = 0.035$. We can view a TM as a *a tube around the function* in (a). The actual error $R_2(x) = \exp(x) - (1 + x + 0.5x^2)$ is plotted in (b).

As we mentioned above, TMs are based on Taylor approximations. In fact, for functions like trigonometric, exponential, logarithmic functions, as well as operations like $1/x$ or the power function, all referred to in this work as basic functions (Definition 1.3.15, or as intrinsics in [97]), the polynomial coefficients and the error bounds can be easily computed based on the Taylor-Lagrange Formula and automatic differentiation as discussed in Section 1.4. However, the approach is different for composite functions: simple algebraic rules like addition, multiplication and composition with TMs are applied recursively on the structure of the function f, such that the final TMs is an RPA for f over $[a,b]$. As we will see in the following, due to this different approach, TMs offer usually a much tighter error bound than the one directly computed for the whole function, for example using the technique from Section 1.4.

In what follows we define more formally what such models are, and then describe algorithms for computing Taylor Models of functions. We note that we consider *only* univariate functions f, which are supposed to be sufficiently *smooth* *.

*. This means that given n, the order of TM we want to compute, over a given interval $[a,b]$, f is $n+1$ times continuously differentiable over an interval $[a,b]$.

2.1.1 Definitions and their ambiguities

In literature, the consecrated definition of a Taylor Model states that it is a couple $(T,\ \boldsymbol{\Delta})$, where T is a polynomial and $\boldsymbol{\Delta}$ is an interval bounding the absolute error between T and the function it represents. The name implicitly suggests that the method used is based on Taylor approximations. However, since we present in this work several ways of obtaining such couples, and some of the methods do not use Taylor approximations, the consecrated definition becomes improper. Hence, we make the following important remark:

Remark 2.1.2. *We say that Taylor Models provide one special kind of rigorous polynomial approximations and for short each time we refer to Taylor Models, it is implicit that the underlying methods used for obtaining this RPA are based on:*
- *using truncated Taylor series expansions for basic functions;*
- *recursively applying algebraic rules with TMs on the structure of composite functions.*

Moreover, several more clarifications are necessary:
- Usually, $\boldsymbol{\Delta}$ is an interval bound for the absolute error between the polynomial and the function, however, as we will see in the following, it does not permit to compute a finite error bound for functions with *removable discontinuities* such as $\sin(x)/x$. In order to overcome this issue, we were led to modify this specification and we will discuss this in detail in Section 2.3. To make the distinction between the standard TMs and the modified ones, we call them Taylor Models with *absolute* remainder and respectively Taylor Models with *relative* remainder. Sometimes, for brevity, and when no ambiguity occurs, when we refer to the standard TMs, the term *absolute* remainder is omitted.
- We chose to represent the polynomial T with *tight interval coefficients*: this is not usual. Usually, one represents T with floating-point coefficients: it saves both memory (half the memory is needed with floating-point numbers in comparison with interval coefficients) and time (interval arithmetic is at least twice slower than floating-point arithmetic [74]). When performing floating-point operations on the coefficients, roundings occur: these rounding errors are usually added to the remainder $\boldsymbol{\Delta}$ [97]. In contrast, using interval coefficients makes programming easier (you do not have to care about the rounding errors) and, more importantly, it ensures that the true coefficients of the Taylor polynomial lie inside the corresponding intervals. This property will be essential for handling removable discontinuities as explained later in Section 2.3.

Now, the formal definition is the following:

Definition 2.1.3 (Taylor Model with Absolute Remainder)**.** *Let $f : \boldsymbol{I} \to \mathbb{R}$ be a function, $\boldsymbol{x_0}$ be a small interval around an expansion point x_0. Let $M = (\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta})$ be an RPA structure. We say that M is a Taylor Model with Absolute Remainder of f at $\boldsymbol{x_0}$ on $\boldsymbol{I}$ when*

$$\begin{cases} \boldsymbol{x_0} \subseteq \boldsymbol{I}, \\ 0 \in \boldsymbol{\Delta}, \\ \forall \xi_0 \in \boldsymbol{x_0}, \exists \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta}, f(x) - \sum\limits_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i = \delta. \end{cases}$$

Remark 2.1.4. *The order (degree) of the Taylor model defined above is n.*

We observe that evaluating or bounding the image of f reduces to evaluating or bounding the image of T, a polynomial with *tight interval coefficients*, written in the monomial basis. We will give several ideas about that in the following.

2.1.2 Bounding polynomials with interval coefficients

Let us first give a formal definition for a valid polynomial bound for T, over $\boldsymbol{I}$ and then discuss methods for obtaining such valid bounds.

Definition 2.1.5 (Valid polynomial bound)**.** *Let $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ be intervals around polynomial coefficients. Let $\boldsymbol{I}$ be an interval and $\boldsymbol{x_0}$ an expansion interval.*

An interval $\boldsymbol{B}$ is called a valid polynomial bound when

$$\forall \xi_0 \in \boldsymbol{x_0}, \forall \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \forall x \in I, \sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i \in \boldsymbol{B}.$$

Remark 2.1.6. *We remark that, if $\boldsymbol{x_0} \subseteq \boldsymbol{I}$, we can take $x = \xi_0$ in the previous equation and in this particular case we obtain $\forall \alpha_0 \in \boldsymbol{a_0}, \alpha_0 \in \boldsymbol{B}$. In other words* if $\boldsymbol{x_0} \subseteq \boldsymbol{I}$ and if $\boldsymbol{B}$ is a valid polynomial bound for $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}$ and $\boldsymbol{I}$, we have $\boldsymbol{a_0} \subseteq \boldsymbol{B}$.

One simple way to obtain a valid polynomial bound is to use a Horner scheme based evaluation, that is known to reduce the overestimation and is described in Algorithm 2.1.1.

```
1 Algorithm: ComputeBound(a_0, ..., a_n, x_0, I)
  Input: a_0, ..., a_n tight intervals
  I an interval
  x_0 an interval
  Output: a valid polynomial bound
2 B ← [0; 0] ;
3 for i ← n downto 0 do
4 |   B ← B · (I − x_0) + a_i ;
5 end
6 return B ;
```

Algorithm 2.1.1: Computation of a valid polynomial bound

Proof of Algorithm 2.1.1. We prove the following loop invariant: at the end of each iteration of the loop, $\boldsymbol{B}$ is a valid polynomial bound for $\boldsymbol{a_i}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{I}$. For $i = n$, it is trivial, since $\boldsymbol{B} = \boldsymbol{a_n}$ in this case. We suppose that the property holds at the end of i-th iteration, and prove that it holds at the end of the $(i-1)$-th. We have

$$\boldsymbol{B}_{\text{new}} \leftarrow \boldsymbol{B}_{\text{old}} \cdot (\boldsymbol{I} - \boldsymbol{x_0}) + \boldsymbol{a_i}.$$

Let $\xi_0 \in \boldsymbol{x_0}$, $\alpha_k \in \boldsymbol{a_k}$ for $k = i-1, \ldots, n$, and $x \in \boldsymbol{I}$: we shall prove that

$$\sum_{k=i-1}^{n} \alpha_k \left(x - \xi_0\right)^{k-i+1} \in \boldsymbol{B}_{\text{new}}.$$

By induction hypothesis, we know that $\sum_{k=i}^{n} \alpha_k \left(x - \xi_0\right)^{k-i} \in \boldsymbol{B}_{\text{old}}$. Since $x \in \boldsymbol{I}$ and $\xi_0 \in \boldsymbol{x_0}$, we have $x - \xi_0 \in \boldsymbol{I} - \boldsymbol{x_0}$ and hence,

$$\left(\sum_{k=i}^{n} \alpha_k \left(x - \xi_0\right)^{k-i}\right) \left(x - \xi_0\right) \in \boldsymbol{B}_{\text{old}} \cdot (\boldsymbol{I} - \boldsymbol{x_0}).$$

Finally, $\left(\sum_{k=i}^{n} \alpha_k \left(x-\xi_0\right)^{k-i}\right)(x-\xi_0)+\alpha_{i-1} \in \boldsymbol{B}_{\text{new}}$. □

Remark 2.1.7. *We note that we could roughly see Algorithm 2.1.1 as an evaluation of the polynomial, using natural interval extension of the polynomial expression written in Horner form.*

We have seen that in general, natural interval extension does not provide a sharp interval bound. But, when using Taylor models, obtaining sharp interval bounds for polynomials is very important [124]. As we will see in the following, for each operation involving Taylor models, interval bounds on polynomials appearing in intermediary computations are necessary. These bounds influence directly the final quality of the remainder obtained. We deal here only with univariate polynomials, and we mention just that for multivariate polynomials, many techniques for range bounding were proposed in [152]. Also, Taylor models interval extensions using Bernstein expansions were studied [121].

Nevertheless, for univariate polynomials, obtaining the *exact* range of a polynomial is possible by using well-established techniques for isolating the roots of its derivative. Specifically, this means that given a univariate real polynomial T, we now want to compute a list $\mathcal{Z}$ of disjoint thin intervals $\boldsymbol{z}$ such that each root of T lies in one of the intervals $\boldsymbol{z}$. We may use general methods such as the interval Newton method [139]. But we can also take advantage of the fact that T is a polynomial and use a specific method for isolating the real roots of a polynomial. There are two main classes of such specific methods which we mention for completeness, but we do not enter the technical details in this work: methods based on "Descartes' rule of signs", see for example [143] and methods based on Sturm's method, see [144] for an overview.

For this, we need an automatic and reversible process that given a TM consisting of a polynomial with *tight interval coefficients* and a *remainder bound* as defined above, returns a polynomial with floating-point coefficients and a new rigorous remainder bound. It is not difficult to see that we can do that by by taking a FP number contained in each interval coefficient (for example the middle of the interval, if it happens to be a FP, otherwise an endpoint of the interval) as the new coefficient and accumulating the small errors in the coefficients to the final error bound. This is the idea of Algorithm 2.1.2.

Proof of Algorithm 2.1.2. Let us fix FP numbers $\xi_0 \in \boldsymbol{x_0}$, $t_i \in \boldsymbol{a_i}$, with the common choice $t_i = \operatorname{mid}(\boldsymbol{a_i})$ and the polynomial $\widetilde{T}(x)$ of degree n,

$$\widetilde{T}(x)=\sum_{i=0}^{n} t_i(x-\xi_0)^i.$$

We have from Definition 2.1.3:

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

The difference between $\sum_{i=0}^{n} \alpha_i\left(x-\xi_0\right)^i$ and $\widetilde{T}(x)$ can easily be bounded using interval arithmetic. We have $\alpha_i \in \boldsymbol{a_i}$, and thus $(\alpha_i - t_i) \in [\inf(\boldsymbol{a_i}) - t_i, \sup(\boldsymbol{a_i}) - t_i]$, which leads to

$$\sum_{i=0}^{n-1}(\alpha_i-t_i)(x-\xi_0)^i \in \sum_{i=0}^{n-1}[\inf(\boldsymbol{a_i})-t_i, \sup(\boldsymbol{a_i})-t_i] \cdot (\boldsymbol{I}-\xi_0)^i=\boldsymbol{\theta}.$$

Finally, the error between f and $\widetilde{T}$ is bounded by $\boldsymbol{\theta}+\boldsymbol{\Delta}$. □

We will now give detailed algorithms for computing Taylor Models with absolute remainder.

```
1  Algorithm: ComputeFP-RPAfromTM((a_0, ..., a_n, Δ), x_0, I)
   Input: I an interval,
   (a_0, ..., a_n, Δ) a TM of degree n at x_0 on I for some function.
   Output: an RPA for the respective function, whose polynomial has FP coefficients
2  ξ_0 = mid (x_0);
3  if ξ_0 is not a FP number then
4  |  ξ_0 = inf(x_0);
5  end
6  for k ← 0 to n do
7  |  t_i = mid (a_i);
8  |  if t_i is not a FP number then
9  |  |  t_i = inf(a_i);
10 |  end
11 |  b_i = a_i − t_i;
12 end
13 θ ← ComputeBound(b_0, ..., b_n, x_0, I);
14 Δ ← Δ + θ;
15 return (t_0, ..., t_n, ξ_0, Δ) ;
```

Algorithm 2.1.2: Computation of a polynomial with FP coefficients and a rigorous error bound from a TM.

2.2 Taylor Models with Absolute Remainder

We start with algorithms for computing Taylor Models for basic functions and then we continue with describing operations like addition, multiplication, composition with this models.

2.2.1 Taylor Models for basic functions

For each basic function f like identity, $\sin, \cos, \exp$, etc. a case-by-case algorithm has to be written. This can be tedious when programming Taylor Models, because it has mainly to be done by hand, and proved case by case. We give below the example of the constant, identity and then we describe the process to follow for other functions and exemplify on $\sin$. We note that we will use and have available an algorithm $\texttt{eval}(f, \boldsymbol{x})$ that for a function f and an interval $\boldsymbol{x}$ returns a valid bound for f over $\boldsymbol{x}$ as explained in Remark 1.3.20.

```
1  Algorithm: TMConst(c, n)
   Input: a (usually small) interval c, n ∈ ℕ
   Output: a Taylor structure which is a Taylor Model for any γ ∈ c
2  a_0 ← c ;
3  a_1 ... a_n ← [0, 0] ;
4  Δ ← [0, 0] ;
5  M ← (a_0, ..., a_n, Δ);
6  return M ;
```

Algorithm 2.2.1: Computation of a Taylor Model of a constant function

Constant function

Proof of Algorithm 2.2.1. We prove that, for any interval $\boldsymbol{I}$, any expansion interval $\boldsymbol{x_0} \subseteq \boldsymbol{I}$ and any value $\gamma \in \boldsymbol{c}$, the returned structure M is a Taylor Model of the constant function $x \mapsto \gamma$, around $\boldsymbol{x_0}$ over $\boldsymbol{I}$.

Let $\boldsymbol{I}$, $\boldsymbol{x_0}$ and γ as above. Assume that $\xi_0 \in \boldsymbol{x_0}$. We choose $\alpha_0 = \gamma \in \boldsymbol{a_0}$ and, for $i \in \{1, \ldots, n\}$, $\alpha_i = 0 \in \boldsymbol{a_i}$. Then, for any $x \in \boldsymbol{I}$, we choose $\delta = 0 \in \boldsymbol{\Delta} = [0, 0]$. The property trivially holds. □

Algorithm: `TMVar`$(\boldsymbol{I}, \boldsymbol{x_0}, n)$
Input: $I, \boldsymbol{x_0} \subseteq \boldsymbol{I}, n \in \mathbb{N}$
Output: a Taylor Model M of the identity function $x \mapsto x$
$\boldsymbol{a_0} \leftarrow \boldsymbol{x_0}$;
$\boldsymbol{a_1} \leftarrow [1, 1]$;
$\boldsymbol{a_2} \ldots \boldsymbol{a_n} \leftarrow [0, 0]$;
$\boldsymbol{\Delta} \leftarrow [0, 0]$;
$M \leftarrow (\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 2.2.2: Computation of a Taylor Model of the identity function

Identity function

Proof of Algorithm 2.2.2. Assume that $\xi_0 \in \boldsymbol{x_0}$ and $\alpha_0 = \xi_0 \in \boldsymbol{a_0}$. Moreover, we choose $\alpha_1 = 1$ and $\alpha_2 = \cdots = \alpha_n = 0$. Assume that $x \in \boldsymbol{I}$. We choose $\delta = 0 \in \boldsymbol{\Delta} = [0, 0]$. The property trivially holds. □

Now, let us see how we can compute TMs for basic functions like $f = \sin, \cos, \exp$, etc.

Computing the coefficients and the remainder for basic functions. We use like in Section 1.4, the Taylor-Lagrange formula (see Lemma 1.4.1) that we recall below for clarity:

Let $n \in \mathbb{N}$. Any $n+1$ times differentiable function f over an interval $\boldsymbol{I} = [a, b]$ around $x_0 \in \boldsymbol{I}$ can be written according to Lagrange formula:

$$f(x) = \sum_{i=0}^{n} \frac{f^{(i)}(x_0)}{i!} \cdot (x - x_0)^i + \Delta_n(x, \xi),$$

where $\Delta_n(x, \xi) = \dfrac{f^{(n+1)}(\xi)(x - x_0)^{n+1}}{(n+1)!}$, $x \in [a, b]$ and ξ lies strictly between x and x_0.

We face two problems: enclosing the coefficients and the error. We have already seen in Section 1.4 that this can be easily done for basic functions and we already have available either closed formulas or recurrence relations between successive Taylor coefficients of these functions. The only inconvenient comes from the fact that when using Interval Arithmetic and Lagrange formula for bounding $\Delta_n(\boldsymbol{I}, \boldsymbol{I})$, we have sometimes an important overestimation for the true error bound, so, we will try to take advantage of Lemma 5.12 of [176]. It is based on the following remark: often, the absolute error between a function and its Taylor polynomial is monotonic (in fact, it behaves more or less like the first neglected term, i.e. $(x - \xi_0)^{n+1}$); if we manage to prove the monotonicity, we obtain a very sharp (yet safe) bound for the error by evaluating it at the endpoints of the interval.

Proposition 2.2.1 (Adaptation of Lemma 5.12 of [176]). *Let f be a function defined over an interval $\boldsymbol{I} = [a, b]$; let $\xi_0 \in \boldsymbol{I}$ and let $n \geqslant 0$ be an integer. If the sign of $f^{(n+1)}$ is constant over $\boldsymbol{I}$, then the remainder between f and its Taylor expansion of degree n around ξ_0 is monotonic on $[a, \xi_0]$ and on $[\xi_0, b]$.*

Proof. For any function f, we denote by $T_n(f)$ the Taylor polynomial of degree n and by $R_n(f) = f - T_n(f)$ the corresponding remainder. We remark that $R_n(f)' = f' - T_n'(f) = f' - T_{n-1}(f') = R_{n-1}(f')$. We shall show that that $R_n(f)$ is monotonic on $[a, \xi_0]$ and on $[\xi_0, b]$. Thus it suffices to show that the sign of $R_{n-1}(f')$ is constant.

We can express $R_{n-1}(f')$ by means of the Lagrange formula (applied to f'):

$$R_{n-1}(f')(x) = \frac{(x-\xi_0)^n}{n!} f'^{(n)}(\xi)$$

for some ξ between x and ξ_0. By hypothesis $f'^{(n)}(\xi) = f^{(n+1)}(\xi)$ has a constant sign over $\boldsymbol{I}$. Hence the sign of $R_{n-1}(f')(x)$ is constant over $[a, \xi_0]$ and over $[\xi_0, b]$. This concludes the proof. □

We can use the above proposition for bounding the remainder of basic functions when we construct their Taylor Model. Namely, we evaluate $f^{(n+1)}(\boldsymbol{I})$ by interval arithmetic: if the resulting interval contains numbers of different signs, we cannot apply the technique and fall back to the classical bound given by Lagrange formula. If, on the contrary, the resulting interval contains only nonnegative numbers or only nonpositive numbers, the hypothesis of Proposition 2.2.1 holds. Let R be the remainder: it is hence monotonic on $[a, \xi_0]$ and $[\xi_0, b]$. Hence, $R([a, \xi_0])$ is the interval with bounds $R(a)$ and $R(\xi_0)$ (and similarly for $R([\xi_0, b])$). We conclude that $R(\boldsymbol{I})$ is the smallest interval containing $R(a)$, $R(\xi_0)$ and $R(b)$.

Example 2.2.2. *Let $f = \exp$, $\boldsymbol{I} = [-0.5, 0.5], n = 2, \xi_0 = 0$ and $T_2(x) = 1 + x + 0.5x^2$. The error $R_2(x) = \exp(x) - T_2(x)$ is plotted in Figure 2.1(b). Using Lagrange formula (Lemma 1.4.1) we obtain: $R_2(\boldsymbol{I}) \subseteq \frac{\exp(\boldsymbol{I})}{3!} \cdot \boldsymbol{I}^3$, which gives: $R_2(\boldsymbol{I}) \subseteq [-0.035, 0.035]$. This is an overestimation of $R_2(\boldsymbol{I})$ as we can observe from the figure.*

By computing the 3rd derivative of f over $\boldsymbol{I} = [-0.5, 0.5]$, we obtain: $f^{(3)}(\boldsymbol{I}) \subseteq [0.60, 1.65]$, hence according to Prop. 2.2.1, R_2 is monotonic over $[-0.5, 0]$ and $[0, 0.5]$. We can compute almost the exact[*] *image $R_2(\boldsymbol{I})$ by 3 evaluations: $R_2([-0.5]) \subseteq [-0.0185, -0.0184]$, $R_2([0]) \subseteq [0]$ and $R_2([0.5]) \subseteq [0.0236, 0.0237]$[†]. We obtain $R_2(\boldsymbol{I}) \subseteq [-0.0185, 0.0237]$.*

Let us now describe and prove the algorithm used for computing a Taylor Model for sin.

The sine function

Proof of `TMSin`. Assume that $\xi_0 \in \boldsymbol{x_0}$. For all i we choose $\alpha_i = \sin^{(i)}(\xi_0)/i!$. We know that

- $\sin^{(i)}(\xi_0) = \sin(\xi_0)$ when $i = 0 \mod 4$;
- $\sin^{(i)}(\xi_0) = \cos(\xi_0)$ when $i = 1 \mod 4$;
- $\sin^{(i)}(\xi_0) = -\sin(\xi_0)$ when $i = 2 \mod 4$;
- $\sin^{(i)}(\xi_0) = -\cos(\xi_0)$ when $i = 3 \mod 4$.

Hence, for all i, we have $\alpha_i \in \boldsymbol{a_i}$. Moreover, we remark that, for all $\xi \in \boldsymbol{I}$, $\sin^{(n+1)}(\xi)/(n+1)! \in \Gamma$. Now, assume that $x \in \boldsymbol{I}$. From Lemma 1.4.1, there exists ξ between ξ_0 and x such that

$$\delta := \sin(x) - \sum_{i=0}^{n} \alpha_i (x-\xi_0)^i = (x-\xi_0)^{n+1} \frac{\sin^{(n+1)}(\xi)}{(n+1)!}.$$

∗. the only overestimation errors in this case are due to rounding errors occurring in computations with point intervals and are of the order of precision used for computation.

†. Note that this is a toy example, only 3 significant digits were used for the sake of clarity.

```
1  Algorithm: TMSin(I, x_0, n)
   Input: I, x_0 ⊆ I, n ∈ ℕ
   Output: a Taylor Model M of the sine function sin
2  for i ← 0 to n do
3      switch i mod 4 do
4          case 0: a_i ← eval (sin (x) /i!, x_0);
5          case 1: a_i ← eval (cos (x) /i!, x_0);
6          case 2: a_i ← eval (− sin (x) /i!, x_0);
7          case 3: a_i ← eval (− cos (x) /i!, x_0);
8      endsw
9  end
10 switch (n + 1) mod 4 do
11     case 0: Γ ← eval (sin(x)/i!, I);
12     case 1: Γ ← eval (cos(x)/i!, I);
13     case 2: Γ ← eval (− sin(x)/i!, I);
14     case 3: Γ ← eval (− cos(x)/i!, I);
15 endsw
16 if (sup(Γ) ≤ 0) or (inf(Γ) ≥ 0) then
17     a ← [inf(I)];
18     b ← [sup(I)];
19     Δ_a ← eval (sin(x), a) − ComputeBound(a_0, ..., a_n, x_0, a);
20     Δ_b ← eval (sin(x), b) − ComputeBound(a_0, ..., a_n, x_0, b);
21     Δ_x0 ← eval (sin(x), x_0) − ComputeBound(a_0, ..., a_n, x_0, x_0);
22     Δ ← [min(inf(Δ_a), inf(Δ_x0), inf(Δ_b)), max(sup(Δ_a), sup(Δ_x0), sup(Δ_b))];
23 else
24     V ← eval ((x − y)^(n+1), (I, x_0));
25     Δ ← V · Γ;
26 end
27 M ← (a_0, ..., a_n, Δ);
28 return M ;
```

Algorithm 2.2.3: Computation of a Taylor Model of the sine function

Since $\xi_0 \in \boldsymbol{x_0} \subseteq \boldsymbol{I}$ and since $x \in \boldsymbol{I}$, we have $\xi \in \boldsymbol{I}$ (convexity of $\boldsymbol{I}$). Hence $\delta(x) \in \boldsymbol{V} \cdot \boldsymbol{\Gamma} = \boldsymbol{\Delta}$. Finally, since $\boldsymbol{x_0} \subseteq \boldsymbol{I}$ and $n \geq 0$, $\boldsymbol{V}$ contains 0. Thus, $\boldsymbol{\Delta}$ contains 0. This concludes the proof in the case when the execution goes through the "else" branch.

If, on the contrary, the execution goes through the "then" branch: in this case, $\sup(\boldsymbol{\Gamma}) \leq 0$ or $\inf(\boldsymbol{\Gamma}) \geq 0$. Suppose, for instance, that we are in the case when $\sup(\boldsymbol{\Gamma}) \leq 0$ (the other case is handled similarly). Since $\forall x \in \boldsymbol{I}, \sin^{(n+1)}(x)/(n+1)! \in \boldsymbol{\Gamma}$, we have, in particular,

$$\forall x \in \boldsymbol{I},\ \sin^{(n+1)}(x)/(n+1)! \leq \sup(\boldsymbol{\Gamma}) \leq 0.$$

Hence, the hypothesis of Proposition 2.2.1 holds and it follows that $x \mapsto \delta(x)$ is monotonic on $[\inf(\boldsymbol{I}),\, \xi_0]$ and on $[\xi_0,\, \sup(\boldsymbol{I})]$. Since `ComputeBound` computes a valid polynomial bound, and since $\alpha_i \in \boldsymbol{a_i}$ for all i, $\xi_0 \in \boldsymbol{x_0}$ and $\inf(\boldsymbol{I}) \in \boldsymbol{a}$, we know that

$$\sum_{i=0}^{n} \alpha_i \left(\inf(\boldsymbol{I}) - \xi_0\right)^i \in \texttt{ComputeBound}(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{a}).$$

It follows that $\delta(\inf(\boldsymbol{I})) \in \boldsymbol{\Delta_a}$. The same arguments show that $\delta(\xi_0) \in \boldsymbol{\Delta_{x_0}}$ and $\delta(\sup(\boldsymbol{I})) \in \boldsymbol{\Delta_b}$. By definition of $\boldsymbol{\Delta}$, we have $\boldsymbol{\Delta_a} \subseteq \boldsymbol{\Delta}$, $\boldsymbol{\Delta_{x_0}} \subseteq \boldsymbol{\Delta}$ and $\boldsymbol{\Delta_b} \subseteq \boldsymbol{\Delta}$. In conclusion, the three values $\delta(\inf(\boldsymbol{I}))$, $\delta(\xi_0)$ and $\delta(\sup(\boldsymbol{I}))$ belong to $\boldsymbol{\Delta}$.

Since $x \in \boldsymbol{I}$ and $\xi_0 \in \boldsymbol{x_0} \subseteq \boldsymbol{I}$, we either have $x \in [\inf(\boldsymbol{I}),\, \xi_0]$ or $x \in [\xi_0,\, \sup(\boldsymbol{I})]$. Since δ is monotonic on these intervals, we know for sure that $\delta(x)$ lies between $\delta(\inf(\boldsymbol{I}))$ and $\delta(\xi_0)$ or between $\delta(\xi_0)$ and $\delta(\sup(\boldsymbol{I}))$. In all cases, $\delta(x) \in \boldsymbol{\Delta}$.

In order to conclude the proof, it only remains to prove that $0 \in \boldsymbol{\Delta}$. Consider any point $\mu_0 \in \boldsymbol{x_0}$. So, $\sin(\mu_0) \in \boldsymbol{a_0}$. Using this result and the correctness of `ComputeBound`, we get $\sin(\mu_0) \in \texttt{ComputeBound}(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{x_0})$ (we take $\xi_0 = x = \mu_0 \in \boldsymbol{x_0}$, $\alpha_0 = \sin(\mu_0)$ and any value for $\alpha_1, \ldots, \alpha_n$). Since obviously $\sin(\mu_0) \in \texttt{eval}(\sin(x), \boldsymbol{x_0})$, we conclude that $0 \in \boldsymbol{\Delta_{x_0}}$. Hence $0 \in \boldsymbol{\Delta}$, which concludes the proof. □

Example 2.2.3. *Executing an implementation of Algorithm 2.2.3* $\texttt{TMSin}([-0.5, 0.5], [0], 8)$, *with a toy precision of* 5 *digits we obtain the TM below. We denote by* $a = 1234[5|6]$ *the interval* $[12345, 12346]$ *in order have a shortcut representation that allows us to see faster the common digits in the decimal representation of* a.

$$([0],\ [0.9998, 1.0002],\ [0],\ -0.1667[1|3],\ [0],\ 0.833[2|3]\mathrm{e}-2,\ [0], -0.198[4|3]\mathrm{e}-3,\ [-0.16001\mathrm{e}-3, 0.16001\mathrm{e}-3]).$$

We mention for the sake of clarity that the last interval is the enclosure of the remainder $\boldsymbol{\Delta}$. *Using Algorithm 2.1.2, one can obtain a polynomial with numeric coefficients, ex. the polynomial* $\widetilde{T}(x) = x - 0.16672\, x^3 + 0.8332 \cdot 10^{-2}\, x^5 - 0.1984 \cdot 10^{-3}\, x^7$ *and the remainder bound* $[-0.26201 \cdot 10^{-3}, 0.26201 \cdot 10^{-3}]$.

We note that here we chose a toy precision for the computation of the coefficients, and hence the error in the accuracy of the coefficients is similar to the order of the remainder. In practice, the precision for computing the coefficients is much higher, and hence, when passing from a polynomial with "tight interval coefficients" to a numerical one, the error that is added is much smaller compared to the remainder.

Reciprocal function $x \mapsto 1/x$. With Taylor Models, divisions are performed by computing a reciprocal followed by a multiplication. The reciprocal $x \mapsto 1/x$ is handled like any other unary basic function. We give below the pseudo-code for the constructor of the Taylor Model of the reciprocal. It is exactly the same as for $\sin$ except that we compute the derivatives of $x \mapsto 1/x$ instead of the derivatives of $\sin$.

Proof of `TMInv`. Similar to the previous proof, remarking that the ith derivative of $x \mapsto 1/x$ is $x \mapsto \frac{(-1)^i\, i!}{x^{i+1}}$. □

Algorithm: $\texttt{TMInv}(\boldsymbol{I}, \boldsymbol{x_0}, n)$
Input: $\boldsymbol{I}, \boldsymbol{x_0} \subseteq \boldsymbol{I}, n \in \mathbb{N}$
Output: a Taylor Model M of the reciprocal function $x \mapsto 1/x$
for $i \leftarrow 0$ **to** n **do**
  $\boldsymbol{a_i} \leftarrow \texttt{eval}\left(\frac{(-1)^i}{x^{i+1}}, \boldsymbol{x_0}\right)$;
end
$\boldsymbol{\Gamma} \leftarrow \texttt{eval}\left(\frac{(-1)^{n+1}}{x^{n+2}}, \boldsymbol{I}\right)$;
if $(\sup(\boldsymbol{\Gamma}) \leq 0)$ **or** $(\inf(\boldsymbol{\Gamma}) \geq 0)$ **then**
  $\boldsymbol{a} \leftarrow [\inf(\boldsymbol{I})]$;
  $\boldsymbol{b} \leftarrow [\sup(\boldsymbol{I})]$;
  $\boldsymbol{\Delta_a} \leftarrow \texttt{eval}\,(1/x, \boldsymbol{a}) - \texttt{ComputeBound}(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{a})$;
  $\boldsymbol{\Delta_b} \leftarrow \texttt{eval}\,(1/x, \boldsymbol{b}) - \texttt{ComputeBound}(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{b})$;
  $\boldsymbol{\Delta_{x_0}} \leftarrow \texttt{eval}\,(1/x, \boldsymbol{x_0}) - \texttt{ComputeBound}(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{x_0})$;
  $\boldsymbol{\Delta} \leftarrow [\min(\inf(\boldsymbol{\Delta_a}), \inf(\boldsymbol{\Delta_{x_0}}), \inf(\boldsymbol{\Delta_b})),\ \max(\sup(\boldsymbol{\Delta_a}), \sup(\boldsymbol{\Delta_{x_0}}), \sup(\boldsymbol{\Delta_b}))]$;
else
  $\boldsymbol{V} \leftarrow \texttt{eval}\left((x-y)^{n+1}, (\boldsymbol{I}, \boldsymbol{x_0})\right)$;
  $\boldsymbol{\Delta} \leftarrow \boldsymbol{V} \cdot \boldsymbol{\Gamma}$;
end
$M \leftarrow (\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 2.2.4: Computation of a Taylor Model of the reciprocal function

2.2.2 Operations with Taylor Models

Once we have given algorithms for creating TMs for all basic functions, we need to see how to add, multiply and compose Taylor Models. The Taylor Model of an expression is computed by induction on the structure of the expression: we will give more detail on this later. For now on, we only need to keep in mind that the addition and multiplication algorithms take as input two RPA structures that are effectively Taylor Models of two functions f and g, developed around the same expansion *almost point* interval $\boldsymbol{x_0}$ and on the same interval $\boldsymbol{I}$ (this will be ensured by the induction hypothesis), and we shall prove that the addition and multiplication algorithms produce valid Taylor Models for $f + g$ and $f\,g$ respectively.

Addition of Taylor Models. It is roughly speaking based on the observation that given two valid TMs for two functions, we can add the polynomials and the remainders respectively and obtain a new polynomial and a new remainder that form a valid TM for the sum of the two functions.

Example 2.2.4. *Consider* $f = \sin$, $g = \cos$, $\boldsymbol{I} = [-0.5, 0.5]$, $x_0 = 0$, $n = 6$. *We have two TMs for* f *and* g*:*

$$M_f = ([0,0], [0.9979, 1.002], [0,0], [-0.1671, -0.1663], [0,0], [0.8323\mathrm{e}-2, 0.8335\mathrm{e}-2], [-0.1501\mathrm{e}-2, 0.1501\mathrm{e}-2]),$$

$$M_g = ([0.9979, 1.002], [0,0], [-0.5007, -0.4993], [0,0], [0.04161, 0.04169], [0,0]], [-0.21771\mathrm{e}-4, 0]).$$

We can compute a TM for $f+g$, over $\boldsymbol{I}=[-0.5,0.5]$, in $x_0=0$, of order $n=6$, by adding the respective coefficients and remainders of the two models for f and g:

$$M_{f+g}=([0.9979,1.002],[0.9979,1.002],[-0.5007,-0.4993],\ldots,[0.8323\mathrm{e}-2,0.8335\mathrm{e}-2],\\ [-0.1524\mathrm{e}-2,0.1502\mathrm{e}-2]).$$

In this example we also observe that for computing the remainder in the TM for cos *we used Prop. 2.2.1.*

The algorithm and its proof are given below.

Algorithm: `TMAdd`(M_f, M_g, n)
Input: M_f, M_g two Taylor Models corresponding to two functions f and g,
$n \in \mathbb{N}$ the common expansion order of both Taylor Models
Output: a Taylor Model M corresponding to $f+g$
$(\boldsymbol{a_0},\ldots,\boldsymbol{a_n},\boldsymbol{\Delta_f}) \leftarrow M_f$;
$(\boldsymbol{b_0},\ldots,\boldsymbol{b_n},\boldsymbol{\Delta_g}) \leftarrow M_g$;
for $i \leftarrow 0$ **to** n **do**
  $\boldsymbol{c_i} \leftarrow \boldsymbol{a_i}+\boldsymbol{b_i}$;
end
$\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta_f}+\boldsymbol{\Delta_g}$;
$M \leftarrow (\boldsymbol{c_0},\ldots,\boldsymbol{c_n},\boldsymbol{\Delta})$;
return M ;

Algorithm 2.2.5: Addition of Taylor Models

Proof of `TMAdd`. First, we remark that $0 \in \boldsymbol{\Delta_f}$ and $0 \in \boldsymbol{\Delta_g}$ by hypothesis. So $0 \in \boldsymbol{\Delta}$.

Assume that $\xi_0 \in \boldsymbol{x_0}$. Since M_f is a Taylor model of f, we have that

$$\exists \alpha_0 \in \boldsymbol{a_0},\ldots,\alpha_n \in \boldsymbol{a_n}, \forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x)-\sum_{i=0}^{n}\alpha_i\,(x-\xi_0)^i=\delta_f.$$

We consider such values α_i. The same holds for M_g and g with some $\beta_i \in \boldsymbol{b_i}$. We shall show

$$\exists \gamma_0 \in \boldsymbol{c_0},\ldots,\gamma_n \in \boldsymbol{c_n}, \forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta},\ f(x)+g(x)-\sum_{i=0}^{n}\gamma_i\,(x-\xi_0)^i=\delta.$$

We choose $\gamma_i=\alpha_i+\beta_i \in \boldsymbol{c_i}$. Assume that $x \in \boldsymbol{I}$. In consequence, there exists $\delta_f \in \boldsymbol{\Delta_f}$ such that $f(x)-\sum_{i=0}^{n}\alpha_i\,(x-\xi_0)^i=\delta_f$ and accordingly for some $\delta_g \in \boldsymbol{\Delta_g}$. So, we can choose $\delta=\delta_f+\delta_g \in \boldsymbol{\Delta}$ and we have

$$\begin{aligned} f(x)+g(x)-\sum_{i=0}^{n}\gamma_i\,(x-\xi_0)^i &= \left(f(x)-\sum_{i=0}^{n}\alpha_i\,(x-\xi_0)^i\right)+\left(g(x)-\sum_{i=0}^{n}\beta_i\,(x-\xi_0)^i\right)\\ &= \delta_f+\delta_g=\delta. \end{aligned}$$

□

Multiplication of Taylor Models. For multiplication, we give first an informal explanation, and below we give the algorithm and its proof of correctness. Let us consider informally that we have two TMs for two functions f and g, over $\boldsymbol{I}$, in x_0, of order n written as a couple made of a polynomial with real coefficients and a remainder: $(P_f, \boldsymbol{\Delta}_f)$ and $(P_g, \boldsymbol{\Delta}_g)$. Then, $\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta}_f$ and $\delta_g \in \boldsymbol{\Delta}_g$ s.t. $f(x) - P_f(x) = \delta_f$ and $g(x) - P_g(x) = \delta_g$. We have

$$f(x) \cdot g(x) = P_f(x) \cdot P_g(x) + P_g(x) \cdot \delta_f + P_f(x) \cdot \delta_g + \delta_f \cdot \delta_g.$$

We observe that $P_f \cdot P_g$ is a polynomial of degree $2n$. We split it into two parts: the polynomial consisting of the terms that *do not exceed* n, $(P_f \cdot P_g)_{0\ldots n}$ and respectively the upper part $(P_f \cdot P_g)_{n+1\ldots 2n}$, for the terms of the product $P_f \cdot P_g$ whose *order exceeds* n.

Now, a TM for $f \cdot g$ can be obtained by finding an interval bound $\boldsymbol{\Delta}$ for all the terms except $P = (P_f \cdot P_g)_{0\ldots n}$:

$$\boldsymbol{\Delta} = \boldsymbol{B}((P_f \cdot P_g)_{n+1\ldots 2n}) + \boldsymbol{B}(P_g) \cdot \boldsymbol{\Delta}_f + \boldsymbol{B}(P_f) \cdot \boldsymbol{\Delta}_g + \boldsymbol{\Delta}_f \cdot \boldsymbol{\Delta}_g,$$

where we denote by $\boldsymbol{B}(h)$ a valid interval bound for h over the interval $\boldsymbol{I}$ under consideration.

```
1  Algorithm: TMMul(M_f, M_g, I, x_0, n)
   Input: M_f, M_g two Taylor Models corresponding to two functions f and g,
   I the interval of both Taylor Models,
   x_0 the expansion interval for both Taylor Model,
   n ∈ ℕ the order of both Taylor Models
   Output: a Taylor Model M corresponding to f · g
2  (a_0, ..., a_n, Δ_f) ← M_f;
3  (b_0, ..., b_n, Δ_g) ← M_g;
4  for k ← 0 to 2n do
5  |  c_k ← [0; 0];
6  end
7  for i ← 0 to n do
8  |  for j ← 0 to n do
9  |  |  c_{i+j} ← c_{i+j} + a_i · b_j;
10 |  end
11 end
12 for k ← 0 to n do
13 |  d_k ← [0; 0];
14 end
15 for k ← n + 1 to 2n do
16 |  d_k ← c_k;
17 end
18 B ← ComputeBound(d_0, ..., d_{2n}, x_0, I);
19 B_f ← ComputeBound(a_0, ..., a_n, x_0, I);
20 B_g ← ComputeBound(b_0, ..., b_n, x_0, I);
21 Δ ← B + (Δ_f · B_g) + (Δ_g · B_f) + (Δ_f · Δ_g);
22 M ← (c_0, ..., c_n, Δ);
23 return M ;
```

Algorithm 2.2.6: Multiplication of Taylor Models

Proof of `TMMul`. We assume the M_f and M_g are Taylor Models of two functions f and g, around the same expansion interval $\boldsymbol{x_0}$ over the same interval $\boldsymbol{I}$. We shall prove that `TMMul` returns a Taylor Model of fg around $\boldsymbol{x_0}$ over $\boldsymbol{I}$.

By hypothesis, $0 \in \boldsymbol{\Delta_f}$ and $0 \in \boldsymbol{\Delta_g}$. Hence $0 \in \boldsymbol{\Delta_f} \cdot \boldsymbol{B_g}$ and $0 \in \boldsymbol{\Delta_g} \cdot \boldsymbol{B_f}$ and $0 \in \boldsymbol{\Delta_f} \cdot \boldsymbol{\Delta_g}$.

In order to prove that $0 \in \boldsymbol{B}$, we choose an arbitrary value μ_0 in $\boldsymbol{x_0}$; since $\boldsymbol{x_0} \subseteq \boldsymbol{I}$, we can take $x = \xi_0 = \mu_0$ and use the correctness of `ComputeBound`. We get that $\forall \delta_0 \in \boldsymbol{d_0}, \delta_0 \in \boldsymbol{B}$, and we have that $\boldsymbol{d_0} = [0]$. We conclude that $0 \in \boldsymbol{\Delta}$.

Assume that $\xi_0 \in \boldsymbol{x_0}$. Since M_f is a Taylor model of f, we have $\alpha_i \in \boldsymbol{a_i}$ $(i = 0, \ldots, n)$ such that

$$\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) - \sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i = \delta_f.$$

The same holds for M_g and g with $\beta_i \in \boldsymbol{b_i}$. We have

$$\left(\sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i\right) \cdot \left(\sum_{i=0}^{n} \beta_i \left(x - \xi_0\right)^i\right) = \sum_{i=0}^{n} \sum_{j=0}^{n} \alpha_i \, \beta_j \left(x - \xi_0\right)^{i+j} = \sum_{k=0}^{2n} \gamma_k \left(x - \xi_0\right)^k.$$

Clearly, γ_k is the sum of all the $\alpha_i \, \beta_j$ such that $i \in \{0, \ldots, n\}$, $j \in \{0, \ldots, n\}$ and $i + j = k$. Hence we have $\gamma_k \in \boldsymbol{c_k}$ for all k. Assume now that $x \in \boldsymbol{I}$. We have values δ_f and δ_g such that

$$\begin{aligned}
f(x)\, g(x) &= \left(\sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i + \delta_f\right) \cdot \left(\sum_{i=0}^{n} \beta_i \left(x - \xi_0\right)^i + \delta_g\right) \\
&= \sum_{i=0}^{n} \gamma_i \left(x - \xi_0\right)^i \\
&+ \left(\underbrace{\sum_{i=0}^{n} 0 \left(x - \xi_0\right)^i + \sum_{i=n+1}^{2n} \gamma_i \left(x - \xi_0\right)^i}_{\in \boldsymbol{B}}\right. \\
&\quad \left. + \delta_f \underbrace{\left(\sum_{i=0}^{n} \beta_i \left(x - \xi_0\right)^i\right)}_{\in \boldsymbol{B_g}} + \delta_g \underbrace{\left(\sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i\right)}_{\in \boldsymbol{B_f}} + \delta_f \, \delta_g\right).
\end{aligned}$$

The inclusions in $\boldsymbol{B}$, $\boldsymbol{B_f}$ and $\boldsymbol{B_g}$ are given by the correctness of `ComputeBound`. The conclusion follows. □

Composition of Taylor Models. We now describe the algorithm used for the composition of functions. Suppose that we want to compute a Taylor Model for $g \circ f$: the algorithm takes an RPA structure M_f (supposed being a valid Taylor Model of f around $\boldsymbol{x_0}$ over $\boldsymbol{I}$, by the induction hypothesis) and it takes a symbol representing the function g. The first thing to do is to construct a Taylor Model for g. This is is performed using the algorithms of the basic functions that we already described above, while paying good attention that the parameters used for constructing this Taylor Model are not $\boldsymbol{I}$ and $\boldsymbol{x_0}$ anymore but their image through function f. Then, we mainly see the composition of M_g and M_f as the composition of two polynomials: this is performed by means of `PolynomialEvaluationOfTM`. This algorithm is described and proved below.

```
1 Algorithm: PolynomialEvaluationOfTM(b_0, ..., b_n, M_f, I, x_0, n)
  Input: b_0, ..., b_n tight intervals, I an interval,
  M_f a Taylor Model of a function f developed around a tight interval x_0 over I
  A tight interval expansion x_0,
  n ∈ ℕ
  Output: an RPA structure M that is a Taylor Model of x ↦ Σ_{i=0}^{n} β_i f(x)^i around x_0 over I, for
          any (β_0, ..., β_n) such that β_i ∈ b_i for each i.
2 M ← ([0;0], ..., [0;0], [0;0]) ;
3 for i ← n downto 0 do
4 |   M ← TMMul(M, M_f, I, x_0, n);
5 |   M ← TMAdd(M, TMConst(b_i, n), n) ;
6 end
7 return M ;
```

Algorithm 2.2.7: Composition of a polynomial with a Taylor Model

Proof of `PolynomialEvaluationOfTM`. By hypothesis, M_f is a Taylor Model of a function f around an interval $\boldsymbol{x_0}$ over $\boldsymbol{I}$. We prove that, for any values $\beta_i \in \boldsymbol{b_i}$ ($i \in \{0, \ldots, n\}$), the returned structure M is a Taylor Model of the function $x \mapsto \sum_{i=0}^{n} \beta_i f(x)^i$ around $\boldsymbol{x_0}$ over $\boldsymbol{I}$.

Let $\boldsymbol{I}$, $\boldsymbol{x_0}$ and values β_i as above. The proof is the same as the proof of correctness of the algorithm `ComputeBound` (see Algorithm 2.1.1 on page 48). The loop invariant here is the following: at the end of each iteration of the loop, M is a Taylor Model of the function $x \mapsto \sum_{k=i}^{n} \beta_k f(x)^k$ around $\boldsymbol{x_0}$ over $\boldsymbol{I}$. Otherwise, the proof is the same and is based on the correctness of `TMConst`, `TMAdd` and `TMMul`. □

Remark 2.2.5. *Any other polynomial evaluation algorithm could be used, provided that it only uses additions and multiplications. The proof would then exactly follow the proof of correctness of the polynomial evaluation algorithm.*

Proof of `TMComp`. Assume that $\xi_0 \in \boldsymbol{x_0}$. Since M_f is a Taylor model of f, we have values $\alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}$ such that

$$\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) - \sum_{i=0}^{n} \alpha_i (x - \xi_0)^i = \delta_f. \tag{2.1}$$

As per Remark 2.1.6 (see page 48), $\boldsymbol{a_0} \subseteq \boldsymbol{B_f}$. Moreover, since $0 \in \boldsymbol{\Delta_f}$ (induction hypothesis), we have $\boldsymbol{a_0} \subseteq \boldsymbol{B_f} + \boldsymbol{\Delta_f}$. This ensures that the call to `TMSin`, `TMInv`, etc., is correct and M_g is hence a Taylor Model of g around $\boldsymbol{a_0}$ over $\boldsymbol{B_f} + \boldsymbol{\Delta_f}$. Since $\alpha_0 \in \boldsymbol{a_0}$, we have values $\beta_0 \in \boldsymbol{b_0}, \ldots, \beta_n \in \boldsymbol{b_n}$ such that

$$\forall y \in \boldsymbol{B_f} + \boldsymbol{\Delta_f}, \exists \delta_g \in \boldsymbol{\Delta_g},\ g(y) - \sum_{i=0}^{n} \beta_i (y - \alpha_0)^i = \delta_g. \tag{2.2}$$

Now, as per Equation (2.1), and the correctness of `ComputeBound`, we have

$$\forall x \in \boldsymbol{I}, f(x) \in \boldsymbol{B_f} + \boldsymbol{\Delta_f}.$$

```
1  Algorithm: TMComp(I, x_0, M_f, g, n)
   Input: I an interval,
   x_0 ⊆ I an expansion interval,
   M_f a Taylor Model corresponding to a function f developed in x_0 over I,
   g : ℝ → ℝ a base function,
   n ∈ ℕ
   Output: a Taylor Model M corresponding to g ∘ f developed in x_0 over I
2  (a_0, ..., a_n, Δ_f) ← M_f;
3  B_f ← ComputeBound(a_0, ..., a_n, x_0, I);
4  switch g do
5      case sin: M_g ← TMSin(a_0, B_f + Δ_f, n);
6      case x ↦ 1/x: M_g ← TMInv(a_0, B_f + Δ_f, n);
7      ⋮
8  endsw
9  (b_0, ..., b_n, Δ_g) ← M_g;
10 M_1 ← ([0,0], a_1, ..., a_n, Δ_f) ;
11 (c_0, ..., c_n, Δ) ← PolynomialEvaluationOfTM(b_0, ..., b_n, M_1, I, x_0, n);
12 M ← (c_0, ..., c_n, Δ + Δ_g);
13 return M ;
```

Algorithm 2.2.8: Composition of a Taylor Model with a Function

Combining this result with Equation (2.2), we get

$$\forall x \in \boldsymbol{I}, \exists \delta_g \in \boldsymbol{\Delta_g},\ (g \circ f)(x) - \sum_{i=0}^{n} \beta_i \left(f(x) - \alpha_0\right)^i = \delta_g. \tag{2.3}$$

We now prove that M_1 is a Taylor Model of $x \mapsto f(x) - \alpha_0$ around $[\xi_0,\, \xi_0]$ over $\boldsymbol{I}$. Whatever the expansion point $\mu_0 \in [\xi_0; \xi_0]$, we trivially see that $\mu_0 = \xi_0$. Taking $\alpha'_0 = 0 \in [0]$ and, for $i \geq 1$, $\alpha'_i = \alpha_i \in \boldsymbol{a_i}$, we get, with Equation (2.1),

$$\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) - \left(\alpha_0 + \sum_{i=1}^{n} \alpha_i \left(x - \xi_0\right)^i\right) = \delta_f.$$

Hence

$$\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ \left(f(x) - \alpha_0\right) - \sum_{i=0}^{n} \alpha'_i \left(x - \mu_0\right)^i = \delta_f.$$

So, M_1 is a Taylor Model of $f - \alpha_0$ around $[\xi_0; \xi_0]$ over $\boldsymbol{I}$.

By correction of `PolynomialEvaluationOfTM` (and since $[\xi_0,\, \xi_0] \subseteq \boldsymbol{x_0}$, and all the β_i belong to the corresponding $\boldsymbol{b_i}$), we get that $(\boldsymbol{c_0},\, \ldots,\, \boldsymbol{c_n},\, \boldsymbol{\Delta})$ is a Taylor Model of $x \mapsto \sum_{i=0}^{n} \beta_i \left(f(x) - \alpha_0\right)^i$ around $[\xi_0,\, \xi_0]$ over $\boldsymbol{I}$. Hence we have values $\gamma_i \in \boldsymbol{c_i}$ ($i \in \{0, \ldots, n\}$) such that

$$\forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta},\ \left(\sum_{i=0}^{n} \beta_i \left(f(x) - \alpha_0\right)^i\right) - \left(\sum_{i=0}^{n} \gamma_i \left(x - \xi_0\right)^i\right) = \delta.$$

Combining this last result with Equation (2.3), we get

$$\forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta} + \boldsymbol{\Delta_g},\ (g \circ f)(x) - \left(\sum_{i=0}^{n} \gamma_i \left(x - \xi_0\right)^i\right) = \delta,$$

which is the property that we wanted to prove.

Finally, we remark that $0 \in \boldsymbol{\Delta} + \boldsymbol{\Delta_g}$. Indeed $0 \in \boldsymbol{\Delta_g}$ and $0 \in \boldsymbol{\Delta}$ by correctness of `TMSin`, `TMInv`, etc. and `PolynomialEvaluationOfTM`, hence the result. □

Division of Taylor Models. Computing a Taylor Model for $\dfrac{f}{g}$ reduces to computing $f \cdot \left(\dfrac{1}{x} \circ g\right)$. This implies multiplication and composition of Taylor Models as well as computing a Taylor Model for the basic function $x \mapsto 1/x$, which was already explained before in this section. The algorithm is given below.

1 **Algorithm:** `TMBaseDiv`$(M_f, M_g, \boldsymbol{I}, \boldsymbol{x_0}, n)$
Input: M_f, M_g two Taylor Models corresponding to two functions f and g,
$\boldsymbol{I}$ an interval,
$\boldsymbol{x_0} \subseteq \boldsymbol{I}$ an expansion interval,
$n \in \mathbb{N}$
Output: a Taylor Model M corresponding to f/g
2 $M \leftarrow$ `TMMul`$(M_f,$ `TMComp`$(\boldsymbol{I}, \boldsymbol{x_0}, M_g, x \mapsto 1/x, n), \boldsymbol{I}, \boldsymbol{x_0}, n)$;
3 **return** M ;

Algorithm 2.2.9: Basic Division of Taylor Models

Proof of `TMBaseDiv`. Trivial by the correction of `TMMul` and `TMComp` and the fact that f/g can be written $f \cdot 1/g$. □

Finally, we can write the complete algorithm that computes a Taylor Model for any expression, by induction on this expression.

Computing Taylor Models for any function given by an expression

Proof of `TM`. We use structural induction on the expression tree for function h for the proof. In all cases when the algorithm calls one of `TMConst`, `TMVar`, `TMAdd`, `TMMul`,`TMDiv` and `TMComp`, the correction is trivial by correction of the called sub-algorithm. □

We will give in Chapter 3 examples of application of Taylor Models.

2.3 The problem of removable discontinuities – the need for Taylor Models with relative remainder

Suppose that the function f for which we want to compute a Taylor Model on an interval $\boldsymbol{I}$, is given by an expression where a division occurs $f = \frac{u}{v}$ and both u and v vanish on $\boldsymbol{I}$ at some points $z_i \in \boldsymbol{I}$, points called removable discontinuities of f. Mathematically speaking, this is not a problem for the function f. It may stay defined – and bounded – by *continuity*. As we will see in Chapter 3, the matter is not purely theoretical but quite common in practice.

Example 2.3.1. *Let us consider the function* $\varepsilon(x) = \dfrac{x - \frac{1}{6}x^3}{\sin(x)} - 1$, *on a tight interval around zero, say* $\boldsymbol{I} = \left[-\frac{\pi}{64}; \frac{\pi}{64}\right]$. *This function is a relative error function which quantifies the approximation error between* $\sin$ *and the approximation polynomial* $p(x) = x - \frac{1}{6}x^3$, *which is a quite common approximation for* $\sin$. *Both* p *and* $\sin$ *vanish at* $z_0 = 0$, *but the approximation error stays bounded.*

```
1  Algorithm: TM(I, x_0, h, n)
   Input: I an interval,
   x_0 ⊆ I an expansion interval,
   h the expression of a function,
   n ∈ ℕ an expansion order
   Output: a Taylor Model M corresponding to h
2  switch h do
3      case h = x ↦ c: M ← TMConst(c, I, x_0, n);
4      case h = x ↦ x: M ← TMVar(I, x_0, n);
5      case h = f + g:
6          M_f ← TM(n, I, x_0, f, n);
7          M_g ← TM(n, I, x_0, g, n);
8          M ← TMAdd(M_f, M_g, n);
9      endsw
10     case h = f · g:
11         M_f ← TM(n, I, x_0, f, n);
12         M_g ← TM(n, I, x_0, g, n);
13         M ← TMMul(M_f, M_g, I, x_0, n);
14     endsw
15     case h = f/g: M ← TMBaseDiv(n, I, x_0, f, g, n);
16     case h = g ∘ f:
17         M_f ← TM(n, I, x_0, f, n);
18         M ← TMComp(I, x_0, M_f, g, n);
19     endsw
20 endsw
21 return M ;
```

Algorithm 2.2.10: Computation of Taylor Models

The technique of Taylor models, as usually described, is not able to handle such functions f. The usual way of performing a division u/v is to compute a Taylor model for v, compute its inverse, and then multiply by a Taylor model of u. The inversion fails if v vanishes in the interval (more precisely, if extended division is available, it does not fail, but the computed remainder bound is infinite). We propose an adaptation of Taylor models that allows for performing such a division.

Consider an expression of the form u/v. We suppose that a numerical heuristic has already found a point z_0 where u and v seem to vanish simultaneously. If there are several distinct zeros of v in the interval, we split it into several ones. So, we will assume that z_0 is presumably the unique zero of v in the interval. There are two important remarks:

- We additionally assume that z_0 is exactly representable as a floating-point number. This hypothesis may seem restrictive at first sight. However this is usually the case in the practice of floating-point code development. More on, it is hopeless to handle non-representable singularities without introducing symbolic methods;
- The numerical procedure that found z_0 might have missed a zero of v. This is not a problem: in this case, our variant of Taylor models will simply return an infinite remainder bound the same way as the classical Taylor models do. We do not pretend to have a technique that computes models for every possible function u/v: we only try to improve the classical algorithms in order to handle the most common practical cases. In all other cases, our algorithm is permitted to return useless (though correct) infinite bounds.

Our method is based on what everyone would do manually in such a situation: develop both u and v in Taylor series with center z_0 and factor out the leading part $(z-z_0)^{k_0}$ in both expressions. Let us take an example: $v(x) = \log(1+x)$ vanishes at $z_0 = 0$ and its Taylor series with center z_0 is

$$\log(1+x) = 0 + \sum_{i=1}^{+\infty} (-1)^{i+1} \frac{(x-z_0)^i}{i}.$$

Note that the constant term is exactly 0: this is a direct consequence of the fact that $\log(1+x)$ vanishes at z_0. More generally, if z_0 is a zero of order k_0 of a function v, the first k_0 coefficients of the Taylor series of v are exactly zero. If we take for instance $u(x) = \sin(x)$, we have

$$\sin(x) = 0 + \sum_{i=0}^{+\infty} \frac{(-1)^i}{(2i+1)!} (x-z_0)^{2i+1}.$$

So, we can factor out $(x - z_0)$ both from the denominator and the numerator and hence obtain

$$\frac{\sin(x)}{\log(1+x)} = \frac{\displaystyle\sum_{i=0}^{+\infty} \frac{(-1)^i}{(2i+1)!} (x-z_0)^{2i}}{\displaystyle\sum_{i=1}^{+\infty} \frac{(-1)^{i+1}}{i} (x-z_0)^{i-1}}.$$

The denominator of this division does not vanish anymore, so this 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 tight 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 tight 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 at 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

$$\frac{T_u + \boldsymbol{\Delta_u}}{T_v + \boldsymbol{\Delta_v}}. \tag{2.4}$$

Hence, even if both T_u and T_v have a common factor $(x - z_0)^{k_0}$, we cannot divide both the numerator and denominator by $(x - z_0)^{k_0}$ because the division will not simplify in the remainder bounds.

The solution to this problem is easy: 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 $\boldsymbol{\Delta_u}$ of $R_u(\boldsymbol{I})$ through the computation. Instead, we can propagate an enclosure $\boldsymbol{\Delta_u}$ of $\widetilde{R_u}(\boldsymbol{I})$. In this case, equation (2.4) becomes

$$\frac{T_u + (x - z_0)^{n+1}\, \boldsymbol{\Delta_u}}{T_v + (x - z_0)^{n+1}\, \boldsymbol{\Delta_v}},$$

and it becomes possible to factor out the term $(x - z_0)^{k_0}$ from the numerator and the denominator. This led us to define some modified Taylor models that we call *Taylor Models with relative remainder*. Their formal definition is given below. We note that in this definition, z_0 itself is replaced by a tight 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 $[z_0,\, z_0]$. Furthermore, we remark that similar to the definition of Taylor Models with absolute remainder, we suppose implicit the fact that Taylor approximations are used for obtaining this type of RPA.

Definition 2.3.2 (Taylor Model with relative remainder (TMRelative))**.**
Let $M = (\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta})$ be an RPA structure. Let $f : \mathbb{R} \to \mathbb{R}$ be a function, $\boldsymbol{z_0} \subseteq \mathbb{R}$ be a tight interval around an expansion point (usually a point-interval) and $\boldsymbol{I}$ be an interval.
We say that M is a Taylor Model with relative remainder of f at $\boldsymbol{z_0}$ on $\boldsymbol{I}$ when $\boldsymbol{z_0} \subseteq \boldsymbol{I}$ and

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

Using this definition, we can prove that effectively, the true coefficients of the Taylor polynomial lie inside the corresponding intervals $\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ and in this way, a TMRelative "contains" the Taylor expansion. This justifies in a way the use ot the name of "Taylor".

Lemma 2.3.3. *Let $\boldsymbol{I}$ be an interval. Let f be a function that is at least $n+1$ times differentiable on $\boldsymbol{I}$. Let $\boldsymbol{x_0} \subseteq \boldsymbol{I}$ be an expansion interval. Let $M = (\boldsymbol{a_0},\, \ldots,\, \boldsymbol{a_n}, \boldsymbol{\Delta})$ be a TMRelative of f at $\boldsymbol{x_0}$ on $\boldsymbol{I}$. We have:*

$$\forall \xi_0 \in \boldsymbol{x_0},\, \forall i \in \{0, \ldots, n\},\, \frac{f^{(i)}(\xi_0)}{i!} \in \boldsymbol{a_i}.$$

Proof. We choose $\xi_0 \in \boldsymbol{x_0}$. Since M is a TMRelative of f at $\boldsymbol{x_0}$ on $\boldsymbol{I}$, there exist $\alpha_0 \in \boldsymbol{a_0},\, \ldots,\, \alpha_n \in \boldsymbol{a_n}$ such that

$$\forall x \in \boldsymbol{I},\, \exists \delta \in \boldsymbol{\Delta},\, f(x) = \sum_{i=0}^{n} \alpha_i\, (x - \xi_0)^i + \delta\, (x - \xi_0)^{n+1}. \tag{2.5}$$

We shall prove by recurrence over i that for all $i \in \{0, \ldots, n\}$, $\frac{f^{(i)}(\xi_0)}{i!} = \alpha_i$.

For $i = 0$, taking $x = \xi_0$ in Equation 2.5, we get that $f(\xi_0) = \alpha_0$.

Assume now that the property holds for $i \in \{0, \ldots, k\}$ with $k \leq n-1$. Hence equation (2.5) can be rewritten as

$$\forall x \in \boldsymbol{I},\ \exists \delta \in \boldsymbol{\Delta},\ f(x) = \sum_{i=0}^{k} \frac{f^{(i)}(\xi_0)}{i!} (x - \xi_0)^i + \sum_{i=k+1}^{n} \alpha_i (x - \xi_0)^i + \delta (x - \xi_0)^{n+1}.$$

This yields

$$\forall x \in \boldsymbol{I}, x \neq \xi_0,\ \exists \delta \in \boldsymbol{\Delta},\ \frac{f(x) - \sum_{i=0}^{k} \frac{f^{(i)}(\xi_0)}{i!} (x - \xi_0)^i}{(x - \xi_0)^{k+1}} = \sum_{i=0}^{n-k-1} \alpha_{i+k+1} (x - \xi_0)^i + \delta (x - \xi_0)^{n-k}.$$

With x tending to ξ_0, the right-hand side of the equality tends to α_{k+1}. Expanding f into Taylor series at ξ_0, we have

$$\forall x \in \boldsymbol{I},\ f(x) = \sum_{i=0}^{k+1} \frac{f^{(i)}(\xi_0)}{i!} (x - \xi_0)^i + \sum_{i=k+2}^{\infty} \frac{f^{(i)}(\xi_0)}{i!} (x - \xi_0)^i.$$

Thus the left-hand side of the previous equation reduces to

$$\frac{f^{(k+1)}(\xi_0)}{(k+1)!} + \sum_{i=k+2}^{\infty} \frac{f^{(i)}(\xi_0)}{i!} (x - \xi_0)^{i-k-1},$$

which tends to $f^{(k+1)}(\xi_0)/(k+1)!$ with x tending to ξ_0. This finishes the proof. □

Remark 2.3.4. *It is not necessary to consider the expansion interval $\boldsymbol{x_0}$ as a tight, pseudo-point interval merely capturing our incapability to exactly represent the expansion point. In fact, $\boldsymbol{x_0}$ can be an interval as wide as $\boldsymbol{I}$ on which the TMRelative is valid. Such a TMRelative developed in such a wide interval $\boldsymbol{x_0}$ could be seen as capturing any Taylor expansion of the respective function at any expansion point and includes the remainder bound.*

More importantly, as per Lemma 2.3.3, if $M = (\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta})$ is a Taylor Model of f developed at $\boldsymbol{x_0} = \boldsymbol{I}$ on $\boldsymbol{I}$, we have for all $i \in \{0, \ldots, n\}$:

$$\frac{f^{(i)}(\boldsymbol{I})}{i!} \subseteq \boldsymbol{a_i}.$$

This means we can bound the derivatives of a function using Taylor Models.

In what follows we give algorithms for computing Taylor Models with relative remainders for basic functions, which are very similar to those presented in Section 2.2.1.

2.3.1 Taylor Models with relative remainders for basic functions

Constant function

Proof of Algorithm 2.3.1. Assume that $\xi_0 \in \boldsymbol{x_0}$ and $\alpha_0 = c \in \boldsymbol{a_0}$ and $\alpha_1 = \cdots = \alpha_n = 0$. Assume that $x \in \boldsymbol{I}$. We choose $\delta = 0 \in \boldsymbol{\Delta} = [0, 0]$. The property trivially holds. □

```
1 Algorithm: TMConstRelRem(c, I, x_0, n)
  Input: c ∈ ℝ, I, x_0 ⊆ I, n ∈ ℕ
  Output: a TMRelative M of the constant function c
2 a_0 ← [c] ;
3 a_1 ... a_n ← [0, 0] ;
4 Δ ← [0, 0] ;
5 M ← (a_0, ..., a_n, Δ);
6 return M ;
```

Algorithm 2.3.1: Computation of a TMRelative of a constant function

```
1 Algorithm: TMVarRelRem(I, x_0, n)
  Input: I, x_0 ⊆ I, n ∈ ℕ
  Output: a TMRelative M of the identity function x ↦ x
2 a_0 ← x_0 ;
3 if n = 0 then
4 |   Δ ← [1, 1] ;
5 else
6 |   a_1 ← [1, 1] ;
7 |   a_2 ... a_n ← [0, 0] ;
8 |   Δ ← [0, 0] ;
9 end
10 M ← (a_0, ..., a_n, Δ);
11 return M ;
```

Algorithm 2.3.2: Computation of a TMRelative of the identity function

Identity function

Proof of Algorithm 2.3.2. Assume that $\xi_0 \in \boldsymbol{x_0}$ and $\alpha_0 = \xi_0 \in \boldsymbol{a_0}$. If $n = 0$, the property trivially holds with $\delta = 1$. Otherwise we choose $\alpha_1 = 1$ and $\alpha_2 = \cdots = \alpha_n = 0$. Assume that $x \in \boldsymbol{I}$. We choose $\delta = 0 \in \boldsymbol{\Delta} = [0, 0]$. The property trivially holds. □

Computing the coefficients and the remainder for basic functions.

Similarly to the case of Taylor Models with *absolute* remainder, using the Taylor-Lagrange formula for bounding the remainder gives sometimes an important overestimation for the true error bound, so, we can adapt Lemma 5.12 of [176] for Taylor Models with *relative* remainder also. We recall without proof the following proposition which can be found in most calculus schoolbooks.

Proposition 2.3.5 (Integral form of the Taylor remainder)**.** *Let $n \in \mathbb{N}$. Let f be an $n+1$ times continuously differentiable function on an interval $\boldsymbol{I}$ and $\xi_0 \in \boldsymbol{I}$. Then*

$$f(x) = \underbrace{\sum_{i=0}^{n} \frac{f^{(i)}(\xi_0)}{i!} \cdot (x - \xi_0)^i}_{T(x)} + \underbrace{\int_{\xi_0}^{x} \frac{f^{(n+1)}(t)}{n!}(x-t)^n \mathrm{d}t}_{R_n(x)}. \tag{2.6}$$

Remark 2.3.6. *In the previous proposition, with the change of variable $t = \xi_0 + u(x - \xi_0)$, $u \in [0, 1]$, we have:*

$$R_n(x) = \frac{(x-\xi_0)^{n+1}}{n!} \int_0^1 f^{(n+1)}(x_0 + u(x - \xi_0)) \cdot (1-u)^n \mathrm{d}u. \tag{2.7}$$

We denote:

$$I_n(x) = \frac{1}{n!} \int_0^1 f^{(n+1)}(\xi_0 + u(x - \xi_0)) \cdot (1-u)^n \mathrm{d}u. \tag{2.8}$$

Proposition 2.3.7 (Adaptation of Lemma 5.12 of [176] for TMRelative)**.** *Let f be a function defined on an interval $\boldsymbol{I} = [a, b]$; let $\xi_0 \in \boldsymbol{I}$ and let $n \in \mathbb{N}$. If $f^{(n+1)}$ is increasing (resp. decreasing) on I, then I_n is increasing (resp. decreasing) on $\boldsymbol{I}$.*

Proof. We prove that if $f^{(n+1)}$ is increasing (resp. decreasing) on $\boldsymbol{I}$, then $I_n(x)$ is also increasing (resp. decreasing) on $\boldsymbol{I}$. Assume that $f^{(n+1)}$ is increasing. Let $x, y \in \boldsymbol{I}$, $x \leq y$, we notice that

$$I_n(y) - I_n(x) = \frac{1}{n!} \int_0^1 \left(f^{(n+1)}(\xi_0 + u(y - \xi_0)) - f^{(n+1)}(\xi_0 + u(x - \xi_0)) \right) (1-u)^n \mathrm{d}u. \tag{2.9}$$

Since $u \geq 0$, we have $\xi_0 + u(y - \xi_0) \geq x_0 + u(x - \xi_0)$. Since $f^{(n+1)}$ is increasing, it follows that the integrand is nonnegative, which implies $I_n(y) \geq I_n(x)$. □

We can use the above proposition for bounding the remainder of basic functions when we construct their Taylor Model with relative remainder. Namely, we evaluate $f^{(n+2)}(\boldsymbol{I})$ by interval arithmetic: if the resulting interval contains numbers of different signs, we cannot apply the technique and fall back to the classical bound given by Lagrange formula. If, on the contrary, the resulting interval contains only nonnegative numbers or only nonpositive numbers, then it means that $f^{(n+1)}$ is increasing (resp. decreasing) on $\boldsymbol{I}$ and hence, the hypothesis of Proposition 2.3.7 holds. Hence $I_n(x) = \dfrac{f(x) - T(x)}{(x - \xi_0)^{n+1}}$ is monotonic on $[a, b]$. Hence, $I_n([a, b])$ is the interval with bounds $I_n(a)$ and $I_n(b)$. We conclude that $I_n(\boldsymbol{I})$ is the tightest interval containing $I_n(a)$ and $I_n(b)$.

```
1  Algorithm: TMSinRelRem(I, x_0, n)
   Input: I, x_0 ⊆ I, n ∈ ℕ
   Output: a TMRelative M of the sine function
2  for i ← 0 to n do
3      switch i mod 4 do
4          case 0: a_i ← eval (sin (x) /i!, x_0);
5          case 1: a_i ← eval (cos (x) /i!, x_0);
6          case 2: a_i ← eval (− sin (x) /i!, x_0);
7          case 3: a_i ← eval (− cos (x) /i!, x_0);
8      endsw
9  end
10 switch (n + 2) mod 4 do
11     case 0: Γ ← eval (sin(x)/(n + 2)!, I);
12     case 1: Γ ← eval (cos(x)/(n + 2)!, I);
13     case 2: Γ ← eval (− sin(x)/(n + 2)!, I);
14     case 3: Γ ← eval (− cos(x)/(n + 2)!, I);
15 endsw
16 a ← [inf(I)];
17 b ← [sup(I)];
18 if (a ∩ x_0 = ∅) and (b ∩ x_0 = ∅ ) and ((sup(Γ) ≤ 0) or (inf(Γ) ≥ 0)) then
19     Δ_a ← eval (sin(x), a) − ComputeBound(a_0, ..., a_n, x_0, a);
20     Δ_b ← eval (sin(x), b) − ComputeBound(a_0, ..., a_n, x_0, b);
21     V_a ← eval ((x − y)^{n+1}, (a, x_0));
22     V_b ← eval ((x − y)^{n+1}, (b, x_0));
23     Δ_a ← Δ_a/V_a;
24     Δ_b ← Δ_b/V_b;
25     Δ ← [min(inf(Δ_a), inf(Δ_b)), max(sup(Δ_a), sup(Δ_b))];
26 else
27     switch (n + 1) mod 4 do
28         case 0: Δ ← eval (sin(x)/(n + 1)!, I);
29         case 1: Δ ← eval (cos(x)/(n + 1)!, I);
30         case 2: Δ ← eval (− sin(x)/(n + 1)!, I);
31         case 3: Δ ← eval (− cos(x)/(n + 1)!, I);
32     endsw
33 end
34 M ← (a_0, ..., a_n, Δ);
35 return M ;
```

Algorithm 2.3.3: Computation of a TMRelative of the sine function

The sine function

Proof of Algorithm 2.3.3. Assume that $\xi_0 \in \boldsymbol{x_0}$. For all i we choose $\alpha_i = \sin^{(i)}(\xi_0)/i!$. We know that
- $\sin^{(i)}(\xi_0) = \sin(\xi_0)$ when $i = 0 \mod 4$;
- $\sin^{(i)}(\xi_0) = \cos(\xi_0)$ when $i = 1 \mod 4$;
- $\sin^{(i)}(\xi_0) = -\sin(\xi_0)$ when $i = 2 \mod 4$;
- $\sin^{(i)}(\xi_0) = -\cos(\xi_0)$ when $i = 3 \mod 4$.

Hence, for all i, we have $\alpha_i \in \boldsymbol{a_i}$. Moreover, we remark that, for all $\xi \in \boldsymbol{I}$, $\sin^{(n+2)}(\xi)/(n+2)! \in \boldsymbol{\Gamma}$.

Let us prove first the computation of the remainder $\boldsymbol{\Delta}$ when the execution goes through the "else" branch, i.e. we can not apply Proposition 2.3.7. Let $x \in \boldsymbol{I}$. As per Taylor theorem, there exists ξ strictly between ξ_0 and x such that

$$\sin(x) - \sum_{i=0}^{n} \alpha_i \, (x - \xi_0)^i = (x - \xi_0)^{n+1} \, \frac{\sin^{(n+1)}(\xi)}{(n+1)!}.$$

Since $\xi_0 \in \boldsymbol{x_0} \subseteq \boldsymbol{I}$ and since $x \in \boldsymbol{I}$, we have $\xi \in \boldsymbol{I}$ (convexity of $\boldsymbol{I}$). Hence $\frac{\sin^{(n+1)}(\xi)}{(n+1)!} \in \boldsymbol{\Delta}$. This concludes the proof for the execution of the "else" branch.

If, on the contrary, we have $\sup(\boldsymbol{\Gamma}) \leq 0$ or $\inf(\boldsymbol{\Gamma}) \geq 0$, then suppose for instance that $\sup(\boldsymbol{\Gamma}) \leq 0$ (the other case is handled similarly). Since $\forall x \in \boldsymbol{I}, \sin^{(n+2)}(x)/(n+2)! \in \boldsymbol{\Gamma}$, we have, in particular,

$$\forall x \in \boldsymbol{I}, \, \sin^{(n+2)}(x)/(n+2)! \leq \sup(\boldsymbol{\Gamma}) \leq 0.$$

Hence, the hypothesis of Proposition 2.3.7 holds (since $\sin^{(n+1)}$ is monotonic on $\boldsymbol{I}$). It follows that $x \mapsto \dfrac{\sin(x) - \sum_{i=0}^{n} \alpha_i (x - \xi_0)}{(x - \xi_0)^{n+1}}$ is monotonic on $[\inf(\boldsymbol{I}), \sup(\boldsymbol{I})]$. The supplementary conditions $\boldsymbol{a} \cap \boldsymbol{x_0} = \emptyset$ and $\boldsymbol{b} \cap \boldsymbol{x_0} = \emptyset$ (at line 18 of the algorithm) ensure that the range computed using the division in lines 23 and 24 does not becomes infinite due to the presence of 0 in the denominator. This would be correct, but completely useless for our purposes.

Finally, since `ComputeBound` computes a valid polynomial bound, and since $\alpha_i \in \boldsymbol{a_i}$ for all i, $\xi_0 \in \boldsymbol{x_0}$ and $\inf(\boldsymbol{I}) \in \boldsymbol{a}$, we know that

$$\sum_{i=0}^{n} \alpha_i \, (\inf(\boldsymbol{I}) - \xi_0)^i \in \texttt{ComputeBound}(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{a}).$$

Similar arguments hold for computing the final bound $\boldsymbol{\Delta}$. □

Reciprocal function $x \mapsto 1/x$.

Proof of Algorithm 2.3.4. Like the previous proof, remarking that the ith derivative of $x \mapsto 1/x$ is $x \mapsto \frac{(-1)^i}{x^{i+1}}$. □

2.3.2 Operations with Taylor Models with relative remainders

All the classical operations on Taylor Models (addition, multiplication, composition) translate easily to modified Taylor Models with relative remainder. In what follows we give the algorithms and their proofs of correctness, while keeping in mind that the intuition is similar to Taylor Models with absolute remainder presented in Section 2.2.2.

Algorithm: `TMInvRelRem`$(\boldsymbol{I}, \boldsymbol{x_0}, n)$
Input: $\boldsymbol{I}, \boldsymbol{x_0} \subseteq \boldsymbol{I}, n \in \mathbb{N}$
Output: a TMRelative M of the reciprocal function $x \mapsto 1/x$
for $i \leftarrow 0$ **to** n **do**
  $\boldsymbol{a_i} \leftarrow \texttt{eval}\left(\frac{(-1)^i}{x^{i+1}}, \boldsymbol{x_0}\right)$;
end
$\boldsymbol{\Gamma} \leftarrow \texttt{eval}\left(\frac{(-1)^{(n+2)}}{x^{n+3}}, \boldsymbol{I}\right)$;
$\boldsymbol{a} \leftarrow [\inf(\boldsymbol{I})]$;
$\boldsymbol{b} \leftarrow [\sup(\boldsymbol{I})]$;
if $(\boldsymbol{a} \cap \boldsymbol{x_0} = \emptyset)$ **and** $(\boldsymbol{b} \cap \boldsymbol{x_0} = \emptyset)$ **and** $((\sup(\boldsymbol{\Gamma}) \leq 0)$ **or** $(\inf(\boldsymbol{\Gamma}) \geq 0))$ **then**
  $\boldsymbol{\Delta_a} \leftarrow \texttt{eval}\,(1/x, \boldsymbol{a}) - \texttt{ComputeBound}(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{a})$;
  $\boldsymbol{\Delta_b} \leftarrow \texttt{eval}\,(1/x, \boldsymbol{b}) - \texttt{ComputeBound}(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{b})$;
  $\boldsymbol{V_a} \leftarrow \texttt{eval}\left((x-y)^{n+1}, (\boldsymbol{a}, \boldsymbol{x_0})\right)$;
  $\boldsymbol{V_b} \leftarrow \texttt{eval}\left((x-y)^{n+1}, (\boldsymbol{b}, \boldsymbol{x_0})\right)$;
  $\boldsymbol{\Delta_a} \leftarrow \boldsymbol{\Delta_a}/\boldsymbol{V_a}$;
  $\boldsymbol{\Delta_b} \leftarrow \boldsymbol{\Delta_b}/\boldsymbol{V_b}$;
  $\boldsymbol{\Delta} \leftarrow [\min(\inf(\boldsymbol{\Delta_a}), \inf(\boldsymbol{\Delta_b})), \max(\sup(\boldsymbol{\Delta_a}), \sup(\boldsymbol{\Delta_b}))]$;
else
  $\boldsymbol{\Delta} \leftarrow \texttt{eval}\left(\frac{(-1)^{n+1}}{x^{n+2}}, \boldsymbol{I}\right)$;
end
$M \leftarrow (\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 2.3.4: Computation of a TMRelative of the reciprocal function

Algorithm: `TMAddRelRem`(M_f, M_g, n)
Input: M_f, M_g two TMRelative corresponding to two functions f and g,
$n \in \mathbb{N}$ a expansion order
Output: a TMRelative M corresponding to $f + g$
$(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta_f}) \leftarrow M_f$;
$(\boldsymbol{b_0}, \ldots, \boldsymbol{b_n}, \boldsymbol{\Delta_g}) \leftarrow M_g$;
for $i \leftarrow 0$ **to** n **do**
  $\boldsymbol{c_i} \leftarrow \boldsymbol{a_i} + \boldsymbol{b_i}$;
end
$\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta_f} + \boldsymbol{\Delta_g}$;
$M \leftarrow (\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 2.3.5: Addition of TMRelative

Addition

Proof of Algorithm 2.3.5. Let $\xi_0 \in \boldsymbol{x_0}$. Since M_f is a TMRelative of f, we have that

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

The same holds for M_g and g with $\beta_i \in \boldsymbol{b_i}$. It suffices to choose $\gamma_i = \alpha_i + \beta_i \in \boldsymbol{c_i}$. Then, for all $x \in \boldsymbol{I}$, there exists $\delta = \delta_f + \delta_g \in \boldsymbol{\Delta}$ such that

$$\begin{aligned} f(x) + g(x) - \left(\sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i + \sum_{i=0}^{n} \beta_i \left(x - \xi_0\right)^i \right) &= \left(\delta_f + \delta_g\right) \left(x - \xi_0\right)^{n+1} \\ &= f(x) + g(x) - \sum_{i=0}^{n} \gamma_i \left(x - \xi_0\right)^i. \end{aligned}$$

□

Multiplication

Proof of Algorithm 2.3.6. Assume that $\xi_0 \in \boldsymbol{x_0}$. Since M_f is a TMRelative of f, we have that

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

The same holds for M_g and g with $\beta_i \in \boldsymbol{b_i}$. We have

$$\left(\sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i \right) \cdot \left(\sum_{i=0}^{n} \beta_i \left(x - \xi_0\right)^i \right) = \sum_{i=0}^{2n} \underbrace{\left(\sum_{k=0}^{i} \alpha_k \, \beta_{i-k} \right)}_{\gamma_i} \left(x - \xi_0\right)^i.$$

It is easy to verify that $\gamma_i \in \boldsymbol{c_i}$. Assume that $x \in I$. We have

$$\begin{aligned} f(x)\, g(x) &= \left(\sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i + \delta_f \ \left(x - \xi_0\right)^{n+1} \right) \cdot \left(\sum_{i=0}^{n} \beta_i \left(x - \xi_0\right)^i + \delta_g \ \left(x - \xi_0\right)^{n+1} \right) \\ &= \sum_{i=0}^{n} \gamma_i \left(x - \xi_0\right)^i + \left(x - \xi_0\right)^{n+1} \left(\underbrace{\sum_{i=0}^{n-1} \gamma_{n+1+i} \left(x - \xi_0\right)^i}_{\in \boldsymbol{B}} + \delta_f \underbrace{\left(\sum_{i=0}^{n} \beta_i \left(x - \xi_0\right)^i \right)}_{\in \boldsymbol{B_g}} \right. \\ &+ \left. \delta_g \underbrace{\left(\sum_{i=0}^{n} \alpha_i \left(x - \xi_0\right)^i \right)}_{\in \boldsymbol{B_f}} + \delta_f \, \delta_g \underbrace{\left(x - \xi_0\right)^{n+1}}_{\in \boldsymbol{V}} \right). \end{aligned}$$

The conclusion is trivial. □

Remark 2.3.8. *Let $i, j \in \{0 \ldots n\}$. If we have $\boldsymbol{a_0} = \cdots = \boldsymbol{a_i} = \boldsymbol{b_0} = \cdots = \boldsymbol{b_j} = [0; 0]$, it is easy to remark that*

$$\boldsymbol{c_0} = \cdots = \boldsymbol{c_{i+j}} = [0; 0].$$

In this case, the algorithm does nothing but applying $+$ and $\cdot$ on intervals $[0; 0]$ for the respective $\boldsymbol{c_k}$.

```
1  Algorithm: TMMulRelRem(M_f, M_g, I, x_0, n)
   Input: M_f, M_g two TMRelative corresponding to two functions f and g,
   I an interval,
   x_0 ⊆ I a expansion interval,
   n ∈ ℕ
   Output: a TMRelative M corresponding to f · g
2  (a_0, ..., a_n, Δ_f) ← M_f;
3  (b_0, ..., b_n, Δ_g) ← M_g;
4  for k ← 0 to 2n do
5  |  c_k ← [0, 0];
6  end
7  for i ← 0 to n do
8  |  for j ← 0 to n do
9  |  |  c_{i+j} ← c_{i+j} + a_i · b_j;
10 |  end
11 end
12 B ← ComputeBound(c_{n+1}, ..., c_{2n}, x_0, I);
13 B_f ← ComputeBound(a_0, ..., a_n, x_0, I);
14 B_g ← ComputeBound(b_0, ..., b_n, x_0, I);
15 V ← eval((x − y)^{n+1}, (I, x_0));
16 Δ ← B + (Δ_f · B_g) + (Δ_g · B_f) + (Δ_f · Δ_g · V);
17 M ← (c_0, ..., c_n, Δ);
18 return M ;
```

Algorithm 2.3.6: Multiplication of TMRelative

Composition The composition algorithm described below is similar to that presented in Section 2.2.2. For computing a Taylor Model with relative remainder for $g \circ f$, the algorithm inputs are: a TMRelative structure M_f (supposed being a valid Taylor Model with relative remainder of f around $\boldsymbol{x_0}$ on $\boldsymbol{I}$, by the induction hypothesis) and a function g. The first thing to do is to construct a Taylor Model with relative remainder for g. This is done using the constructors of the basic functions that we already described above, while paying good attention that the parameters used for constructing this Taylor Model are not $\boldsymbol{I}$ and $\boldsymbol{x_0}$ anymore but their image through function f. Then, we mainly see the composition of M_g and M_f as the composition of two polynomials: this is done by means of Algorithm 2.3.7. This algorithm is described and proved below and is similar to `PolynomialEvaluationOfTM`.

1 **Algorithm:** `PolynomialEvaluationOfTMRelRem`($\boldsymbol{b_0}, \ldots, \boldsymbol{b_n}, M_f, \boldsymbol{I}, \boldsymbol{x_0}, n$)
Input: $\boldsymbol{b_0}, \ldots, \boldsymbol{b_n}$ tight intervals, $\boldsymbol{I}$ an interval,
M_f a TMRelative of a function f expanded around a tight interval $\boldsymbol{x_0}$ on $\boldsymbol{I}$,
$n \in \mathbb{N}$
Output: an RPA M that is a TMRelative of $x \mapsto \sum_{i=0}^{n} \beta_i\, f(x)^i$ around $\boldsymbol{x_0}$ on $\boldsymbol{I}$, for any $(\beta_0, \ldots, \beta_n)$ such that $\beta_i \in \boldsymbol{b_i}$ for each i.
2 $M \leftarrow ([0;0], \ldots, [0;0], [0;0])$;
3 **for** $i \leftarrow n$ **downto** 0 **do**
4 $\quad M \leftarrow$ `TMMulRelRem`($M, M_f, \boldsymbol{I}, \boldsymbol{x_0}, n$);
5 $\quad M \leftarrow$ `TMAddRelRem`(M, `TMConstRelRem`($\boldsymbol{b_i}, n$), n) ;
6 **end**
7 **return** M ;

Algorithm 2.3.7: Composition of a polynomial with a TMRelative

Proof of Algorithm 2.3.7. By hypothesis, M_f is a TMRelative of a function f around an interval $\boldsymbol{x_0}$ on $\boldsymbol{I}$. We prove that, for any values $\beta_i \in \boldsymbol{b_i}$ ($i \in \{0, \ldots, n\}$), the returned structure M is a TMRelative of the function $x \mapsto \sum_{i=0}^{n} \beta_i\, f(x)^i$ around $\boldsymbol{x_0}$ on $\boldsymbol{I}$.

Let $\boldsymbol{I}$, $\boldsymbol{x_0}$ and values β_i as above. The proof is the same as the proof of correctness of the algorithm `ComputeBound` (see Algorithm 2.1.1, page 48). The loop invariant here is the following: at the end of each iteration of the loop, M is aTMRelative of the function $x \mapsto \sum_{k=i}^{n} \beta_k\, f(x)^k$ around $\boldsymbol{x_0}$ on $\boldsymbol{I}$. Otherwise, the proof is the same and is based on the correctness of `TMConstRelRem`, `TMAddRelRem` and `TMMulRelRem`.

Remark 2.3.9. *Any other polynomial evaluation algorithm could be used, provided that it only uses additions and multiplications. The proof would then exactly follow the proof of correctness of the polynomial evaluation algorithm.*

□

Proof of Algorithm 2.3.8. Let us prove first that the model M_g obtained at line 11 of the algorithm is a TMRelative of g at $\boldsymbol{a_0}$ on $\boldsymbol{J}$.

Let $\xi_0 \in \boldsymbol{x_0}$. Since M_f is a TMRelative of f, there exist $\alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}$ such that

$$\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) - \sum_{i=0}^{n} \alpha_i\, (x - \xi_0)^i = \delta_f\ (x - \xi_0)^{n+1}\,. \tag{2.10}$$

```
1  Algorithm: TMCompRelRem(I, x_0, M_f, g, n)
   Input: I an interval,
   x_0 ⊆ I an expansion interval,
   M_f a TMRelative corresponding to a function f expanded at x_0 on I,
   g : ℝ → ℝ a basic function,
   n ∈ ℕ
   Output: a TMRelative M corresponding to g ∘ f expanded at x_0 on I
2  (a_0, ..., a_n, Δ_f) ← M_f;
3  B_f ← ComputeBound(a_0, ..., a_n, x_0, I);
4  V ← eval ((x − y)^{n+1}, (I, x_0));
5  J ← B_f + (Δ_f · V);
6  switch g do
7      case sin: M_g ← TMSinRelRem(a_0, J, n);
8      case x ↦ 1/x: M_g ← TMInvRelRem(a_0, J, n);
9      ⋮
10 endsw
11 (b_0, ..., b_n, Δ_g) ← M_g;
12 M_1 ← ([0, 0], a_1, ..., a_n, Δ_f) ;
13 (c_0, ..., c_n, Δ) ← PolynomialEvaluationOfTMRelRem(b_0, ..., b_n, M_1, I, x_0, n);
14 for i ← 1 to n do
15     M_{i+1} ← TMMulRelRem(M_i, M_1, I, x_0, n);
16 end
17 (r_{n+1,0}, ..., r_{n+1,n}, Δ_{n+1}) ← M_{n+1};
18 M ← (c_0, ..., c_n, Δ + Δ_{n+1} · Δ_g);
19 return M ;
```

Algorithm 2.3.8: Composition of a TMRelative with a Function

In particular, taking $x = \xi_0$, $f(\xi_0) = \alpha_0$ because all other terms vanish. Hence $f(\xi_0) \in \boldsymbol{a_0}$. Moreover, from Remark 2.1.6 we have $\boldsymbol{a_0} \subseteq \boldsymbol{B_f}$. Since $0 \in \boldsymbol{V}$, $0 \in (\boldsymbol{\Delta_f} \cdot \boldsymbol{V})$ and thus $\boldsymbol{a_0} \subseteq \boldsymbol{B_f} \subseteq \boldsymbol{J}$. Hence, calling the basic function algorithms `TMSinRelRem`, `TMInvRelRem`, etc. on $(\boldsymbol{a_0}, \boldsymbol{J}, n)$ is valid. By correction of these algorithms $M_g = (\boldsymbol{b_0}, \ldots, \boldsymbol{b_n}, \boldsymbol{\Delta_g})$ is a TMRelative of g at $\boldsymbol{a_0}$ on $\boldsymbol{J}$.

Hence, there exists $\beta_0 \in \boldsymbol{b_0}, \ldots, \beta_n \in \boldsymbol{b_n}$ such that

$$\forall y \in \boldsymbol{J}, \exists \delta_g \in \boldsymbol{\Delta_g},\ g(y) - \sum_{i=0}^{n} \beta_i \left(y - \alpha_0\right)^i = \delta_g \left(y - \alpha_0\right)^{n+1}. \tag{2.11}$$

From Equation (2.10) and the correctness of `ComputeBound`, we have $\forall x \in \boldsymbol{I}, f(x) \in \boldsymbol{J}$. Combining this result with Equation (2.11), we get

$$\forall x \in \boldsymbol{I}, \exists \delta_g \in \boldsymbol{\Delta_g},\ (g \circ f)(x) - \sum_{i=0}^{n} \beta_i \left(f(x) - \alpha_0\right)^i = \delta_g \left(f(x) - \alpha_0\right)^{n+1}. \tag{2.12}$$

Next, the key observation is that M_1 is a TMRelative of $x \mapsto f(x) - \alpha_0$ around $[\xi_0, \xi_0]$ on $\boldsymbol{I}$. To prove this, we use trivially the definition of TMRelative, taking the expansion point $\mu_0 \in [\xi_0; \xi_0]$, $\mu_0 = \xi_0$, $\alpha'_0 = 0 \in [0]$ and, for $i \geq 1$, $\alpha'_i = \alpha_i \in \boldsymbol{a_i}$ and using Equation (2.10),

$$\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ (f(x) - \alpha_0) - \sum_{i=0}^{n} \alpha'_i \left(x - \mu_0\right)^i = \delta_f \left(x - \mu_0\right)^{n+1}.$$

So, M_1 is a TMRelative of $f - \alpha_0$ around $[\xi_0; \xi_0]$ on $\boldsymbol{I}$, and by correction of `PolynomialEvaluationOfTMRelRem` (and since $[\xi_0, \xi_0] \subseteq \boldsymbol{x_0}$, and all the β_i belong to the corresponding $\boldsymbol{b_i}$), we deduce that $(\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}, \boldsymbol{\Delta})$ is a TMRelative of $x \mapsto \sum_{i=0}^{n} \beta_i \left(f(x) - \alpha_0\right)^i$ around $[\xi_0, \xi_0]$ on $\boldsymbol{I}$. Hence we have values $\gamma_i \in \boldsymbol{c_i}$, $i \in \{0, \ldots, n\}$ such that

$$\forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta},\ \left(\sum_{i=0}^{n} \beta_i \left(f(x) - \alpha_0\right)^i\right) - \left(\sum_{i=0}^{n} \gamma_i \left(x - \xi_0\right)^i\right) = \delta \left(x - \xi_0\right)^{n+1}. \tag{2.13}$$

Since M_1 is a TMRelative of $f - \alpha_0$ at $[\xi_0; \xi_0]$ on $\boldsymbol{I}$, by trivial induction using the correction of `TMMulRelRem`, we know that for all $i = 0, \ldots, n$, M_i is a TMRelative of $(f - \alpha_0)^i$ in $[\xi_0; \xi_0]$ on $\boldsymbol{I}$. In particular, for $i = n + 1$, using Remark 2.3.8, the intervals $\boldsymbol{r_{n+1,k}}$, $k = 0, \ldots, n$ computed are all $[0, 0]$. Furthermore, there exist $\rho_{n+1,k} \in \boldsymbol{r_{n+1,k}}$ and $\delta_{n+1} \in \boldsymbol{\Delta_{n+1}}$ such that

$$\forall x \in \boldsymbol{I},\ (f(x) - \alpha_0)^{n+1} - \sum_{k=0}^{n} \rho_{n+1,k} \left(x - \xi_0\right)^k = \delta_{n+1} \left(x - \xi_0\right)^{n+1}.$$

This reduces to:

$$\forall x \in \boldsymbol{I}, \exists \delta_{n+1} \in \boldsymbol{\Delta_{n+1}},\ (f(x) - \alpha_0)^{n+1} = \delta_{n+1} \left(x - \xi_0\right)^{n+1}. \tag{2.14}$$

Combining Equations (2.13), (2.12) and (2.14), we get

$$\forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta} + \boldsymbol{\Delta_g} \cdot \boldsymbol{\Delta_{n+1}},\ (g \circ f)(x) - \left(\sum_{i=0}^{n} \gamma_i \left(x - \xi_0\right)^i\right) = \delta \left(x - \xi_0\right)^{n+1},$$

which is the property that we wanted to prove. □

Division We can now modify the division rule, in order to handle correctly the case when both the numerator and denominator of $\frac{u}{v}$ vanish at z_0. We note that usually the common root of u and v is determined beforehand, using for example a numerical heuristic. We do not discuss numerical root finders here and we suppose that we are already given an expansion point z_0, which we verify *a posteriori* whether it is a root of order (multiplicity) k_0 of both u and v such that we can factor out the term $(x - z_0)^{k_0}$. For that, we proceed in two steps:

- first, we use Algorithm `DetermineRootOrderUpperBound` to find an upper bound $k \geqslant k_0$ for the multiplicity of z_0 as a root of v. This algorithm uses Remark 2.3.4 to compute enclosures of successive derivatives of a given function v, at z_0, until these enclosures do not contain 0. This gives us an upper bound for the order of z_0 as a root of v.
- second, we compute TMRelative of order $n+k$ for u and v in z_0. We then find k_0 such that the first k_0 coefficients of T_u and T_v are all $[0,\ 0]$. We can then factor out the term $(x-z_0)^{k_0}$, which leads to models of order $n + k - k_0 \geqslant n$. We use Algorithm 2.3.10 to reduce the order of the models to n and proceed to "classical" division of Taylor Models, given in Algorithm 2.2.9. All these algorithms are given and proved below. The complete division algorithm is given in Algorithm 2.3.12.

Algorithm: `DetermineRootOrderUpperBound`$(\overline{n}, \boldsymbol{x_0}, f)$
Input: $\overline{n} \in \mathbb{N}$ a maximal internal expansion order,
$\boldsymbol{x_0}$ an expansion interval,
f a function that is at least $\overline{n} + 1$ times continuously differentiable on $\boldsymbol{x_0}$,
Output: k an upper bound for the order of a zero of f at $\boldsymbol{x_0}$
or $\perp$ if the algorithm fails to find such an order.
$n \leftarrow 4$;
while $n \leq \overline{n}$ **do**
  $(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta}) \leftarrow$ `TMRelRem`$(\overline{n}, \boldsymbol{x_0}, \boldsymbol{x_0}, f, n)$;
  $k \leftarrow 0$;
  while $k \leq n \wedge 0 \in \boldsymbol{a_k}$ **do** $k \leftarrow k + 1$;
  if $k \leq n \wedge 0 \notin \boldsymbol{a_k}$ **then return** k;
  $n \leftarrow 2n$;
end
return $\perp$;

Algorithm 2.3.9: Determine an upper bound of the order of a root of a function

Proof of partial correction of `DetermineRootOrderUpperBound`. We have not yet proven the correction of function `TMRelRem` used by `DetermineRootOrderUpperBound` to compute a Taylor Model of the function g at $\boldsymbol{x_0}$ on $\boldsymbol{x_0}$. We shall hence show a mere implication: the algorithm `DetermineRootOrderUpperBound` is correct for functions f on which `TMRelRem` is correct.

Assume that `TMRelRem` is correct. The algorithm is trivially correct if it returns $\perp$. Assume it returns an integer k. So we have a Taylor Model with relative remainder $(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta})$ of the function f which is at least $n + 1 \leq \overline{n} + 1$ times continuously differentiable such that $0 \notin \boldsymbol{a_k}$. As per Lemma 2.3.3, this means that $f^{(k)}(\xi)$ does not vanish for any $\xi \in \boldsymbol{x_0}$. In consequence k is an upper bound of the order of a zero of f in $\boldsymbol{x_0}$. □

Proof of `TMReduceOrder`. Assume that $\xi_0 \in \boldsymbol{x_0}$. Since M_f is a TMRelative of f at $\boldsymbol{x_0}$ on $\boldsymbol{I}$, there

1 **Algorithm:** `TMReduceOrder`$(M_f, \boldsymbol{I}, \boldsymbol{x_0}, n, n')$
Input: M_f a TMRelative corresponding to a function f,
$\boldsymbol{I}$ an interval,
$\boldsymbol{x_0} \subseteq \boldsymbol{I}$ an expansion interval,
$n \in \mathbb{N}$ an expansion order,
$n' \in \mathbb{N}, n' \leqslant n$, a new expansion order
Output: a TMRelative M corresponding to the function f of order n'
2 $(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta_f}) \leftarrow M_f$;
3 $\boldsymbol{B_f} \leftarrow$ `ComputeBound`$(\boldsymbol{a_{n'+1}}, \ldots, \boldsymbol{a_n}, \boldsymbol{x_0}, \boldsymbol{I})$;
4 $\boldsymbol{V} \leftarrow$ `eval` $\left((x-y)^{n-n'}, (\boldsymbol{I}, \boldsymbol{x_0})\right)$;
5 $\boldsymbol{\Delta'_f} \leftarrow \boldsymbol{B_f} + \boldsymbol{\Delta_f} \cdot \boldsymbol{V}$;
6 $M \leftarrow \left(\boldsymbol{a_0}, \ldots, \boldsymbol{a_{n'}}, \boldsymbol{\Delta'_f}\right)$;
7 **return** M ;

Algorithm 2.3.10: Order reduction for TMRelative

exists $\alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}$ such that

$$\forall x \in \boldsymbol{I},\ \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) = \sum_{i=0}^{n'} \alpha_i\ (x-\xi_0)^i + \sum_{i=n'+1}^{n} \alpha_i\ (x-\xi_0)^i + \delta_f\ (x-\xi_0)^{n+1}.$$

Hence we get $\forall x \in \boldsymbol{I},\ \exists \delta_f \in \boldsymbol{\Delta_f}$,

$$f(x) = \sum_{i=0}^{n'} \alpha_i\ (x-\xi_0)^i + (x-\xi_0)^{n'+1} \left(\underbrace{\sum_{i=0}^{n-n'-1} \alpha_{i+n'+1}\ (x-\xi_0)^i}_{\in \boldsymbol{B_f}} + \delta_f\ \underbrace{(x-\xi_0)^{n-n'}}_{\in \boldsymbol{V}} \right).$$

So M is a TMRelative of f of order n' at $\boldsymbol{x_0}$ on $\boldsymbol{I}$. □

We give in the following the "basic division algorithm", i.e. the algorithm similar to the *absolute case*, when we consider that no removable discontinuity is present.

1 **Algorithm:** `TMBaseDiv`$(M_f, M_g, \boldsymbol{I}, \boldsymbol{x_0}, n)$
Input: M_f, M_g two TMRelative corresponding to two functions f and g,
$\boldsymbol{I}$ an interval,
$\boldsymbol{x_0} \subseteq \boldsymbol{I}$ an expansion interval,
$n \in \mathbb{N}$
Output: a TMRelative M corresponding to f/g
2 $M \leftarrow$ `TMMulRelRem`$(M_f,$ `TMCompRelRem`$(\boldsymbol{I}, \boldsymbol{x_0}, M_g, x \mapsto 1/x, n), \boldsymbol{I}, \boldsymbol{x_0}, n)$;
3 **return** M ;

Algorithm 2.3.11: Basic Division of TMRelative

Proof of `TMBaseDiv`. Trivial by the correction of `TMMulRelRem` and `TMCompRelRem` and the fact that f/g can be written $f \circ 1/g$. □

```
1  Algorithm: TMDivRelRem(n̄, I, x_0, f, g, n)
   Input: n̄ ∈ ℕ a continuity order,
   I an interval,
   x_0 ⊆ I an expansion interval,
   f and g two functions such that g is at least n̄ + 1 times continuously differentiable on x_0,
   n ∈ ℕ
   Output: a TMRelative M corresponding to f/g
2  y_0 ← eval (g(x), x_0);
3  if 0 ∉ y_0 then
4      M_f ← TMRelRem(n̄, I, x_0, f, n);
5      M_g ← TMRelRem(n̄, I, x_0, g, n);
6      M ← TMBaseDiv(M_f, M_g, I, x_0, n);
7  else
8      k ← DetermineRootOrderUpperBound(n̄, x_0, g);
9      if k = ⊥ then
10         M_f ← TMRelRem(n̄, I, x_0, f, n);
11         M_g ← TMRelRem(n̄, I, x_0, g, n);
12         M ← TMBaseDiv(M_f, M_g, I, x_0, n);
13     else
14         (a_0, ..., a_{n+k}, Δ_f) ← TMRelRem(n̄, I, x_0, f, n + k);
15         (b_0, ..., b_{n+k}, Δ_g) ← TMRelRem(n̄, I, x_0, g, n + k);
16         l ← 0;
17         while a_l = [0; 0] ∧ b_l = [0; 0] do l ← l + 1;
18         M_f ← TMReduceOrder((a_l, ..., a_{n+k}, Δ_f), I, x_0, n + k − l, n);
19         M_g ← TMReduceOrder((b_l, ..., b_{n+k}, Δ_g), I, x_0, n + k − l, n);
20         M ← TMBaseDiv(M_f, M_g, I, x_0, n);
21     end
22 end
23 return M ;
```

Algorithm 2.3.12: Division of TMRelative

```
1  Algorithm: TMRelRem(n̄, I, x_0, h, n)
   Input: n̄ ∈ ℕ a continuity order,
   I an interval,
   x_0 ⊆ I an expansion interval,
   h a function that is at least n̄ times continuously differentiable on x_0,
   n ∈ ℕ a expansion order
   Output: a TMRelative M corresponding to h
2  switch h do
3      case h = x ↦ c: M ← TMConstRelRem(c, I, x_0, n);
4      case h = x ↦ x: M ← TMVarRelRem(I, x_0, n);
5      case h = f + g:
6          M_f ← TMRelRem(n̄, I, x_0, f, n);
7          M_g ← TMRelRem(n̄, I, x_0, g, n);
8          M ← TMAddRelRem(M_f, M_g, n);
9      endsw
10     case h = f · g:
11         M_f ← TMRelRem(n̄, I, x_0, f, n);
12         M_g ← TMRelRem(n̄, I, x_0, g, n);
13         M ← TMMulRelRem(M_f, M_g, I, x_0, n);
14     endsw
15     case h = f/g: M ← TMDivRelRem(n̄, I, x_0, f, g, n);
16     case h = g ∘ f:
17         M_f ← TMRelRem(n̄, I, x_0, f, n);
18         M ← TMCompRelRem(I, x_0, M_f, g, n);
19     endsw
20 endsw
21 return M ;
```

Algorithm 2.3.13: Computation of TMRelative

Proof of `TMRelRem` *(and* `TMDivRelRem`*).* We use structural induction on the expression tree for function h for the proof. When the algorithm calls one of `TMConstRelRem`, `TMVarRelRem`, `TMAddRelRem`, `TMMulRelRem` and `TMCompRelRem`, the correction is trivial by correction of the called sub-algorithm.

The problem is hence reduced to the case when the algorithm calls `TMDivRelRem`. By induction hypothesis, we can suppose that calls to `TMRelRem` on f and g produce correct TMRelative for f and g. This also allows us to use the proof for the partial correction of `DetermineRootOrderUpperBound`: that function is correct when called with input functions for which `TMRelRem` is correct.

We shall now prove that the result of `TMDivRelRem` on these particular functions f and g is correct. We use in what follows the notations of Algorithm `TMDivRelRem`. We separate two cases:

If $0 \notin \boldsymbol{y_0}$ or if `DetermineRootOrderUpperBound` returns $\perp$, the correction of `TMDivRelRem` reduces to the correction of `TMBaseDiv` and the correction of `TMRelRem` on f and g, which we have already shown.

Otherwise, we have $0 \in \boldsymbol{y_0}$ and k such that there exists $k' \leq k$ such that it is proven that $g^{(k')}$ does not vanish on $\boldsymbol{x_0}$. We shall now prove that the loop iterating on l terminates with $l \leq k'$. Suppose the contrary. Hence we have $\boldsymbol{b_0} = \cdots = \boldsymbol{b_{k'}} = [0;0]$. Since $(\boldsymbol{b_0}, \ldots, \boldsymbol{b_{n+k-1}}, \boldsymbol{\Delta_g})$ is a TMRelative of g at $\boldsymbol{x_0}$ on $\boldsymbol{I}$, we have

$$\forall \xi_0 \in \boldsymbol{x_0},\ \exists \underbrace{\beta_0}_{=0} \in \boldsymbol{b_0},\ \ldots \underbrace{\beta_{k'}}_{=0} \in \boldsymbol{b_{k'}}, \beta_{k'+1} \in \boldsymbol{b_{k'+1}},\ \ldots \beta_{n+k-1} \in \boldsymbol{b_{n+k-1}},\ \forall x \in \boldsymbol{I},\ \exists \delta_g \in \boldsymbol{\Delta_g},$$
$$g(x) = \sum_{i=0}^{n+k-1} \beta_i\ (x-\xi_0)^i + \delta_g\ (x-\xi_0)^{n+k}\,.$$

This yields

$$\forall \xi_0 \in \boldsymbol{x_0},\ \exists \beta_{k'+1} \in \boldsymbol{b_{k'+1}},\ \ldots \beta_{n+k-1} \in \boldsymbol{b_{n+k-1}},\ \forall x \in \boldsymbol{I},\ \exists \delta_g \in \boldsymbol{\Delta_g},$$
$$g(x) = \sum_{i=k'+1}^{n+k-1} \beta_i\ (x-\xi_0)^i + \delta_g\ (x-\xi_0)^{n+k}\,,$$

which finally gives

$$\forall \xi_0 \in \boldsymbol{x_0},\ \exists \beta_{k'+1} \in \boldsymbol{b_{k'+1}},\ \ldots \beta_{n+k-1} \in \boldsymbol{b_{n+k-1}},\ \forall x \in \boldsymbol{I},\ \exists \delta_g \in \boldsymbol{\Delta_g},$$
$$g(x) = (x-\xi_0)^{k'+1} \left(\sum_{i=0}^{n+k-k'-2} \beta_{i+k'+1}\ (x-\xi_0)^i + \delta_g\ (x-\xi_0)^{n+k-k'-1} \right).$$

Hence we know that for all $\xi_0 \in \boldsymbol{x_0}$, g has a zero of order at least $k'+1$ at ξ_0. It follows $g^{(k')}$ vanishes on the whole $\boldsymbol{x_0}$, which is a contradiction. The loop terminates with $l \leq k'$.

Now, in function of l, we separate two cases:

First, if $l = 0$, M_f and M_g are Taylor Models of f and g of degree n at $\boldsymbol{x_0}$ on $\boldsymbol{I}$ by correction of `TMReduceOrder` and induction hypothesis. So the result M is a Taylor Model with relative remainder of f/g at $\boldsymbol{x_0}$ on $\boldsymbol{I}$ by correction of `TMBaseDiv`.

Second, we suppose that $l \neq 0$. Hence $\boldsymbol{b_0} = [0;0]$ which means that for all $\xi_0 \in \boldsymbol{x_0}$, $g(\xi_0) = 0$. We can deduce from that that $\boldsymbol{x_0}$ is a point-interval $\boldsymbol{x_0} = [\xi_0;\xi_0]$: suppose the contrary. Then g vanishes everywhere at $\boldsymbol{x_0}$ and thus all its derivatives vanish at any interior point of $\boldsymbol{x_0}$. This implies that there is no suitable order k, returned by `DetermineRootOrderUpperBound`, for which $g^{(k)}$ does not vanish at $\boldsymbol{x_0}$. This is a contradiction. So $\boldsymbol{x_0}$ must be a point-interval $\boldsymbol{x_0} = [\xi_0;\xi_0]$.

Since the loop terminates, we have with its postcondition that $\boldsymbol{a_0} = \cdots = \boldsymbol{a_{l-1}} = [0;0]$ and $\boldsymbol{b_0} = \cdots = \boldsymbol{b_{l-1}} = [0;0]$. We shall now prove that M_f is a Taylor Model with relative remainder of $x \mapsto f(x) \,/\, (x-\xi_0)^l$ at $[\xi_0;\xi_0]$ on $\boldsymbol{I}$ and M_g is a Taylor Model of $x \mapsto g(x) \,/\, (x-\xi_0)^l$ at $[\xi_0;\xi_0]$ on $\boldsymbol{I}$.

Since $(\boldsymbol{a_0}, \ldots, \boldsymbol{a_{n+k-1}}, \boldsymbol{\Delta_f})$ is a Taylor Model of f at $\boldsymbol{x_0}$ on $\boldsymbol{I}$ we have

$$\forall \xi_0 \in \boldsymbol{x_0},\ \exists \underbrace{\alpha_0}_{=0} \in \boldsymbol{a_0},\ \ldots\ \underbrace{\alpha_{l-1}}_{=0} \in \boldsymbol{a_{l-1}}, \alpha_l \in \boldsymbol{a_l},\ \ldots \alpha_{n+k-1} \in \boldsymbol{a_{n+k-1}},\ \forall x \in \boldsymbol{I},\ \exists \delta_f \in \boldsymbol{\Delta_f},$$
$$f(x) = \sum_{i=0}^{n+k-1} \alpha_i\ (x-\xi_0)^i + \delta_f\ (x-\xi_0)^{n+k}.$$

which gives

$$\forall \xi_0 \in \boldsymbol{x_0},\ \exists \alpha_l \in \boldsymbol{a_l},\ \ldots, \alpha_{n+k-l-1} \in \boldsymbol{a_{n+k-l-1}},\ \forall x \in \boldsymbol{I},\ \exists \delta_f \in \boldsymbol{\Delta_f},$$
$$f(x) = (x-\xi_0)^l \left(\sum_{i=0}^{n+k-l-1} \alpha_{i+l}\ (x-\xi_0)^i + \delta_f\ (x-\xi_0)^{n+k-l} \right).$$

Hence $(\boldsymbol{a_l}, \ldots, \boldsymbol{a_{n+k-l-1}}, \boldsymbol{\Delta_f})$ is a Taylor Model with relative remainder of $x \mapsto f(x) \,/\, (x-\xi_0)^l$ of order $n+k-l$ at $[\xi_0;\xi_0]$ on $\boldsymbol{I}$. Since we have already shown that $l \leq k' \leq k$, we have $n \leq n+k-l$ and thus the correction of `TMReduceOrder` ensures that M_f is a Taylor Model with relative remainder of $x \mapsto f(x) \,/\, (x-\xi_0)^l$ of order n in $\boldsymbol{x_0} = [\xi_0;\xi_0]$. The same proof applies to g and M_g.

Since for all x, we have

$$\frac{\frac{f(x)}{(x-\xi_0)^l}}{\frac{g(x)}{(x-\xi_0)^l}} = \frac{f(x)}{g(x)},$$

we deduce that M is a Taylor Model with relative remainder of f/g of degree n at $\boldsymbol{x_0} = [\xi_0;\xi_0]$ on $\boldsymbol{I}$.

Remark that having now proven the correction of `TMRelRem` on any function h, we have also proven that correction of `TMDivRelRem` and `DetermineRootOrderUpperBound` on any function. In fact these functions are mutually recursive. □

Remark 2.3.10. *We implemented these modified Taylor models in the Sollya software tool and used them for handling functions with removable discontinuities. We report on several such examples in the next chapter.*

Remark 2.3.11. *A modified Taylor model is easy to convert into a classical model: it mainly suffices to multiply the bound $\boldsymbol{\Delta}$ by $(\boldsymbol{I} - \boldsymbol{x_0})^{n+1}$. We observe in practice that the enclosures obtained this way are generally a bit wider than the ones obtained with the usual Taylor models. This comes from the fact that the remainder bounds in our modified approach are wider intervals than in the classical approach, hence more subject to the dependency phenomenon. However, the gap between both methods stayed reasonably small in all examples we tried.*

2.3.3 Conclusion

All the algorithms presented in this chapter are implemented in Sollya. We will see in the next chapter one of their applications. A prototype implementation is also available as a Maple module at `http://www.ens-lyon.fr/LIP/Arenaire/Ware/ChebModels/Code/taylorModels.mw`. We strived for giving sufficiently simple and detailed algorithms for allowing their formal proof. Currently, one of our current goals in the TaMaDi project [116] is to have

formally proven polynomial approximations in COQ using the algorithms presented in this chapter. As future works, we are interested in extending this detailed account and our implementation to multivariate functions.

CHAPTER 3

Efficient and Accurate Computation of Upper Bounds of Approximation Errors

If the handbook says nothing much about the accuracy of the functions, then they had better be so accurate that nothing much need be said.

W. Kahan, Mathematics Written in Sand

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

Numerical algorithms for supremum norms are efficient but 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 chapter 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 that were already discussed in Chapter 2 and a non-negativity test based on a sum-of-squares expression of polynomials.

The new algorithm was implemented in the Sollya tool. We include experimental results on real-life examples. The work presented in this chapter is published in [36] and it is a joint work with S. Chevillard, J. Harrison and Ch. Lauter.

3.1 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 `libm`s. Such libraries are available on most systems and many numerical programs depend

on them. Examples include `CRlibm`, `glibc`, Sun * `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, as mentioned in Section 1.2, the IEEE 754-2008 standard [81] 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 [48].

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 Figure 3.1 the approximation error is plotted in a typical case when a minimax approximation polynomial of degree 5 is used [33, Chapter 3].

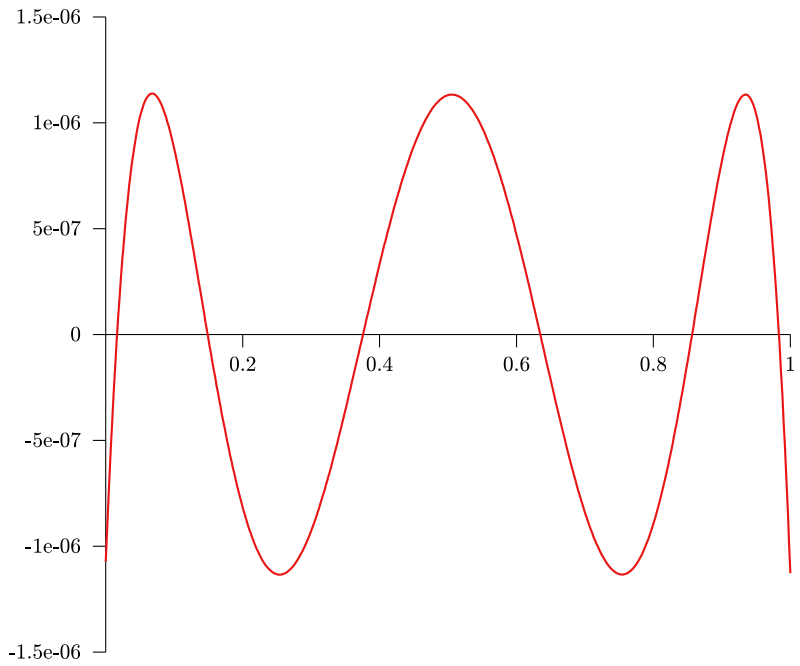


Figure 3.1: Approximation error in a case typical for a `libm`

A numerical algorithm tells us that the maximal absolute value of ε in this case (Figure 3.1) 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 chapter. In fact, this problem is part of the generic *rigorous polynomial approximation* class, see Problem 1. In this case, the approximation polynomial is given by a numerical routine, and our goal is to validate the approximation error. More precisely, we present an algorithm for computing a tight interval $\boldsymbol{r} = [\ell, u]$, such that $\|\varepsilon\|_\infty^I \in \boldsymbol{r}$. Here, $\|\varepsilon\|_\infty^I$

*. Other names and brands may be claimed as the property of others.

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 [119, 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 [40] for instance. The only requirement for basic functions is that they be differentiable up to a sufficiently high order.

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 `libm`s: 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 the same multiplicity as f, at least when this is possible.
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 $0 \le \frac{u-\ell}{\ell} \le \overline{\eta}$. 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 [141] 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.

3.1.1 Outline

In the next section, we explain the main ideas of previous approaches that partially fulfilled these goals and analyze their deficiencies. In Section 3.3, our algorithm is presented. It provides all the features presented above. As we will see, the algorithm relies on automatically computing an intermediate polynomial: for that we use the Taylor Models methods discussed in Chapter 2, especially TMs with relative remainders since these methods are able to handle removable discontinuities. In Section 3.5, we explain how a formal proof can be generated by the algorithm and checked by the HOL Light proof checker *. Finally, in Section 3.6 we show how our new algorithm behaves on real-life examples and compare its results with the ones given by previous approaches.

3.2 Previous work

3.2.1 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 [134]. 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 [134]). 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 [141] (briefly described also in Section 1.3).

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.

3.2.2 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 [106, 85, 76]. 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 rejected. The rejection tests are based on using interval arithmetic for bounding the

*. http://www.cl.cam.ac.uk/~jrh13/hol-light/

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 [139] 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" [124] (briefly described also in Section 1.3). 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 can not 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 [35]) 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 [39] a similar recursive technique was tried, but the authors 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.

3.2.3 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 [77, 124, 98, 38] 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 and for which we know that $\|f - T\|_\infty$ is much smaller than $\|f - p\|_\infty$: in practice the degree of T will be significantly larger than the degree of p. 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 [38].

Approaches that follow this idea have been developed in the past fifteen years [88, 77, 39, 79, 38]. 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 [88]. 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 [77] and more recently in [79], John Harrison 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 [77] and [79] mainly use the same techniques as Krämer: in particular, the formal proof is written by hand. This is safer than in [88], 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 [39, 38], we tried to automate the computation of an upper bound on $\|\varepsilon\|_\infty$. It did not seem obvious how to automate the technique used in [88, 77, 79], while correctly handling the particular difficulty of removable discontinuities. The algorithm presented in [39] was a first attempt. The idea was to enhance interval arithmetic [114], such that it could correctly handle expressions exhibiting removable discontinuities. As mentioned in Section 3.2.2 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 [38], was more suitable but not completely satisfying. This approach uses Taylor expansions, automatic differentiation and Lagrange formula for computing both the coefficients of T and for obtaining an interval bound $\mathbf{\Delta}$ for the Taylor remainder as discussed in Section 1.4.1. 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.

As we have already seen in Chapter 2, another way of computing automatically a Taylor form was introduced by Berz and his group [98] under the name of "Taylor models". 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 [12], written in FORTRAN by Berz and his group. Although highly optimized and used for rigorous global optimization problems in [14, 15] 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. In Section 2.3 we gave a solution to this problem, that we are going to use in the following.

3.3 Computing a safe and guaranteed supremum norm

3.3.1 Computing a validated supremum norm vs. 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 3.2.1, 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 [39, 38]. In contrast, our method ensures *by construction* that the quantity $|(u - \ell)/\ell|$ is smaller than a bound $\overline{\eta}$ given by the user.

3.3.2 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.

3.3.3 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 3.3.5. Our method is summed up in Algorithm 3.3.1.

We first run `computeLowerBound` with $\gamma = \overline{\eta}/32$. Roughly speaking, it 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 $\|\varepsilon\|_\infty \leq m$.

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

$$\|p - f\|_\infty \leq \|p - T\|_\infty + \|T - f\|_\infty\,. \tag{3.1}$$

```
1 Algorithm: supremumNormAbsolute
  Input: p, f, I, η̄
  Output: ℓ, u such that ℓ ≤ ‖p − f‖_∞^I ≤ u and |(u − ℓ)/ℓ| ≤ η̄
2 ℓ ← computeLowerBound(p − f, I, η̄/32);
3 m′ ← ℓ (1 + η̄/2);    u ← ℓ (1 + 31 η̄/32);    δ̄ ← 15 ℓ η̄/32;
4 T ← findPolyWithGivenError(f, I, δ̄);
5 s1 ← m′ − (p − T);    s2 ← m′ − (T − p);
6 if showPositivity(s1, I) ∧ showPositivity(s2, I) then return (ℓ, u);
7 else return ⊥ ;    /* Numerically obtained bound ℓ not accurate enough */
```

Algorithm 3.3.1: Complete supremum norm algorithm for $\|\varepsilon\|_\infty = \|p - f\|_\infty$

We can choose the polynomial T as close as we want to f. More precisely, we will describe in Section 3.4 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 3.5. 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 [143]. 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 (3.1), 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 in $m' = \ell\,(1 + \overline{\eta}/2)$.

3.3.4 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 $\|p - f\|_\infty \leq m$. 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 3.4.

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.

3.3.5 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 3.3.6.

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

Algorithm 3.3.2: Complete supremum norm algorithm for $\|\varepsilon\|_\infty = \|p/f - 1\|_\infty$

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.

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:

$$\left\|\frac{p}{T} - 1\right\|_\infty \leq \left\|\frac{p}{f} - 1\right\|_\infty + \left\|\frac{p}{f}\right\|_\infty \left\|\frac{1}{T}\right\|_\infty \|T - f\|_\infty \tag{3.2}$$

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

$$\overline{\delta} := \ell\,\mu\,\overline{\eta}\left(\frac{1}{1+u}\right)\left(\frac{F}{1+\mu\,\overline{\eta}}\right). \tag{3.3}$$

Lemma 3.3.1. *We have* $\|1/T\|_\infty \leq (1 + \mu\,\overline{\eta})/F$.

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

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

This concludes the proof. □

Using this lemma, and provided that the hypothesis $\|p/f - 1\|_\infty \leq m$ is correct, we see that Equation (3.2) implies $\|p/T - 1\|_\infty \leq m'$. 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 $|p/T - 1| \leq m'$ is equivalent to $|p - T| \leq s\, m'\, T$. 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 $\|p/T - 1\|_\infty \leq m'$ implies $\|p/f - 1\|_\infty \leq u$. For this purpose, we use Equation (3.2) where the roles of f and T are inverted:

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

In this equation, $\|p/T - 1\|_\infty$ is bounded by m', $\|p/T\|_\infty$ is bounded by $1 + m' \leq 1 + u$, $\|1/f\|_\infty$ is bounded by $1/F$ and $\|f - T\|_\infty$ is bounded by $\overline{\delta}$. Using the expression of $\overline{\delta}$ given by Equation (3.3), we finally have $\|p/f - 1\|_\infty \leq m' + \ell\, \mu\, \overline{\eta} \leq u$.

3.3.6 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 the introduction, the matter is not purely theoretical but 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 the sequel, 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 [39]: 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 $q(x) = \frac{p(x)}{(x-z_0)^{k_0} \dots (x-z_{s-1})^{k_{s-1}}}$ and $g(x) = \frac{f(x)}{(x-z_0)^{k_0} \dots (x-z_{s-1})^{k_{s-1}}}$.

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 3.3.2.

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 3.3.2 will fail while trying to compute F. This indicates the limits of our heuristic which, in practice, just works fine. See Section 3.6 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 addressed in Section 2.3 the problem of computing an intermediate approximation polynomial T and a finite bound $\overline{\delta}$ when the expression of f contains a removable discontinuity.

3.4 Obtaining the intermediate polynomial T and its remainder

In what follows, we detail the procedure `findPolyWithGivenError` needed in the algorithms 3.3.1 and 3.3.2. Given a function f, a domain I and a bound $\overline{\delta}$, it computes an intermediate approximation 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 $\mathbf{\Delta}$ rigorously enclosing $R(I)$: $\mathbf{\Delta} \supseteq R(I)$. We note that this means in fact that we are interested in finding an RPA for f that is easy to compute. We have already seen in Chapter 2 all the algorithms for computing RPAs based on Taylor approximations. We will use them in the following. However, these are not the only *easy to compute* RPAs. Furthermore, 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. For the moment, a simple bisection strategy over n, as described in Figure Algorithm 3.4.1, allows us to implement `findPolyWithGivenError` using the procedure `findPoly`. In the next chapter, we will see how to reduce the degree n while keeping the same error bound using better *easy to compute* RPAs.

The quality of an *RPA* is directly related to the quality of $\mathbf{\Delta}$. Two factors influence the quality of $\mathbf{\Delta}$: first, how well T approximates f over I, i.e. how small $\|R\|_\infty$ is, and second, how much overestimation will be introduced in $\mathbf{\Delta}$ when rigorously bounding R over I.

In that respect, the clever reader, would have already noticed that several kinds of approximation polynomials and methods for rigorously bounding the error could be considered. We recall below the "state-of-the-art" ones, which we analyzed and compared in detail in Section 1.4. We just mention that the keyword "rigorous" does not have to be over-looked. We are interested in bounding all the occurring errors, otherwise, a pleiad of numerical methods could be considered. Hence, the procedure `findPoly` could be implemented using:

- Methods that separately compute a polynomial T and afterwards a bound $\mathbf{\Delta}$:
  - The *interpolation polynomial method* that computes a near-optimal approximation (i.e. tries to minimize $\|R\|_\infty$) and then computes $\mathbf{\Delta}$ using *automatic differentiation*.
  - Methods based on Taylor expansions, where $\|R\|_\infty$ is larger, but $\mathbf{\Delta}$ is computed with less overestimation.
- The *Taylor models method* that simultaneously computes the polynomial T along with a bound $\mathbf{\Delta}$ discussed in detail in Chapter 2. We recall that the modified Taylor Models presented in Section 2.3 are able to handle removable discontinuities in all practical cases.

```
1  Algorithm: findPolyWithGivenError
   Input: f, I, δ̄
   Output: T such that ‖T − f‖_∞^I ≤ δ̄ while trying to minimize the degree of T
2  n ← 1;
3  do
4  |  (T, Δ_try) ← findPoly(f, I, n);    n ← 2n;
5  while Δ_try ⊈ [−δ̄; δ̄] ;
6  n ← n/2;    n_min ← n/2;    n_max ← n;
7  while n_min + 1 < n_max do
8  |  n ← ⌊(n_min + n_max)/2⌋;    (T, Δ_try) ← findPoly(f, I, n);
9  |  if Δ_try ⊆ [−δ̄; δ̄] then n_max ← n else n_min ← n ;
10 end
11 (T, Δ_try) ← findPoly(f, I, n_max);
12 return T ;
```

Algorithm 3.4.1: Find a polynomial T such that $\|T - f\|_\infty \leq \overline{\delta}$ with n as small as possible.

We have tested all these methods. In our experiments, we could not find one clearly dominating the others by tightness of the obtained bound $\mathbf{\Delta}$. 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 Chapter 4 we will introduce a new kind of RPA based on near-minimax approximation using Truncated Chebyshev Series or Chebyshev interpolators. We will show that this tool gives better results that the above state-of-the-art methods.

When this work was published, we were implementing and using "Taylor Models" (Algorithm 2.2.10) and in particular Algorithm 2.3.13 when removable discontinuities are detected. So, we give the results obtained like this. The comparison with the newer tool is given in the next chapter. We note that multiple precision interval arithmetic (MPFI library) is used throughout the implementation of our supremum norm algorithm in the Sollya tool.

3.5 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 [77, 111, 148, 125, 83, 78, 82, 18]. 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 [22]. 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 [17] and appears in many other contexts such as computational geometry [104].) 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 nonnegativity 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.

3.5.1 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 [78], 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 [175, 41], but much remains to be done, and performing the whole operation inside a formal checker appears to be very time-consuming.

3.5.2 Formalizing polynomial nonnegativity

Several approaches to formally proving polynomial nonnegativity have been reported, including a formalization of Sturm's theorem [77] and recursive isolation of roots of successive derivatives [78]. Many of these, as well as others that could be amenable to formalization [56], 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 [105]. 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 stays then 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 nonnegative, a 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 nonnegativity 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 [129] 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 [79] 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 nonnegative. 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 nonnegative 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 nonnegative. The complex roots can then be located sufficiently accurately using the excellent arbitrary-precision complex root finder in PARI/GP *, which implements a variant of an algorithm due to Schönhage [71].

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

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

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

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 $0 \leq i \leq n-1$, so

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

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

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

and then similarly for each $1 \leq i \leq n$ 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

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

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 $s(x)$.

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 $p_\varepsilon(x) = p(x) - \varepsilon(1 + x^2 + ... + x^{2m})$ is also (strictly) positive. For provided $\varepsilon < a_n$, this is certainly positive for sufficiently large x, say $|x| > R$, since the highest term of the difference $p(x) - \varepsilon(1 + x^2 + ... + x^{2m})$ will eventually dominate. And on the compact set $|x| \leq R$ we can just also choose $\varepsilon < \inf_{|x| \leq R} p(x) / \sup_{|x| \leq R}(1 + x^2 + ... + x^{2m})$.

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 nonnegative, 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 $c = 0$.

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 the roots $a_j \pm ib_j$ of $p_\varepsilon(x)$, we could obtain a SOS decomposition with two terms. Assuming 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.

Absorption of remainder term

We now have $p(x) = ls(x)^2 + lt(x)^2 + \varepsilon(1 + x^2 + ... + x^{2m}) + u(x)$, so it will suffice to express $\varepsilon(1 + x^2 + ... + x^{2m}) + u(x)$ 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 $u(x) = a_0 + a_1x + \ldots + a_{2m-1}x^{2m-1}$. 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

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

and so:

$$\begin{aligned} \varepsilon(1 + x^2 + ... + x^{2m}) + u = \; & \textstyle\sum_{k=0}^{m-1} |a_{2k+1}|(x^{k+1} + \mathrm{sgn}(a_{2k+1})\, x^k/2)^2 + \\ & \textstyle\sum_{k=0}^{m} (\varepsilon + a_{2k} - |a_{2k-1}| - |a_{2k+1}|/4)\, x^{2k}, \end{aligned}$$

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

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 nonnegativity 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 nonnegative 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 a SOS decomposition $q(y) = \sum_i c_i s_i(y)^2$ for nonnegative rational numbers c_i, it is useful to be able to transform back to a Positivstellensatz certificate [129] for the nonnegativity on $[a, b]$ of the original polynomial $p(x)$. So suppose 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

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

However, note that by construction $q(y)$ is an even polynomial, and so by comparing the odd terms on both sides we see that $\sum_i y r_i(y^2) t_i(y^2) = 0$. 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

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

while if d is odd, we have

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

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

3.6 Experimental results

We have implemented our novel algorithm for validated supremum norms in the Sollya software tool *. The sum-of-squares decomposition necessary for the certification step has been implemented using the PARI/GP software tool †. The formal certification step has been performed using the HOL light theorem prover ‡.

During the computation step before formal verification, the positivity of difference polynomials s_1 and s_2 (see Section 3.3) is shown using an interval arithmetic based implementation of the Sturm Sequence algorithm [144]. 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 [39]. In the examples presented, this fall-back has never been invoked. However, beside this method, other well known techniques exist [143].

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

*. http://sollya.gforge.inria.fr/
†. http://pari.math.u-bordeaux.fr/
‡. http://www.cl.cam.ac.uk/~jrh13/hol-light/

Section 2.3). Our implementation of Taylor models also contains some optimizations for computing tighter remainder bounds that were presented in Proposition 2.2.1 and 2.3.7.

We have compared the implementation of our novel supremum norm algorithm on 10 examples with implementations of the following existing algorithms discussed in Section 3.2:

- A pure numerical algorithm for supremum norms available in the Sollya tool through the command `dirtyinfnorm` [40]. 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 herein. 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` [40]. 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 [39]. 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 [38]. 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 [38]. They are summed up in Table 3.1. 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 $\|\varepsilon\|_\infty$.

More precisely, we can classify the examples as follows:

- The two first examples are somehow "toy" examples also presented in [39].
- The third example is a polynomial taken from the source code of `CRlibm`. It is the typical problem that developers of `libm`s 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 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* Fedora 10 x64 Linux 2.6.27.21. Table 3.2 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 3.3 gives the timings that we obtained.

*. Other names and brands may be claimed as the property of others.

N°	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†	$\arcsin(x + m)$	$[a_3,\, b_3]$	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

Table 3.1: Definition of our examples

N°	$\deg(p)$	$\deg(T)$	$\lVert\varepsilon\rVert_\infty \in [\ell,\, u]$
#1	5	13	0.98349131972[2−7]e−7
#2	7	17	0.21506063323225200140627 70[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

Table 3.2: Degree of the intermediate polynomial T chosen by `Supnorm`, and computed enclosure of $\lVert\varepsilon\rVert_\infty$

N°	`Ref1` time (ms)	`Ref2` time (ms)	`Ref3` time (ms)	`Supnorm` time (ms)	`SOS` time (ms)
	not rigorous	rigorous			
#1	14	2,190	121	42	1,631
#2	41	N/A	913	103	11,436
#3	270	N/A	1,803	364	42,735
#4	93	N/A	1,009	139	8,631
#5	337	N/A	2,887	443	155,265
#6	13	3,657	140	39	2,600
#7	180	N/A	3,220	747	81,527
#8	47	66,565	362	94	5,919
#9	27	5,109	315	73	3,839
#10	43	N/A	N/A	168	8,061

Table 3.3: Timing of several algorithms

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 to 5. This positive effect is reinforced by the fact that the absolute execution times for our supremum norm algorithm are less than 1 second in most cases. Hence supremum norm validation needs no longer be a one time - one shot overnight task as previous work suggests [39].

Even certification in a formal proof checker comes into reach with our supremum norm algorithm. In the last column of Table 3.3, we give the execution times for the post-computational rewriting of the difference polynomials s_i as a sum of squares.

Even if 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.

3.7 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, 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 was 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 due either to the unknown influence of parameters or simply because the techniques required too much manual work.

The supremum norm algorithm proposed in this work 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. 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 work, 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.

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

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 and refined in the previous chapter are implemented in a validated way, but will have to be "ported" to the environment of a formal proof checker. Previous works like [175] 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 basic 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 3.3, 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.

CHAPTER 4
Chebyshev Models

Dacă vor fi observat unii că mai judec și piezișș, sau că m-am dedulcit
la mai multe prostii decât alții, am astfel unele justificări formale.
Dan P. Brânzei

In previous chapters we have discussed about Taylor Models, a rigorous computing tool based on Taylor approximations. This approach benefits from the advantages of numerical methods, but also gives the ability to make reliable statements about the approximated function. Despite the fact that better approximations than Taylor exist, an analogous to Taylor models, based on other approximations, has not been yet well-established in the field of validated numerics.

In this chapter we present a new rigorous computing tool called Chebyshev Models, which is based on two approximations proved to be *near-best* in the sense of the uniform norm:

- Truncated Chebyshev series (TCS), i.e., the polynomial obtained by truncating the Chebyshev series of a function f.
- Chebyshev interpolants (CI), i.e., polynomials which interpolate the function at special points called Chebyshev nodes. They are also called "approximate truncated Chebyshev series" in literature.

We give several examples from global optimization or integration and compare the quality and performance of these new rigorous polynomial approximations to the ones provided by Taylor Models. We believe that these examples show that Chebyshev Models can be potentially useful in many alike rigorous computing applications.

This is a joint work with Nicolas Brisebarre, part of it is published in [27].

4.1 Introduction

In the previous chapters we presented a widely used rigorous computation tool: Taylor Models (TMs), which roughly speaking, associate to a function a pair made of a Taylor approximation polynomial and a rigorous approximation error bound. This approach benefits from the advantages of numerical methods, but also gives the ability to make reliable statements about the approximated function.

A natural idea is to try to replace Taylor polynomials with better approximations such as minimax approximation, truncated Chebyshev series or Chebyshev interpolants (also called approximate Chebyshev truncated series when the points under consideration are Chebyshev nodes) for instance. The last two kinds of approximations are of particular relevance for replacing Taylor polynomials since the series they define converge on domains better shaped for various usual

applications than Taylor expansions (see, for instance, Section 4.2.3 for a more detailed account). Moreover, we can take advantage of numerous powerful techniques for computing these approximations. So far, the attempts for using these better approximations, in the context of rigorous computing, do not seem to have succeeded, see for example [98] for a comparison of existing techniques. In this chapter we propose two approaches for computing models: one based on interpolation polynomials at Chebyshev nodes, what we call Chebyshev Interpolants (CI), the other on truncated Chebyshev series (TCS). We believe that bringing a certified error bound to an (approximate) truncated Chebyshev series, and providing effective tools for working with such models, opens the way to adapting to rigorous computing many numerical algorithms based on approximate Chebyshev series, for rigorous ODE solving, quadrature, etc.

4.1.1 Previous works for using tighter polynomial approximations in the context of rigorous computing

The idea of using better approximation polynomials, for example Chebyshev truncated series, in the context of validated computations was introduced in [60] under the name ultra-arithmetic. As explained in [98], in their setting, the advantages of non-Taylor approximations cannot be explicitly maintained due to several drawbacks. It is noted in [98] that Taylor representation is a special case, because for two functions f_1 and f_2 , the Taylor representation of order n, for the product $f_1 \cdot f_2$ can be obtained merely from the Taylor expansions of f_1 and f_2, simply by multiplying the polynomials and discarding the orders $n+1$ to $2n$. On the contrary, the Chebyshev truncated series of a product $f_1 \cdot f_2$ can in general not be obtained from the Chebyshev representations of the factors f_1 and f_2, and no operation analogous to TMs multiplication is given. Moreover, there is no systematic treatment of common basic functions. Finally, [98] explains that the methods developed in ultra-arithmetic can lead to an increase in the magnitude of the coefficients, which will increase both the computational errors and the difficulty of finding good interval enclosures of the polynomials involved.

In [176] a suggestion of directly using the minimax polynomial was made. In fact, this polynomial can be obtained only through an iterative procedure, namely Remez algorithm, which can be considered too computationally expensive in our context. We showed in Chapter 3 that the main drawback is linked to the computation of the approximation error bound: obtaining a certified bound in such a procedure, raises a significant dependency problem as discussed in Section 3.2.2 which led us to use Taylor Models as intermediary rigorous approximations. In [176] the idea that such minimax approximations models could be stored in a database was proposed, however this is very application specific and limited to a finite number of certain intervals over which the approximation was computed. It is hence not a general or scalable solution for such a problem.

Despite these unsuccessful attempts, we tried to find another way to take advantage of near-best approximations.

Trefethen [159] uses the idea of representing the functions by Chebyshev interpolation polynomials. He chooses their expansion length in such a way as to maintain the accuracy of the obtained approximation close to machine precision. Moreover, the idea of using basic functions and then consider an algebra on them is used. The developed software, Chebfun was successfully used for numerically solving differential equations, quadrature problems. However, the Chebfun system does not provide any validation of the obtained results. As mentioned in [159], the aim of this system are numerical computations and no formal proof or safeguards are yet implemented to guarantee a validated computation, although it seems to be a future project.

Although we will make use where possible of some fast numerical algorithms similar to those of Chebfun, one should remember that obtaining good rigorous bounds for enclosing approximation and rounding errors is not a straightforward process. For instance, one should not fall into the

old trap of the beginnings of interval arithmetic: in general one can not pretend that by replacing all floating-point computations with interval computations, one obtains a result that would give a tight enclosure of the rounding or approximation errors. So our work tries to cleverly adapt numerical algorithms or to conceive new ones that provide rigorous useful results in the end.

4.2 Preliminary theoretical statements about Chebyshev series and Chebyshev interpolants

In this section, we collect several theoretical results that we will use in the sequel. We try to explain the advantages, the mathematical meaning behind these advantages and of course, the disadvantages of using Chebyshev Truncated Series and Chebyshev interpolants.

4.2.1 Some basic facts about Chebyshev polynomials

A detailed presentation can be found in [19, 142]. We focus on the statements necessary for our approach. Over $[-1,1]$, Chebyshev polynomials can be defined as

$$T_n(x) = \cos\left(n \arccos x\right), n \geq 0.$$

We give several classical properties without proof:
– These polynomials can also be defined recursively as:

$$\begin{cases} T_n(x) &= 1 \\ T_1(x) &= x \\ T_n(x) &= 2xT_{n-1}(x) - T_{n-2}(x), n \geq 2; \end{cases} \tag{4.1}$$

– $\{T_0, T_1, \ldots, T_n\}$ form a *basis* for P_n, the space of polynomials of degree at most n over $[-1,1]$.
– T_{n+1} has $n+1$ distinct real roots in $[-1,1]$, called *Chebyshev nodes of first kind*:

$$\mu_i = \cos\left(\frac{(i+1/2)\,\pi}{n+1}\right), i = 0, \ldots, n. \tag{4.2}$$

– T_n has $n+1$ distinct local extrema over $[-1,1]$, called *Chebyshev nodes of second kind*:

$$\nu_i = \cos\left(\frac{i\,\pi}{n}\right), i = 0, \ldots, n. \tag{4.3}$$

Remark 4.2.1. *Over any interval $I = [a,b]$, $a \neq b$, a simple map of the interval I to $[-1,1]$ allows us to extend Chebyshev polynomials over I as*

$$T_n^{[a,b]}(x) = T_n\left(\frac{2x-b-a}{b-a}\right). \tag{4.4}$$

We now recall

Lemma 4.2.2. *The polynomial $W(x) = \prod_{i=0}^{n}(x-\mu_i)$, is the monic degree-$n+1$ polynomial that minimizes the supremum norm over $[-1,1]$ of all monic polynomials in $\mathbb{C}[x]$ of degree at most $n+1$. We have*

$$W(x) = \frac{T_{n+1}(x)}{2^n}$$

and

$$\max_{x\in[-1,1]} |W(x)| = \frac{1}{2^n}.$$

Beside this, Chebyshev polynomials possess another and equally important property: they form a family of orthogonal polynomials. Before going into details about this property, we mention that it is exploited in Chebyshev series expansions and Galerkin methods for differential equations [23].

Of course, there are other families of orthogonal polynomials beside Chebyshev, and they constitute a future extension of this work. That is why, several properties given in the sequel are presented in general setting of orthogonal polynomials. On the other hand, Chebyshev polynomials have further properties, which are peculiar to them and have a trigonometric origin, hence there is a strong link with Fourier series (see Remark 4.2.11) or, in the complex plane, Laurent series (see Remark 4.2.21).

We recall:

Definition 4.2.3. *Two functions $f(x)$ and $g(x)$ in $\mathcal{L}_2[a,b]$ are said to be orthogonal on the interval $[a,b]$ with respect to a given continuous and non-negative weight function $w(x)$ if $\langle f,\, g\rangle = 0$, where*

$$\langle f,\, g\rangle = \int_a^b w(x)f(x)g(x)\mathrm{d}x, \tag{4.5}$$

is the inner product between f and g.

The formal definition of an inner product (in the context of real functions of a real variable) is:

Definition 4.2.4. *Let V be a real vector space. An inner product $\langle\cdot,\,\cdot\rangle$ is a function from $V\times V$ to $\mathbb{R}$ such that, for all $f,g,h\in V$, $\alpha\in\mathbb{R}$ we have:*

1. *$\langle f,\, f\rangle \geqslant 0$, with equality if and only if $f=0$;*
2. *$\langle f,\, g\rangle = \langle g,\, f\rangle$;*
3. *$\langle f+g,\, h\rangle = \langle f,\, h\rangle + \langle g,\, h\rangle$;*
4. *$\langle \alpha f,\, g\rangle = \alpha\langle f,\, g\rangle$.*

An inner product defines an $\mathcal{L}_2$-type norm: $\|f\|_2 = \sqrt{\langle f,\, f\rangle}$.

A family of orthogonal polynomials $(\varphi_n)_{n\geqslant 0}$ where φ_n is of degree n verifies:

$$\langle \varphi_i,\, \varphi_j\rangle = 0, (i\neq j).$$

Lemma 4.2.5. *Chebyshev polynomials $(T_n)_{n\geqslant 0}$ form an orthogonal polynomial system i.e. $\langle T_i,\, T_j\rangle = 0$, with respect to the interval $[-1,1]$, weight function $w(x) = \frac{1}{\sqrt{1-x^2}}$.*

Remark 4.2.6. *The $\mathcal{L}_2$ norm of T_i is given by $\|T_0\|_2^2 = \pi$, $\|T_i\|_2^2 = \pi/2$, $i\geqslant 1$.*

Lemma 4.2.7. *If $\{\varphi_i\}$ is an orthogonal polynomial system on $[a,b]$ then:*

1. *the zero function is the best $\mathcal{L}_2$ polynomial approximation of degree $n-1$ to φ_n on $[a,b]$;*
2. *φ_n is the best $\mathcal{L}_2$ approximation to zero on $[a,b]$ among polynomials of degree n with the same leading coefficient.*

We observe that every polynomial in an orthogonal system has a minimal $\mathcal{L}_2$ property analogous to the minimax property of Chebyshev polynomials. Moreover, it can be proven that

Lemma 4.2.8. *The best $\mathcal{L}_2$ polynomial approximation p_n of degree n to f may be expressed in terms of the orthogonal polynomials family $\{\varphi_i\}$ in the form:*

$$p_n = \sum_{i=0}^{n} c_i\varphi_i,$$

where

$$c_i = \frac{\langle f,\varphi_i\rangle}{\langle \varphi_i,\varphi_i\rangle}.$$

4.2.2 Chebyshev Series

In a similar manner, on the assumption that it is possible to expand a given function f in a (suitably convergent) series based on the orthogonal polynomial system $(\varphi_n)_{n\geqslant 0}$, we may write f as an infinite orthogonal series:

$$f(x) = \sum_{k=0}^{\infty} a_k \varphi_k(x). \tag{4.6}$$

Focusing on Chebyshev series, what are the assumptions about the function f in order to ensure their convergence over $[-1, 1]$?

Here we restate without proof a result from [158, Chap. 3] that covers most applications, although a weaker assumption is sufficient [158, Chap. 7]. We shall assume that f is *Lipschitz continuous* on $[-1, 1]$. Recall that this means that there is a constant C such that $|f(x) - f(y)| \leq C|x-y|$ for all $x, y \in [-1, 1]$. Recall also that a series is *absolutely convergent* if it remains convergent if each term is replaced by its absolute value, and that this implies that one can reorder the terms arbitrarily without changing the result.

Theorem 4.2.9. *Chebyshev series*

If f is Lipschitz continuous on $[-1, 1]$, it has a unique representation as an absolutely and uniformly convergent series

$$f(x) = {\sum_{k=0}^{\infty}}' a_k T_k(x), \tag{4.7}$$

where the coefficients a_k are given for $k \geqslant 0$ by

$$a_k = \frac{2}{\pi} \int_{-1}^{1} \frac{f(x) T_k(x)}{\sqrt{1-x^2}} \mathrm{d}x, \tag{4.8}$$

and the dash in the summation indicating that the first coefficient is halved.

Remark 4.2.10. *Note that* $a_i = \dfrac{\langle f, T_i \rangle}{\langle T_i, T_i \rangle}$, $i \geqslant 0$.

There is a strong link between Chebyshev and Fourier series [102, Chap. 5.3]. In brief, with the usual change of variable $x = \cos\theta$, $0 \leqslant \theta \leqslant \pi$, the function $f(x)$ defines a new function $g(\theta)$, where $g : \mathbb{R} \to \mathbb{R}$ is an even periodic function of period 2π, with $g(\theta) = f(\cos\theta), 0 \leqslant \theta \leqslant \pi$, $g(\theta + 2\pi) = g(\theta)$ and $g(-\theta) = g(\theta)$.

In general, the Fourier series of a 2π-periodic function g may be written as:

$$g(\theta) = {\sum_{k=0}^{\infty}}' (u_k \cos(k\theta) + v_k \sin(k\theta)),$$

where $u_k = \dfrac{1}{\pi} \displaystyle\int_{-\pi}^{\pi} g(\theta) \cos(k\theta)\, \mathrm{d}\theta$ and $v_k = \dfrac{1}{\pi} \displaystyle\int_{-\pi}^{\pi} g(\theta) \sin(k\theta)\, \mathrm{d}\theta$.

Here, since g is even, all coefficients v_k vanish and hence the series simplifies to the Fourier cosine series of g.

Remark 4.2.11. *(Link between Chebyshev and Fourier series) The coefficients u_k of the Fourier cosine series of g defined above are the same as the coefficients of the Chebyshev series of f. This can be easily seen since, with the change of variable $x = \cos\theta$, $0 \leqslant \theta \leqslant \pi$, we have:*

$$a_k = \frac{2}{\pi} \int_0^{\pi} f(\cos\theta) \cos(k\theta)\, \mathrm{d}\theta = \frac{1}{\pi} \int_{-\pi}^{\pi} g(\theta) \cos(k\theta)\, \mathrm{d}\theta. \tag{4.9}$$

Remark 4.2.12. *An orthogonal expansion has the property that its partial sum of degree n is the best $\mathcal{L}_2$ approximation of degree n to its infinite sum.*

Hence it is an ideal expansion to be used in the $\mathcal{L}_2$ context. We will see in this chapter that truncated Chebyshev series are also *near-best* approximations with respect to the uniform norm.

Let us focus for the moment on the computation of the coefficients of this series.

Computation of Chebyshev series coefficients

In general, the Chebyshev coefficients can be obtained by some numerical computation of the integral (eq. (4.8)), but here we are interested in rigorous and fast ways of computing them. There are several functions for which the coefficients a_k in (4.8) may be determined explicitly, like in the following example:

Example 4.2.13. *Coefficients of Chebyshev expansion of $f(x) = \arccos(x)$.*
Integrating using formula (4.8) one obtains:

$$a_0 = \pi,\ a_{2k} = 0,\ a_{2k-1} = -\frac{2}{(2k-1)^2},\ k \geqslant 1.$$

The interested user can find more explicit Chebyshev expansions in [102, Chap. 5.2], but this is not possible in general. Moreover, even when this is possible for some basic functions, the coefficients of the explicit expansion contain sometimes expressions related to special functions like Bessel function, hypergeometric functions, etc. For example, for $\exp$, one finds the following explicit expansion:

Example 4.2.14. *Coefficients of Chebyshev expansion of $f(x) = \exp(x)$.*

$$a_k = 2I_k(1),\ k \geqslant 0,$$

where $I_k(z)$ is the modified Bessel function of the first kind.

So, going back to Definition 1.3.15 of "basic functions", we said that we consider as "basic function" any function for which we can easily compute its range over a given interval, then in Chapter 2 we considered "basic" any function for which the data structure needed to represent it (in our case, Taylor polynomial and interval error bound) was easily computable. Here, for Chebyshev series, in the case of $\exp$, we are faced with a dilemma : if we consider that we dispose of a "powerful" system that is capable of evaluating correctly rounded Bessel functions, than we can consider this expansion as "explicit" and consequently $\exp$ as basic function, otherwise we have to devise another way of computing these coefficients not relying on Bessel special functions.

Chebyshev coefficients for D-finite functions. We recall that in Section 1.4.1 we mentioned that the Taylor coefficients of such functions satisfy a linear recurrence with polynomial coefficients. A similar property holds for Chebyshev series coefficients, for details see Section 5.2.2 and references therein. For the moment, let us suppose that we dispose of an algorithm for obtaining this recurrence and consider the simple example of $\exp$.

Let $\exp(x) = c_0 T_0(x) + \sum_{i=1}^{\infty} c_i T_n(x)$, where $x \in [-1, 1]$. Based on the LODE verified by $\exp$, i.e. $f' - f = 0$, we can find the following recurrence verified by the Chebyshev coefficients $(c_k)_{n\in\mathbb{N}}$:

$$-c_k + (2\,k + 2)\,c_{k+1} + c_{k+2} = 0. \tag{4.10}$$

However, even in this simple example, there are several differences compared to the Taylor case. The first one is related to initial conditions. Unlike the Taylor coefficients of f, the Chebyshev

coefficients c_0, c_1 that could serve as initial conditions for the recurrence are not related to the initial condition $f(0) = 1$ of the differential equation in any simple way. In particular, we see that the order of the recurrence is larger than that of the differential equation, meaning that we need to somehow obtain more initial values for the recurrence than we "naturally" have at hand. We come back in more detail to this and other issues in Chapter 5. For this example, however, given that we know an explicit expansion for exp, we could use it to add the initial conditions $c_0 = \mathrm{I}_0(1)$ and $c_1 = \mathrm{I}_1(1)$.

At a first look this would imply that for computing n coefficients in a rigorous manner, we only need to compute the first two, i.e. $\mathrm{I}_0(1)$ and $\mathrm{I}_1(1)$, and then to unroll the recurrence and obtain all of them in linear time. However, this recurrence is not numerically stable.

Example 4.2.15 (Forward unrolling of the recurrence verified for Chebyshev coefficients of exp.)**.** *Given the recurrence (4.10), we compute in Sollya, using both double precision FP numbers and interval arithmetic the coefficients c_i, supposing that we have initially a high precision rigorous value/enclosure for c_0 and c_1:*

```
prec=54;

I_0_tilde= 1.26606587775200833559824;
I_0=[round(1.26606587775200833559824,54,RD)-1b-54,
      round(1.26606587775200833559824,54,RU)+1b-54];

I_1_tilde= 0.565159103992485027207696;
I_1=[round(0.565159103992485027207696,54,RD)-1b-54,
      round(0.565159103992485027207696,54,RU)+1b-54];

coeffs=[||]; coeffs_tilde=[||];
coeffs[0]=I_0;
coeffs[1]=I_1;

coeffs_tilde[0]=I_0;
coeffs_tilde[1]=I_1;

n=52;
for i from 0 to n-2 do{
coeffs[i+2]=-2*(i+1)*coeffs[i+1]+coeffs[i];
coeffs_tilde[i+2]=-2*(i+1)*coeffs_tilde[i+1]+coeffs_tilde[i];
};
midpointmode=on!;
print(coeffs_tilde);
print(coeffs);
```

Program 4.1: Forward unrolling of the recurrence verified for Chebyshev coefficients of exp using both FP and IA.

The obtained values for the coefficients are given in Table 4.2.15. We observe a very important precision loss. Of course, with IA, we have rigorous results, but they are already useless after the first 10 coefficients.

This issue is classical in the context of computational problems regarding recurrence relations, since we attempt to compute so called *minimal solutions*. We refer the reader for example

Table 4.1: Results obtained with double precision FP and IA for forward unrolling of the recurrence verified for Chebyshev coefficients of exp.

Coeff.	FP value	IA value	5 digit accurate value
0	1.266065877752008	$1.266065877752008[2,4]$	1.26606
5	$2.714631559685987 \cdot 10^{-4}$	$2.7146315[5,7] \cdot 10^{-4}$	$2.71463 \cdot 10^{-4}$
10	$-5.545631842629461 \cdot 10^{-9}$	$[-2 \cdot 10^{-8}, 2 \cdot 10^{-8}]$	$2.75294 \cdot 10^{-10}$
15	$4.519395853118779 \cdot 10^{-2}$	$[-9.75 \cdot 10^{-2}, 1.88 \cdot 10^{-1}]$	$2.37046 \cdot 10^{-17}$
20	$-2.027469166976402 \cdot 10^{6}$	$[-8 \cdot 10^{6}, 5 \cdot 10^{6}]$	$3.96683 \cdot 10^{-25}$
25	$3.318216535386249 \cdot 10^{14}$	$[-8 \cdot 10^{14}, 3 \cdot 10^{14}]$	$1.93990 \cdot 10^{-33}$
50	$-1.097390053134071 \cdot 10^{61}$	$[-5 \cdot 10^{61}, 3 \cdot 4^{61}]$	$2.93463 \cdot 10^{-80}$

to [66, 168] and briefly explain on this example the basic idea of the problem. We observe that the recurrence (4.10) is also verified by *BesselK function* $K_n(1)$, and that we have $\lim_{n\to\infty} \frac{I_n(1)}{K_n(1)} = 0$. Moreover, $I_n(1)$ and $K_n(1)$ form a basis of solutions of this recurrence. Any solution y_n that is not proportional to $I_n(1)$, can be written in the form $y_n = aI_n(1) + bK_n(1)$, $b \neq 0$. We thus have

$$\lim_{n\to\infty} \frac{I_n(1)}{y_n} = 0, \text{ for all such solutions } y_n. \tag{4.11}$$

When (4.11) is verified we say that $I_n(1)$ is a *minimal solution* and y_n is a *dominant solution*. When attempting to compute the minimal solution $I_n(1)$, and only approximate initial values are available (due to roundings for example), we will compute in fact a solution y_n, which is in general linearly independent of $I_n(1)$. Then, it is not difficult to see that the relative error becomes arbitrary large: $\lim \left| \frac{y_n - I_n(1)}{I_n(1)} \right| \to \infty$, as $n \to \infty$.

So, roughly speaking, before applying a recurrence relation for computing a function, it is necessary to determine whether it has a minimal solution and, if so, if the function under consideration is minimal. In such a case, one can use *Miller's algorithm* [16, 168] which is based on the interesting idea of applying the recurrence *backwards*. Of course, in such a case, initial conditions "at the tail" have to be provided.

For getting around this problem, we need to look at two aspects:

– whether we are interested in computing numerical (e.g. FP) values for coefficients that are "close" to the real values; this problem is dealt with is Chapter 5;

– or whether we need to compute rigorous enclosures of Chebyshev expansions coefficients c_i. For the second problem, we do not have yet a solution for any general recurrence verified by coefficients of D-finite functions, without resorting to rigorous computations of definite integrals. We do have however a solution for Chebyshev coefficients of "basic" functions. Similarly to the Taylor Models approach, we can analyze in a "case-by-case" manner every basic function, and compute *a priori* closed forms for the initial conditions needed when applying the recurrence backwards. To conclude our example, fixing the values of c_{50} and c_{49} in double precision IA and unrolling backwards the recurrence (4.10), we obtain $c_0 = 1.2660658777520[0,2]$.

The above issues tend to suggest that Taylor series have however more advantages. So, it is time to focus also on an important potential advantage of Chebyshev series compared to Taylor series.

4.2.3 Domains of convergence of Taylor versus Chebyshev series

In what follows, we consider a class of "very well-behaved" functions. We denote by $\mathcal{M}$ the set of functions which are *analytic* on a simply-connected open set U, except for a given set of points of U, their *singularities*: $Sg(f) = \{s_1, s_2, \ldots\}$.

Suppose that we want to approximate $f \in \mathcal{M}$ over the real axis segment $[-1, 1]$ by Taylor and Chebyshev series respectively. We suppose also that $f(x)$ is real for $x \in [-1, 1]$. So, the coefficients of both series are real.

Convergence of Taylor series

It is classical [2] that the Taylor series expansion of f at x_0 converges on a complex disc $\mathcal{D}$ centered at x_0 that avoids all the singularities of f.

For simplicity, assume $x_0 = 0$, then the Taylor series expansion of f converges in the region (plotted in Figure 4.1):

$$|x| < \rho, \text{where } \rho \leq \min_j \{|s_j|\}. \tag{4.12}$$

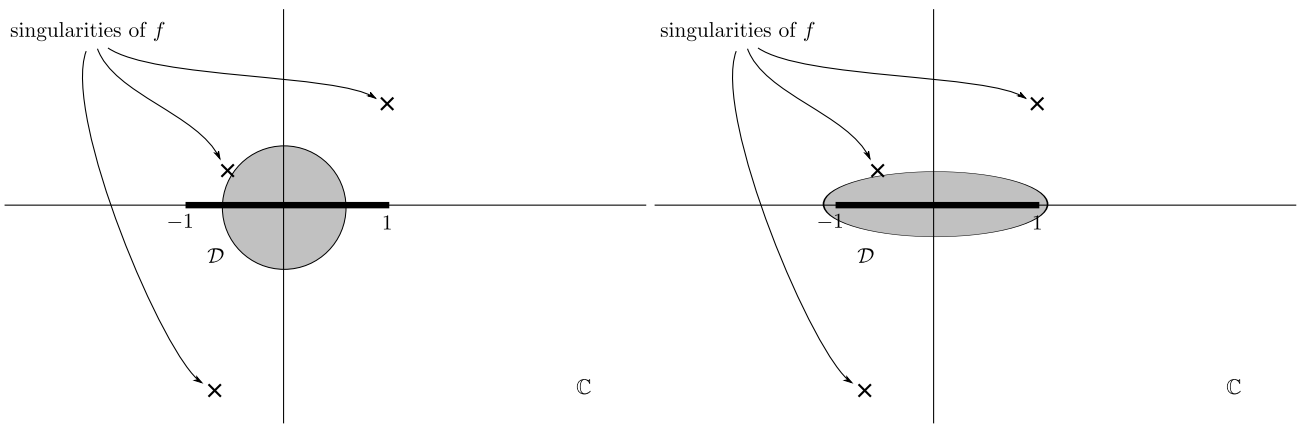


Figure 4.1: Domain of convergence of Taylor and Chebyshev series for f.

It is easily seen that the series can not converge over entire $[-1, 1]$ unless all the singularities s_j lie outside the unit circle.

Let f be an analytic function inside an open set $\Omega \subseteq \mathbb{C}$ containing the disc $\mathcal{D}_\rho = \{z, |z| < \rho\}$. We recall that the speed of convergence of the Taylor series coefficients of f inside $\mathcal{D}$ is given in the following classical theorem:

Theorem 4.2.16. *Cauchy's Estimate for Taylor coefficients of analytic functions inside the disc $\mathcal{D}_\rho = \{z, |z| < \rho\}$.*

Let $f(z) = \sum_{k=0}^{\infty} t_k\, z^k$ be analytic inside an open set $\Omega \subseteq \mathbb{C}$ containing the disc $\mathcal{D}_\rho = \{z, |z| < \rho\}$, for some $\rho > 0$. Then its Taylor coefficients satisfy

$$|t_k| \leq M\rho^{-k}, \text{ where } M = \max_{z \in \mathcal{D}_\rho} |f(z)|. \tag{4.13}$$

Corollary 4.2.17. *If f has the properties of Theorem 4.2.16 then the Taylor truncated series $f_n(z) = \sum_{k=0}^{n} t_k z^k$ satisfies for all $z \in \mathcal{D}_\rho$:*

$$|f(z) - f_n(z)| \leq \frac{M\beta^{-(n+1)}}{1-\beta}, \text{ where } M = \max_{z \in \mathcal{D}_\rho} |f(z)| \text{ and } \beta = \frac{|z|}{\rho}. \tag{4.14}$$

Convergence of Chebyshev series

The region of convergence of the Chebyshev series is the interior of an ellipse with foci at ± 1 (plotted in Figure 4.1); more precisely, it is the largest such ellipse for which all the singularities of f lie outside or on the ellipse.

In order to define the region of convergence algebraically we present without proof several complex geometry elements related to such ellipses ε_ρ , so called *Bernstein ellipses* [158, Chap. 8].

Probably the most widely known definition for an ellipse is: the set of points P in the complex plane such that the sum of the distances from P to the two foci is constant and equal to the length of the major axis. Let α, β the length of its major and minor axis respectively.

Definition 4.2.18. *Ellipse ε_ρ*
For any $\rho \geq 1$, ε_ρ is the ellipse of foci at -1 and 1 such that the sum of the lengths of its two axes is 2ρ.

Definition 4.2.19. *Elliptic disk $\bar{\varepsilon}_\rho$*
We denote by $\bar{\varepsilon}_\rho$ the elliptic disk bounded by ε_ρ i.e. the ellipse ε_ρ and its interior (in the geometric sense).

Proposition 4.2.20. *We have the following basic properties:*
- $\alpha^2 - \beta^2 = 4$;
- $\alpha = \rho + \rho^{-1}$ *and* $\beta = \rho - \rho^{-1}$.

We note that $\alpha(\rho)$ and $\beta(\rho)$ are increasing. Hence, the ellipse ε_{ρ_1} is in the interior of the ellipse ε_{ρ_2} if and only if $\rho_1 \leq \rho_2$. This gives us an intuitive notion of *radius* ρ of the ellipse ε_ρ. However, this can be viewed also through the so-called Joukowsky transform $w(z) = \dfrac{z + z^{-1}}{2}$: the image of the circle $C(0, \rho)$ of center 0 and radius ρ under this transformation is ε_ρ. Also, the image of the circle $C(0, \rho^{-1})$ of center 0 and radius ρ^{-1} under this transformation is ε_ρ. This, as well as the above mentioned properties of ellipses are illustrated in Figure 4.2.

This inverse of this transformation maps the elliptic disk $\bar{\varepsilon}_\rho$ to the annulus $A_\rho = \{z \in \mathbb{C} : \rho^{-1} < |z| < \rho\}$. Moreover, letting $x = \cos\theta$ and $z = e^{i\theta}$, the formula $T_n(\cos\theta) = \cos(n\theta)$ translates into $T_n(\frac{z+z^{-1}}{2}) = \frac{z^n + z^{-n}}{2}$. This allows one to easily see the link between Laurent series and Chebyshev series.

Remark 4.2.21. *Let $f(x) = \sum_{k=0}^{\infty}{}' a_k T_k(x)$. The coefficients $\tilde{a}_k$ of the (doubly infinite) Laurent expansion of the function $\tilde{f}(z) = f(\frac{z+z^{-1}}{2}) = \sum_{k=-\infty}^{\infty} \tilde{a}_k z^k$ around the unit circle verify $\tilde{a}_{|k|} = a_k/2$.*

Theorem 4.2.22. *Chebyshev coefficients of analytic functions inside the elliptic disk $\bar{\varepsilon}_\rho$.*
Let f be an analytic function inside an open set $\Omega \subseteq \mathbb{C}$ containing the elliptic disk $\bar{\varepsilon}_\rho$ for some $\rho \geq 1$. Then its Chebyshev coefficients satisfy

$$|a_k| \leq 2M\rho^{-k}, \text{ where } M = \max_{z \in \bar{\varepsilon}_\rho} |f(z)| . \tag{4.15}$$

Proof. Given for example in [158]. □

Now, suppose we are given the singularities $Sg(f)$ and none of them lies on the segment $[-1, 1]$. How can we compute the smallest $\rho > 1$ such that f is analytic inside the elliptic disk $\bar{\varepsilon}_\rho$?

Consider $z \in Sg(f)$. A simple computation using Proposition 4.2.20 suffices to prove that z lies on the Bernstein ellipse $\varepsilon_{r(z)}$ with $r(z) = \dfrac{\alpha_z + \sqrt{\alpha_z^2 - 4}}{2}$ and $\alpha_z = |z+1| + |z-1|$. Hence we have the following proposition:

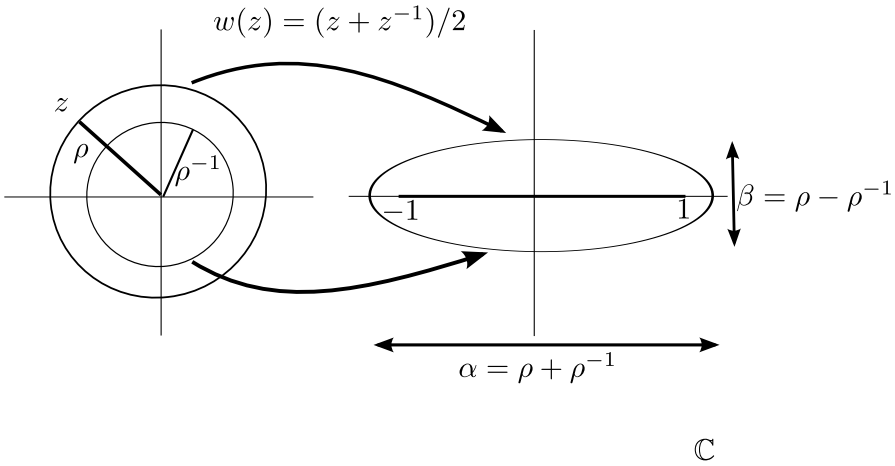


Figure 4.2: Joukowsky transform $w(z) = \dfrac{z + z^{-1}}{2}$ maps $C(0, \rho)$ and $C(0, \rho^{-1})$ respectively to ε_ρ.

Proposition 4.2.23. *Let f analytic on $[-1, 1]$ and let $\rho^* = \min_{s \in Sg(f)} r(s)$. Then, for all $\rho < \rho^*$ Theorem 4.2.22 applies.*

Remark 4.2.24. *If $Sg(f)$ is empty, i.e. f is analytic on the whole complex plane – it is said to be* entire*–, then Theorem 4.2.22 applies with $\rho^* = +\infty$. This means that the Chebyshev series coefficients of an* entire *function decrease faster than any geometric sequence.*

Remark 4.2.25. *A result adapted in [138] from Rivlin shows that ρ^* verifies:*

$$\frac{1}{\rho^*} = \limsup_{n \to \infty} |a_n|^{\frac{1}{n}}.$$

We can thus say that ρ^* represents the speed of convergence of the coefficients of the Chebyshev series of f.

We note that this provides a nice analogy with power series, where the radius of convergence r verifies $\dfrac{1}{r} = \limsup_{n \to \infty} |t_n|^{\frac{1}{n}}$ (Cauchy-Hadamard formula).

Theorem 4.2.26. *Convergence of analytic functions.*

If f has the properties of Theorem 4.2.22 then for each $n \geq 0$, the truncated Chebyshev series over $[-1, 1]$, $f_n(x) = \sum_{k=0}^{n}{}' a_k T_k(x)$ satisfy:

$$\|f - f_n\|_\infty \leq \frac{2M\rho^{-n}}{\rho - 1}, \text{ where } M = \max_{z \in \varepsilon_\rho} |f(z)|. \tag{4.16}$$

Hence, the *speed of convergence of the coefficients* of the Chebyshev series of f can be seen as the *speed of convergence of the Chebyshev series* of f on $[-1, 1]$. We note that convergence theorems exist for less "well-behaved" functions, that are not analytic, but at most k times differentiable, but we do not enter the details here and cite [158, Chap. 7] for interested readers.

Example 4.2.27. *We consider the Runge function $f(x) = \dfrac{1}{1 + 25x^2}$. It has two poles in $z_{1,2} = \pm\dfrac{i}{5}$. Then $\rho^* = \dfrac{1 + \sqrt{26}}{5}$. In Figure 4.3 we plotted the $\log_{10}$ of errors of Chebyshev truncated series in function of*

the truncation degree. These errors match the geometrical rate of convergence given by ρ^*. We note that the coefficients of the Chebyshev truncated series were computed using Maple code based on Chapter 5 of this thesis.

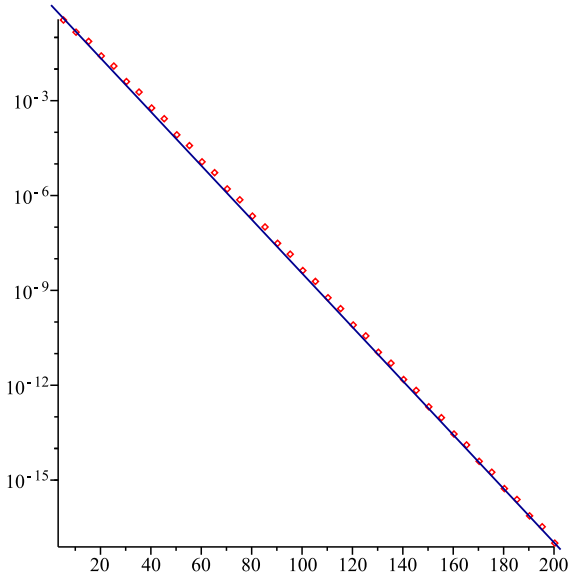


Figure 4.3: The dots are the errors $\log_{10} \|f - f_n\|_\infty$ in function of n, where $f(x) = \dfrac{1}{1+25x^2}$ is the Runge function and f_n is the truncated Chebyshev series of degree n. The line has slope $\log_{10} \rho^*$, $\rho^* = \dfrac{1+\sqrt{26}}{5}$.

Provided that none of the singularities lie on $[-1, 1]$, we can thus guarantee the convergence of Chebyshev series over $[-1, 1]$, while this is not the case for Taylor series. However, in this case, one solution for Taylor series would be to split $[-1, 1]$ into subintervals that can be covered by disks that avoid all singularities and then reason on each subinterval separately.

Now, let us recall the fact that we are interested in computing both the coefficients and an approximation error bound for such a truncated Chebyshev series. Although the convergence domain seems better, from their definition, we can not identify any straightforward computation neither for the coefficients, nor for the tail of the series for a given function f. For bounding the tail, one could of course try to identify ρ^* and compute M, but as we already discussed, like in the case of Taylor series, identifying the optimal ρ is not automatic and sometimes this bound can be very pessimistic.

One solution to bounding the tail of the series can be found in [59]:

Theorem 4.2.28. *Let $n \in \mathbb{N}$ fixed and a function $f \in \mathcal{C}^{n+1}[-1, 1]$. Let f_n the degree n truncated Chebyshev series of f on $[-1, 1]$. Then there exists $\xi \in (-1, 1)$ such that*

$$\|f - f_n\|_\infty = \frac{\left|f^{(n+1)}(\xi)\right|}{2^n\,(n+1)!}. \tag{4.17}$$

It is also well known that truncated Chebyshev series are *near-minimax* uniform approximations [102, Chap. 5.5]:

$$\|f - f_n\|_\infty \leqslant \left(\frac{4}{\pi^2}\log n + O(1)\right)\|f - p_n^*\|_\infty \tag{4.18}$$

where p_n^* is the best uniform approximation polynomial of degree at most n.

That being said, however, it seems that we have to adopt a different approach for computing the coefficients of Chebyshev expansions for most functions. One solution comes from another magnificent property of Chebyshev polynomials: *their discrete orthogonality*.

The link between Chebyshev series and Chebyshev interpolants We note that it is always possible to convert a (continuous) orthogonality relationship, as defined in Definition 4.2.3, into a discrete orthogonality relationship simply by replacing the integral with a summation. In general, of course, the result is only approximately true. However, where trigonometric functions or Chebyshev polynomials are involved, there are many cases in which the discrete orthogonality can be shown to hold exactly. For the Chebyshev polynomials $\{T_i(x), i = 0, \ldots, n\}$ we can prove that a discrete orthogonality holds for Chebyshev nodes of both kinds defined in Equations (4.2) and (4.3). More specifically, we have:

Proposition 4.2.29. *Discrete orthogonality of Chebyshev polynomials*

- *The polynomials $\{T_i(x), i = 0, \ldots, n\}$ are orthogonal over the discrete point set $\{\mu_k, k = 0, \ldots, n\}$, consisting of the zeros of $T_{n+1}(x)$ with respect to the discrete orthogonal product $\langle u, v\rangle = \sum_{k=0}^{n} u(\mu_k)v(\mu_k)$.*
  We have:

$$\langle T_i, T_j\rangle = \begin{cases} 0, & i \neq j, \\ n+1, & i = j = 0, \\ (n+1)/2, & i = j\ (\leqslant n). \end{cases} \tag{4.19}$$

- *The polynomials $\{T_i(x), i = 0, \ldots, n\}$ are orthogonal over the discrete point set $\{\nu_k, k = 0, \ldots, n\}$, consisting of the extrema of $T_n(x)$ with respect to the discrete orthogonal product $\langle u, v\rangle = \sum_{k=0}^{n}{}'' u(\nu_k)v(\nu_k)$, where $\sum''$ denotes that both the first and the last terms in the sum are to be halved.*
  We have:

$$\langle T_i, T_j\rangle = \begin{cases} 0, & i \neq j, \\ n, & i = j = 0 \text{ or } i = j = n, \\ n/2, & 0 < i = j < n. \end{cases} \tag{4.20}$$

This property makes the link between *truncated Chebyshev series* and *Chebyshev interpolation polynomials*.

4.3 Chebyshev Interpolants

Let us briefly review some classical facts about interpolation.

4.3.1 Interpolation polynomials

- Lagrange polynomial interpolation is classical. Consider a continuous function f on $I = [-1, 1]$ and a set of $n+1$ suitably selected distinct points in the interval, say $\{y_i, i = 0, \ldots, n\}$. There exists a unique polynomial p of degree $\leq n$ which interpolates f at these points [151]: $p(y_i) = f(y_i), \forall i = 0, \ldots, n$.

The interpolation polynomial p can be uniquely written as $p(x) = \sum_{i=0}^{n} f(y_i)\ell_i(x)$, where ℓ_i are Lagrange basis polynomials defined by $\ell_i(x) = \prod_{\substack{k=0\\k\neq i}}^{n} \frac{(x-y_k)}{(y_i-y_k)}$, $i = 0, \ldots, n$.

- Hermite polynomial interpolation denotes the similar process in which we require f to be at least $k-1$ times differentiable over I, we can repeat the points y_i at most k times. There exists a unique polynomial p of degree $\leq n$ which interpolates f at the given points and : $p(y_i) = f(y_i), \forall i = 0, \ldots, n$, or if y_i is repeated k times, $p^{(j)}(y_i) = f^{(j)}(y_i), \forall j = 0, \ldots, k-1$.
- Taylor polynomial is in fact the extreme case that all the y_i are equal.

The interpolation polynomial itself is easy to compute and a wide variety of methods and various bases representations for example monomial basis, Lagrange, Newton, Barycentric Lagrange, Chebyshev basis [33, 10, 151] exist (we refer the reader to any numerical analysis book, see [151] for instance, for its computation). We discuss below the so called *divided differences* computation of the coefficients in the Newton basis, since that allows us to directly deduce a formula for the interpolation error.

Let us consider the interpolation polynomial $p(x) = \sum_{i=0}^{n} c_i N_i(x)$, in the Newton basis $\{N_i, i = 0, \ldots n\}$, where

$$N_i(x) = \begin{cases} 1, & i = 0; \\ \prod_{j=0}^{i-1}(x-y_j), & i = 1, \ldots, n. \end{cases}$$

The coefficients c_i are the so-called divided-differences $f[y_0, \ldots, y_i]$ of f at the points $y_0, \ldots, y_i$. As mentioned above, if k points coincide, it suffices to take f and its $k-1$ successive derivatives of f. Two interesting properties of divided differences are the following [69, 151]:

- It can be proven that the following recurrence relation holds:
  $f[y_0, \ldots, y_i] = \dfrac{f[y_0, \ldots, y_{i-1}] - f[y_1, \ldots, y_i]}{y_0 - y_i}, i > 0.$
  From this recurrence a polynomial evaluation algorithm similar to Horner scheme in the monomial basis can be deduced.
- If the function $f \in \mathcal{C}^i$ over I then (see Chap. 4 of [151] for instance):

$$f[y_0, \ldots, y_i] = \int_0^1 \int_0^{t_1} \cdots \int_0^{t_i} f^{(i)}(y_0 + t_1(y_1 - y_0) + \ldots + t_i(y_i - y_{i-1}))\mathrm{d}t_1 \cdots \mathrm{d}t_i. \quad (4.21)$$

Interpolation Error The error between f and p is given [69, 151] by:

$$\forall x \in I, f(x) - p(x) = f[y_0, \ldots, y_n, x] \prod_{i=0}^{n}(x - y_i), \quad (4.22)$$

where $f[y_0, \ldots, y_n, x]$ is the divided-difference of f at the points $y_0, \ldots, y_n, x$.

By a repeated application of Rolle's theorem [33, 69, 151], we can prove that $\forall x \in I,\ \exists \xi \in (a, b)$ s.t.

$$f(x) - p(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \prod_{i=0}^{n}(x - y_i). \quad (4.23)$$

In the following we denote the right member of this formula for the error by $\Delta_n(x, \xi)$. In some cases, for efficiently computing an upper bound $\Delta_n(x, \xi)$ we will take advantage of the following Lemma which generalizes Proposition 2.3.7:

Lemma 4.3.1. *Under the assumptions on f and y_i above, if $f^{(n+1)}$ is increasing (resp. decreasing) over $\boldsymbol{I}$, then $f[y_0, \ldots, y_n, x]$ is increasing (resp. decreasing) over $\boldsymbol{I}$.*

Proof. Assume that $f^{(n+1)}$ is increasing. From Equation (4.21), for all $x \in [a, b]$, we have

$$f[y_0, \ldots, y_n, x] \;=\; \int_0^1 \int_0^{t_1} \cdots \int_0^{t_n} f^{(n+1)}(y_0 + t_1(y_1 - y_0) + \cdots + t_{n+1}(x - y_n))\mathrm{d}t_1 \cdots \mathrm{d}t_{n+1}.$$

Let $x,\ y \in [a, b]$, $x \leq y$, let Z_n denote $y_0 + t_1(y_1 - y_0) + \cdots + t_n(y_n - y_{n-1}) - t_{n+1}y_n$, we notice that

$$f[y_0, \ldots, y_n, y] - f[y_0, \ldots, y_n, x] = \int_0^1 \int_0^{t_1} \cdots \int_0^{t_n} \left(f^{(n+1)}(Z_n + t_{n+1}y) - f^{(n+1)}(Z_n + t_{n+1}x)\right) \mathrm{d}t_1 \cdots \mathrm{d}t_{n+1}.$$

Since $t_{n+1} \geq 0$, we have $Z_n + t_{n+1}y \geq Z_n + t_{n+1}x$. As $f^{(n+1)}$ is increasing, it follows that the integrand is nonnegative, which implies $f[y_0, \ldots, y_n, y] \geq f[y_0, \ldots, y_n, x]$. □

Quantifying good approximations: Lebesgue constant

Obviously $\Delta_n(x, \xi)$ depends on the choice of interpolation points. Suppose we are given a set S of $n + 1$ points in $[-1, 1]$. A famous notion used to quantify the amplification of errors in the interpolation process at these points is *Lebesgue constant*. Let us define it in the following. We recall first that the process of interpolation maps f to p. This defines a mapping $\tau_S : \mathcal{C}[-1, 1] \to \mathcal{C}[-1, 1]$ from the space $\mathcal{C}[-1, 1]$ to itself. This map is linear and it is a projection on the subspace $\mathcal{P}_n$ of polynomials of degree n or less.

The Lebesgue constant $\Lambda_n(S)$ is defined as the operator norm of τ_S, where we equip $\mathcal{C}[-1, 1]$ with the uniform norm $\|\cdot\|_\infty$, for example. Let p_n^* be the best approximation polynomial to f of degree at most n on $[-1, 1]$. Then, it is not difficult to prove the following inequality:

$$\|f - p_n^*\|_\infty \leqslant \|f - \tau_S(f)\|_\infty \leqslant (1 + \Lambda_n(S)) \|f - p_n^*\|_\infty .$$

In other words, this gives us a measure of how much worse can the interpolation polynomial $\tau_S(f)$ be compared to the best possible approximation. This suggests that we look for a set of interpolation nodes with a small Lebesgue constant.

Remark 4.3.2. *For example, when choosing the points y_i, the classical Runge phenomenon [102, Theorem 6.1.] shows that equally spaced points are not necessarily a good idea. The Lebesgue constant shows that, since for interpolation at evenly-spaced points it is asymptotically $2^{n+1}/(e\, n \log n)$.*

Good sets of interpolation points

As seen above, Lebesgue constant could be used as a measure for good interpolation points. Nevertheless, except for several classical sets S, it is not easy in general to find an explicit expression for $\Lambda_n(S)$.

Another way would be to find a set S for which the interpolation error $\Delta_n(x, \xi)$ is "small". Using this, we notice that Lemma 4.2.2 shows that the polynomial that minimizes the supremum norm of $W(x) = \prod_{i=0}^n (x - y_i)$ over $[-1, 1]$ is $\dfrac{T_{n+1}}{2^n}$. While it does not seem obvious how to control $f^{(n+1)}(\xi)$, this indicates that we can optimize at least $W(x)$. Consequently, an optimal choice of interpolation points is the Chebyshev nodes of first kind given in Equation (4.2).

It can be proven that this choice of interpolation points ensures uniform convergence for any Lipschitz continuous function f [33]. This condition is slightly more restrictive than requiring

simply f to be continuous. However, it can be proven that similarly to TCS, convergence in $\mathcal{L}_2$ norm occurs for any continuous f.

But several other choices of interpolation points for which $\Delta_n(x,\xi)$ is "small" are possible, for example Chebyshev nodes of second kind given in Equation (4.3). Lemma 4.3.4 shows that for this choice the interpolation error is as most two times bigger than the one obtained for Chebyshev nodes of first kind. Moreover, if f is Lipschitz continuous on $[-1,1]$, then the sequence of Chebyshev interpolants converges uniformly to f on $[-1,1]$. Actually, this holds under weaker hypothesis [102, Chap. 6] [158, Chap. 7].

For both kinds of Chebyshev nodes, one can prove that the interpolation polynomial is *near-best*.

Lemma 4.3.3. *The Lebesgue constant for Chebyshev nodes of first kind satisfies:*

$$\Lambda_\mu(n) = \frac{1}{\pi}\sum_{k=1}^{n+1}\left|\cot\frac{(k-1/2)\pi}{2(n+1)}\right|, \tag{4.24}$$

which is asymptotically

$$\Lambda_\mu(n) = \frac{2}{\pi}\log n + 0.9625 + \mathcal{O}(1/n). \tag{4.25}$$

The Lebesgue constant for Chebyshev points of second kind satisfies:

$$\Lambda_\nu(n) = \begin{cases} \Lambda_\mu(n-1), & n \text{ odd}, \\ \Lambda_\mu(n-1) - \alpha_n,\ \frac{\pi/8}{n^2} \leqslant \alpha_n < \frac{2\sqrt{2}-2}{n^2}, & n \text{ even}. \end{cases} \tag{4.26}$$

Proof. We refer to [31, Equations 12,13, 24] for an interesting survey and further references. □

Lemma 4.3.4. *Interpolation Error*

Let $f \in \mathcal{C}^{n+1}[-1,1]$, $p_1, p_2 \in \mathcal{P}_n$ interpolation polynomials at $n+1$ Chebyshev nodes of first and second kind respectively. Then there exists $\xi \in (-1,1)$ such that

$$\|f - p_1\|_\infty \leq \frac{\left|f^{(n+1)}(\xi)\right|}{2^n\,(n+1)!} \tag{4.27}$$

and

$$\|f - p_2\|_\infty \leq \frac{\left|f^{(n+1)}(\xi)\right|}{2^{n-1}\,(n+1)!}. \tag{4.28}$$

Proof. Inequality (4.27) is obvious using Equation (4.23) and Lemma 4.2.2. For inequality (4.28), using Equation (4.23) we have:

$$f(x) - p_2(x) = \frac{1}{2^{n-1}(n+1)!} f^{(n+1)}(\xi)(x^2-1)\,U_{n-1}(x),$$

where $U_{n-1}(x) = \sin(n\theta)/\sin\theta$ is the second kind Chebyshev polynomial and $x = \cos\theta$, see for example [102, Sec. 1.2.2]. Using this definition we have $|(x^2-1)\,U_{n-1}(x)| \leqslant 1$. □

In what follows, we focus on interpolation at Chebyshev nodes.

Proposition 4.3.5. *Chebyshev interpolation formulas*

Let $p_{1,n}(x) = {\sum_{i=0}^{n}}' c_{1,i}T_i(x)$ be the degree n polynomial interpolating f on the set $\{\mu_k, k = 0,\dots,n\}$ of roots of T_{n+1}. The coefficients $c_{1,i}$ are given by the explicit formula:

$$c_{1,i} = \frac{2}{n+1}\sum_{k=0}^{n} f(\mu_k)T_i(\mu_k). \tag{4.29}$$

Proposition 4.3.6. *Chebyshev interpolation formulas*

Let $p_{2,n}(x) = \sum_{i=0}^{n}{}'' c_{2,i}T_i(x)$ *be the degree* n *polynomial interpolating* f *on the set* $\{\nu_k, k = 0, \ldots, n\}$ *of extrema of* T_n *over* $[-1, 1]$. *The coefficients* $c_{2,i}$ *are given by the explicit formula:*

$$c_{2,i} = \frac{2}{n}\sum_{k=0}^{n} f(\nu_k)T_i(\nu_k). \tag{4.30}$$

Proof. Both Proposition 4.3.5 and 4.3.6 are based on the discrete orthogonality property of Chebyshev polynomials (Property 4.2.29). □

Relations between Chebyshev interpolants coefficients and truncated Chebyshev series coefficients. We have seen that both the coefficients of TCS and CI can be obtained using the discrete or continuous orthogonality properties of the Chebyshev polynomials. Another way of seeing the connection between them is to look at the *aliasing* phenomenon of Chebyshev polynomials, described as follows.

Theorem 4.3.7. *Aliasing formula for Chebyshev coefficients*

Let f *be Lipschitz continuous on* $[-1, 1]$ *and let* $\{c_{1,k}\}$, $\{c_{2,k}\}$ *be the coefficients of Chebyshev interpolants defined in Proposition 4.3.5 and 4.3.6. Let* $\{a_k\}$ *be the coefficients of Chebyshev expansion of* f. *Then*

$$c_{1,k} = a_k - a_{2n+2-k} - a_{2n+2+k} + a_{4n+4-k} + a_{4n+4+k} - \ldots,\ k = 0, \ldots, n.$$
$$c_{2,k} = a_k + (a_{k+2n} + a_{k+4n} + \ldots) + (a_{-k+2n} + a_{-k+4n} + \ldots),\ k = 0, \ldots, n$$

Proof. We give the ideas of the proof. For details, please see [158, Chap. 4], [102, Section 6.3.1]. Using the definition of T_n, the following identity holds:

$$T_j + T_{2n+2\pm j} = \frac{1}{2}T_{n+1}T_{n+1\pm j}.$$

Hence, when interpolating at $\{\mu_k, k = 0, \ldots, n\}$ the zeros of T_{n+1}:

$$T_{2n+2\pm j}(\mu_k) = -T_j(\mu_k).$$

Now, using Proposition 4.3.5 and Theorem 4.2.9, since f has a unique Chebyshev series and it converges absolutely, we can rearrange the terms of the series without affecting convergence and obtain Eq. (4.3.7). In a similar manner we can deduce a similar identity for coefficients of interpolants at Chebyshev nodes of second kind. □

From this theorem we can deduce an error formula for the interpolation polynomial based on the coefficients of TCS. We have

$$f(x) - p_{2,n}(x) = \sum_{k=n+1}^{\infty} a_k \left(T_k(x) - T_{|(k+n-1)(\mathrm{mod}\,2n)-(n-1)|}(x)\right); \tag{4.31}$$

$$f(x) - p_{1,n}(x) = \sum_{k=n+1}^{\infty} a_k \left(T_k(x) - (-1)^s T_{|\tilde{k}|}(x)\right), \text{ where } \tilde{k} = (k+n+1)(\mathrm{mod}\,2n+2) - (n+1),$$
$$s = \begin{cases} 0, \text{if } (k+\tilde{k})(\mathrm{mod}\,4n+4) = 0, \\ 1, \text{otherwise.} \end{cases} \tag{4.32}$$

From here we can deduce the interesting remark that the interpolation error can be at most 2 times bigger than the TCS error.

$$\|f - f_n\|_\infty \leq \sum_{k=n+1}^{\infty} |a_k|; \tag{4.33}$$

$$\|f - p_n\|_\infty \leq \sum_{k=n+1}^{\infty} 2\,|a_k| = 2 \sum_{k=n+1}^{\infty} |a_k|, \tag{4.34}$$

where p_n is either $p_{1,n}$ or $p_{2,n}$.

Based on this relation, we can also extend Theorem 4.2.26 to CI. We cite [158, Chap. 8] for detailed proofs.

Theorem 4.3.8. *Convergence of Chebyshev interpolants of analytic functions.*

If f has the properties of Theorem 4.2.22 then for each $n \geq 0$, the Chebyshev interpolant at Chebyshev nodes (of first or second order) over $[-1, 1]$, p_n satisfy:

$$\|f - p_n\|_\infty \leq \frac{4M\rho^{-n}}{\rho - 1}, \text{ where } M = \max_{z \in \varepsilon_\rho} |f(z)|\,. \tag{4.35}$$

To summarize, advantages for choosing interpolation polynomials at Chebyshev points are:

- Expressing the polynomial in Chebyshev basis with this choice of interpolation points is proven to be more efficient and stable from a computational point of view than the equally-spaced set and the monomial basis. From Theorem 4.3.8 and 4.3.7 we see that for functions analytic in an elliptic disk which contains $[-1, 1]$ the sequence of interpolation polynomials converges uniformly and the coefficients decrease rapidly and that they converge individually with n. Moreover, it can be proven that Chebyshev basis is well-conditioned, see [67] for more details.
- As discussed above, this approximation is near-minimax as well as that obtained by truncating the Chebyshev series expansion - but in this case it is obtained by a much simpler procedure.
- For bounding the approximation error we benefit both from the closed-error formula given in Lemma 4.3.4 and also from the possibility of computing exactly the range of this error for certain functions using Lemma 4.3.1.

Potential disadvantages are:

- When using an interpolation polynomial of degree n one has to evaluate the function f at $n + 1$ points. The cost of such evaluation is not anodyne in all cases, for multiple precision evaluation of basic functions see [63].
- It would be interesting to be able to use also Theorem 4.3.8 for bounding the approximation error. Unfortunately, similarly to what we discussed in Section 1.4.1 regarding Cauchy's estimate, for the moment we do not dispose of an automatic algorithm for choosing the optimal radius ρ such that sufficiently tight bounds are computed in the end.

Operations with polynomials in Chebyshev basis We now turn to the examination of common operations like addition, multiplication, evaluation, etc. with finite Chebyshev series. Their analogues in power series are well-established and much research was devoted to having good operation complexity bounds for such algorithms.

On the one hand, one may think of converting from Chebyshev to monomial basis, performing the operations in this basis and then convert back to the initial Chebyshev basis. Let us first analyze this case in terms of arithmetic complexity. For this, let $p, q \in \mathbb{K}[x]$ be polynomials of

degree n with coefficients in some field $\mathbb{K}[x]$, written in Chebyshev basis: $p(x) = \sum_{i=0}^{n} p_i T_i(x)$ and $q(x) = \sum_{i=0}^{n} q_i T_i(x)$. In terms of arithmetic complexity, for both back and forth conversions, the best algorithms given in [20, 21] are in $\mathcal{O}(M(n))$ arithmetic operations in $\mathbb{K}$, where M is the usual multiplication time function, such that polynomials of degree less than n in $\mathbb{K}[x]$ can be multiplied in $M(n)$ operations in $\mathbb{K}$, when written in the monomial basis. Using Fast Fourier Transform algorithms, $M(n)$ is usually done in $\mathcal{O}(n\log(n))$ over fields with suitable roots of unity, and in $\mathcal{O}(n\log(n)\log\log(n))$ over any field [166, Chap. 8].

On the other hand, from a numerical point of view, monomial basis is not stable and during the transformation, the coefficients in this basis may be considerably larger than those in Chebyshev basis simply because the power series convergence domain will not be sufficient to cover $[-1, 1]$ as discussed above. So, we are interested in finding algorithms for operations with truncated Chebyshev series that do not make use of conversion to monomial basis. We describe the classical ones in what follows.

Addition. Adding p and q in Chebyshev basis is straightforward and takes $\mathcal{O}(n)$ operations:

$$p(x) + q(x) = \sum_{i=0}^{n} (p_i + q_i) T_i(x). \tag{4.36}$$

Multiplication. The product can be expressed in Chebyshev basis as follows:

$$p(x) \cdot q(x) = \sum_{k=0}^{2n} c_k T_k(x), \text{where } c_k = \frac{1}{2}\left(\sum_{|i-j|=k} p_i \cdot q_j + \sum_{i+j=k} p_i \cdot q_j\right). \tag{4.37}$$

This identity can be obtained noting that $T_i(x) \cdot T_j(x) = (T_{i+j} + T_{|i-j|})/2$ [102]. The cost using this simple identity is $O(n^2)$ operations. We note that in [70] an algorithm of complexity $2M(n)$ is given, but in this case also, there seem to be some numerical stability issues and no thorough study was yet undertaken.

Composition The composition $(p \circ q)(x) = p(q(x))$ can be expressed in Chebyshev basis using a recursive algorithm for evaluation of polynomials in Chebyshev basis which we will discuss below. One apparent difficulty seems to be that the range of $q(x)$ is not necessarily in $[-1, 1]$, while so far this is the interval over which we presented all the results. The solution is simple, using Remark 4.2.1, which says that with the simple change of variable which maps any interval $I = [a, b]$, $a \neq b$ to $[-1, 1]$:

$$x \mapsto \frac{2x - b - a}{b - a},$$

we extend Chebyshev polynomials over I.

For the sake of clarity and implementation reasons we continue the presentation using the basis of Chebyshev Polynomials over $[a, b]$: $\left(T_i^{[a,b]}\right)_{i \geqslant 0}$, with $T_i^{[a,b]}(x) = T_i\left(\frac{2x - b - a}{b - a}\right)$.

Remark 4.3.9. *We note that the image of* $T_i^{[a,b]}(x) = T_i\left(\frac{2x-b-a}{b-a}\right)$ *over* $[a, b]$ *is exactly* $T_i([-1, 1])$, *which is* $[-1, 1]$ *for* $i \geqslant 1$ *and* 1 *for* $i = 0$.

Evaluation. To evaluate p at a given value x, we can use Clenshaw's algorithm [102, Chap. 2.4.1.]. It is a recursive method to evaluate linear combinations of polynomials that verify a three-term recurrence relation, in particular Chebyshev polynomials. It is similar to Horner's scheme, and it has linear complexity. It is described in Algorithm 4.3.1 whose correction proof is given below.

Algorithm: `EvaluateClenshaw`$(p_0, \dots, p_n, x, [a, b])$
Input: $[a, b]$ an interval, $a \neq b$,
x a point in $[a, b]$,
$p_0, \dots, p_n$ coefficients of $p(x) = \sum_{i=0}^{n} p_i T_i^{[a,b]}(x)$.
Output: value of $p(x)$.
$b_{n+1} \leftarrow 0$;
$b_{n+2} \leftarrow 0$;
for $k \leftarrow n$ **downto** 0 **do**
  $b_k \leftarrow p_k + \dfrac{2(2x-b-a)}{b-a} b_{k+1} - b_{k+2}$;
end
return $b_0 - \dfrac{(2x-b-a)}{b-a} b_1$;

Algorithm 4.3.1: Clenshaw's algorithm for evaluating Chebyshev sums

Proof of Algorithm 4.3.1. Let us write the recurrence relations verified by $\left(T_i^{[a,b]}\right)_{i \geqslant 0}$ in matrix notation. Let

$$A = \begin{pmatrix} 1 & & & & & & \\ \frac{-2(2x-b-a)}{b-a} & 1 & & & & & \\ 1 & \frac{-2(2x-b-a)}{b-a} & 1 & & & & \\ & 1 & \frac{-2(2x-b-a)}{b-a} & 1 & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & 1 & \frac{-2(2x-b-a)}{b-a} & 1 \end{pmatrix},$$

and

$$t = \begin{pmatrix} T_0^{[a,b]}(x) \\ T_1^{[a,b]}(x) \\ T_2^{[a,b]}(x) \\ T_3^{[a,b]}(x) \\ \vdots \\ T_n^{[a,b]}(x) \end{pmatrix}, \quad r = \begin{pmatrix} 1 \\ \frac{-(2x-b-a)}{b-a} \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}, \quad p = \begin{pmatrix} p_0 \\ p_1 \\ \\ \vdots \\ p_n \end{pmatrix}, \quad b = \begin{pmatrix} b_0 \\ b_1 \\ \\ \vdots \\ b_n \end{pmatrix}.$$

We have: $At = r$ and $p(x) = p^T t$ and we notice that $\{b_k, k = 0, \dots, n\}$ computed by Algorithm 4.3.1 verify: $b^T A = p^T$. Hence we have: $p(x) = b^T A t = b^T r = b_0 - \dfrac{(2x-b-a)}{b-a} b_1$.

□

Interval Range Bounding. One simple way of obtaining a rigorous range bound over $[a, b]$ for a polynomial expressed in Chebyshev basis $\left(T_i^{[a,b]}\right)_{i \geqslant 0}$ is to use Remark 4.3.9 which gives the following identity which takes $O(n)$ operations:

$$\forall x \in [a, b], p(x) \in p_0 + \sum_{i=1}^{n} p_i \cdot [-1, 1]. \tag{4.38}$$

Integration and differentiation.
The indefinite integral of $T_n(x)$ can be expressed as follows:

$$\int T_n(x)\mathrm{d}x = \begin{cases} \frac{1}{2}\left(\frac{T_{n+1}(x)}{n+1} - \frac{T_{|n-1|}(x)}{n-1}\right), \ n \neq 1; \\ \frac{1}{4}T_2(x), \ n = 1. \end{cases} \tag{4.39}$$

So, the indefinite integral of p is $I_{n+1}(x) = \int p(x)\mathrm{d}x = \sum_{i=0}^{n+1} A_i T_i(x)$, with A_0 determined from the constant of integration, and

$$A_i = \frac{p_{i-1} - p_{i+1}}{2i}, \ i \in \{1, \ldots, n+1\}, \ p_{n+1} = p_{n+2} = 0. \tag{4.40}$$

This means that the coefficients of the integral of p can be determined in $O(n)$ arithmetic operations.

Also, we can compute the derivative of I_{n+1} in $O(n)$ arithmetic operations using Equation (4.40), in the form: $p_{i-1} = 2iA_i + p_{i+1}$.

In the next sections we work in more generality with other intervals besides $[-1, 1]$, so we deem it useful to give in the following a small summary of the simple extension of the above results over $[a, b]$.

4.4 Summary of formulas

With the simple change of variable which maps any interval $I = [a, b]$, $a \neq b$ to $[-1, 1]$:

$$x \mapsto \frac{2x - b - a}{b - a},$$

we extend Chebyshev polynomials over I, using Remark 4.2.1. Several results given above generalize easily over I and we give them in what follows:

1. Chebyshev Nodes:
   - of first kind:
   $$\mu_i^{[a,b]} = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{(i+1/2)\,\pi}{n+1}\right), i = 0, \ldots, n. \tag{4.41}$$
   - of second kind:
   $$\nu_i^{[a,b]} = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{i\,\pi}{n}\right), i = 0, \ldots, n. \tag{4.42}$$
2. Computation of coefficients of Chebyshev interpolants at Chebyshev nodes defined above over $[a, b]$.
   - The analogous of Proposition 4.3.5 for Chebyshev nodes of first kind:
   Proposition 4.4.1. *Chebyshev interpolation formulas*
   Let $p_{1,n}(x) = \sum_{i=0}^{n}{}' c_{1,i} T_i^{[a,b]}(x)$ *be the degree* n *polynomial interpolating* f *on* I *in the set* $\{\mu_k^{[a,b]}, k = 0, \ldots, n\}$. *The coefficients* $c_{1,i}$ *are given by the explicit formula:*
   $$c_{1,i} = \frac{2}{n+1}\sum_{k=0}^{n} f(\mu_k^{[a,b]}) T_i^{[a,b]}(\mu_k^{[a,b]}) \tag{4.43}$$
   $$= \frac{2}{n+1}\sum_{k=0}^{n} f(\mu_k^{[a,b]}) T_i(\mu_k). \tag{4.44}$$

– The analogous of Proposition 4.3.5 for Chebyshev nodes of second kind:
Proposition 4.4.2. *Chebyshev interpolation formulas*
Let $p_{2,n}(x) = \sum_{i=0}^{n}{}'' c_{2,i}T_i(x)$ *be the degree* n *polynomial interpolating* f *in the set* $\{\nu_k^{[a,b]}, k = 0, \ldots, n\}$. *The coefficients* $c_{2,i}$ *are given by the explicit formula:*

$$c_{2,i} = \frac{2}{n}\sum_{k=0}^{n} f(\nu_k^{[a,b]})T_i^{[a,b]}(\nu_k^{[a,b]}); \tag{4.45}$$

$$= \frac{2}{n}\sum_{k=0}^{n} f(\mu_k^{[a,b]})T_i(\mu_k). \tag{4.46}$$

3. The norm over $[a, b]$: $\left\|\prod_{i=0}^{n}(x - \mu_i^{[a,b]})\right\|_\infty = \frac{(b-a)^{n+1}}{2^{2n+1}}$.
4. Interpolation error
   – for Chebyshev nodes of first kind over $[a, b]$:

$$\|f - p_{1,n}\|_\infty = \frac{(b-a)^{n+1}\left|f^{(n+1)}(\xi)\right|}{2^{2n+1}(n+1)!}, \xi \in (a, b); \tag{4.47}$$

   – for Chebyshev nodes of second kind:

$$\|f - p_{2,n}\|_\infty = \frac{(b-a)^{n+1}\left|f^{(n+1)}(\xi)\right|}{2^{2n}(n+1)!}, \xi \in (a, b). \tag{4.48}$$

5. Over $[a, b]$, Theorem 4.2.28 which bounds the error for truncated Chebyshev series is similarly stated:
   Theorem 4.4.3. *Let* $n \in \mathbb{N}$ *fixed and a function* $f \in \mathcal{C}^{n+1}[a, b]$. *Let* f_n *the degree* n *truncated Chebyshev series of* f *on* $[a, b]$. *Then there exists* $\xi \in (a, b)$ *such that:*

$$\|f - f_n\|_\infty = \frac{(b-a)^{n+1}\left|f^{(n+1)}(\xi)\right|}{2^{2n+1}(n+1)!}. \tag{4.49}$$

4.5 Chebyshev Models

Section 4.2 shows that it is of interest to use approximate Chebyshev series, which are of similar quality to the minimax polynomials, but can be obtained much easier. Moreover, closed formulas, hence explicit bounds, for the approximation error exist, cf. (4.47), (4.48) and (4.49). These equations induce that for an approximation polynomial of degree n, roughly speaking, compared to a Taylor error bound, the approximation error will be scaled down by a factor of 2^{1-n}.

We showed in Section 1.4.3 that, in practice, it is significantly more suitable to use a two-step procedure for handling composite functions: first step consists in computing models $(P, \mathbf{\Delta})$ for all basic functions; second, apply algebraic rules specifically designed for handling operations with these mixed models instead of operations with the corresponding functions.

Roughly speaking, we consider a Chebyshev Model (CM) for a function f as a couple $(P, \mathbf{\Delta})$ where P is a polynomial represented in Chebyshev basis whose coefficients are *tight intervals* "close" to the coefficients of the Chebyshev series of f, and $\mathbf{\Delta}$ is an interval bounding the absolute error between P and f. This name implicitly suggests that the method used is based in this case on Chebyshev Series approximations – either truncated Chebyshev series (TCS) or Chebyshev interpolants (CI) which we remark in the following:

Remark 4.5.1. *Each time we refer in this work to Chebyshev models, it is implicit that we discuss about an RPA based on:*
- *using either truncated Chebyshev series or Chebyshev interpolants for basic functions;*
- *recursively applying algebraic rules with CMs on the structure of composite functions.*

In what follows, we consider an interval $\boldsymbol{I} = [a, b]$, $a \neq b$ whose endpoints a and b are exactly representable in the number format used by IA. For example, a and b are two floating-point numbers.

Definition 4.5.2 (Chebyshev Model)**.** *Let $f : \boldsymbol{I} \to \mathbb{R}$ be a function. Let $M = (\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta})$ be a RPA structure. We say that M is a Chebyshev Model of f over $\boldsymbol{I}$ when*

$$\exists \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta}, f(x) - \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) = \delta.$$

Remark 4.5.3. *While the only "obvious" difference to other presented RPAs is the representation basis, it is important to have in mind Remark 4.5.1, which states that it is the way we construct this RPA that is different: we are using Chebyshev series which are naturally expressed in Chebyshev basis.*

We compute in this case also a polynomial with *tight interval coefficients*, but which is represented in Chebyshev basis $\left(T_i^{[a,b]}\right)_{n \geqslant 0}$. When using this approach, one needs:
- to compute bounds for such a polynomial over the whole interval $[a, b]$;
- to compute bounds for such a polynomial over tighter intervals, which can be also point intervals;
- to efficiently compute from it a polynomial with floating-point coefficients and a new rigorous error bound.

This is explained in what follows. We also need an algorithm $\texttt{eval}(f, \boldsymbol{x})$ which, for a function f and an interval $\boldsymbol{x}$, returns a valid bound for f over $\boldsymbol{x}$ as explained in Remark 1.3.20. For simplicity, if f is a constant function, then $\boldsymbol{x}$ can be omitted.

Bounding polynomials with interval coefficients in Chebyshev Basis

Since we are using Chebyshev basis, we need to evaluate or bound a polynomial with *tight interval coefficients*, written in this basis. We give a formal definition for a valid polynomial bound and discuss methods for obtaining such valid bounds.

Definition 4.5.4 (Valid polynomial bound)**.** *Let $[a, b]$ an interval, $a \neq b$ and a, b exactly representable in the numerical format chosen. Let $p(x) = \sum_{i=0}^{n} p_i T_i^{[a,b]}(x)$, $x \in [a, b]$ such that $p_i \in \boldsymbol{a_i}$, for $i = 0, \ldots, n$, i.e. $\boldsymbol{a_i}$ are intervals around polynomial coefficients. Let $\boldsymbol{x} \subseteq [a, b]$ be an interval over which p is to be bounded.*

An interval $\boldsymbol{B}$ is called a valid polynomial bound when

$$\forall \xi \in \boldsymbol{x}, \forall \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \sum_{i=0}^{n} \alpha_i \, T_i^{[a,b]}(\xi) \in \boldsymbol{B}.$$

Several methods exist and a trade-off between their speed and the tightness of the bound is usually considered. For TMs, where monomial basis is used, the fastest but "rough" method was a Horner-like interval evaluation (see Algorithm 2.1.1 and Remark 2.1.7). In this case we focus again on computation speed, and so, when considering interval range bounding for polynomials in Chebyshev basis we used the straightforward adaptation to interval arithmetic of two simple methods described in Section 4.3.1 which are linear in the number of operations used. The first one, Algorithm 4.5.1, uses natural interval extension of the polynomial expression written using Clenshaw's scheme (which is the Chebyshev analogue of Horner's scheme for monomials).

Algorithm: `EvaluateClenshawIA`$(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x}, [a, b])$
Input: $[a, b]$ an interval, $a \neq b$,
$\boldsymbol{x} \subseteq [a, b]$,
$\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ interval coefficients of $P(\boldsymbol{x}) = \sum_{i=0}^{n} \boldsymbol{a_i} T_i^{[a,b]}(\boldsymbol{x})$.
Output: interval bound of $P(\boldsymbol{x})$.
$\boldsymbol{b_{n+1}} \leftarrow [0, 0]$;
$\boldsymbol{b_{n+2}} \leftarrow [0, 0]$;
for $k \leftarrow n$ **downto** 0 **do**
  $\boldsymbol{b_k} \leftarrow \boldsymbol{a_k} + \texttt{eval}\left(\frac{2(2x-b-a)}{b-a}, \boldsymbol{x}\right) \boldsymbol{b_{k+1}} - \boldsymbol{b_{k+2}}$;
end
return $\boldsymbol{b_0} - \texttt{eval}\left(\frac{2(2x-b-a)}{b-a}, \boldsymbol{x}\right) \boldsymbol{b_1}$;

Algorithm 4.5.1: Interval Arithmetic version of Clenshaw's algorithm for evaluating Chebyshev sums

Proof of Algorithm 4.5.1. Based on the fact that Algorithm 4.3.1 is correct and that we simply use the extension to interval arithmetic of all computations. □

Algorithm 4.5.1 computes a valid bound over any interval $\boldsymbol{x} \subseteq [a, b]$. We give another algorithm that computes directly a valid bound over the whole interval $[a, b]$. It is based on Remark 4.3.9 and Equation (4.38).

Algorithm: `EvaluateChebyshevRangeIA`$(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{x}, [a, b])$
Input: $[a, b]$ an interval, $a \neq b$,
$\boldsymbol{x} \subseteq [a, b]$,
$\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}$ interval coefficients of $P(\boldsymbol{x}) = \sum_{i=0}^{n} \boldsymbol{a_i} T_i^{[a,b]}(\boldsymbol{x})$.
Output: interval bound of $P(\boldsymbol{x})$.
$\boldsymbol{B} \leftarrow \boldsymbol{a_0}$;
for $k \leftarrow 1$ **to** n **do**
  $\boldsymbol{B} \leftarrow \boldsymbol{B} + \boldsymbol{a_k} \cdot [-1, 1]$;
end
return $\boldsymbol{B}$;

Algorithm 4.5.2: Interval Arithmetic Range Evaluation of Chebyshev sums

Proof of Algorithm 4.5.2. From Remark 4.3.9 and Equation (4.38) we have for all $i \geqslant 1$, $x \in [a, b]$, $T_i^{[a,b]}(x) \in [-1, 1]$, hence if we want to bound $T_i^{[a,b]}(x)$ over an interval $\boldsymbol{x} \subseteq [a, b]$ we can bound it by $[-1, 1]$. This implies that using interval arithmetic $P(\boldsymbol{x}) = \sum_{i=0}^{n} \boldsymbol{a_i} T_i^{[a,b]}(\boldsymbol{x}) \subseteq \boldsymbol{a_0} + \sum_{i=1}^{n} \boldsymbol{a_i} \cdot [-1, 1]$. □

Both algorithms have linear complexity. Of course, similarly to TMs, for a penalty in speed, more refined algorithms can also be plugged-in. However the simple techniques we used, proved effective in most of the examples we treated so far.

Finally, we can compute from a CM consisting of a polynomial with *tight interval coefficients* and an *approximation error bound* as defined in 4.5.2, a polynomial with floating-point coefficients and a new rigorous error bound, by taking a FP number contained in each interval coefficient (for example the middle of the interval, if it is a FP number, of course) as the new coefficient and accumulating the small errors in the coefficients in the final error bound.

```
Algorithm: ComputeFP-RPAfromCM((a_0, ..., a_n, Δ), [a, b])
Input: [a, b] an interval, a ≠ b,
(a_0, ..., a_n, Δ) a CM of degree n over [a, b] for some function.
Output: an RPA whose polynomial has FP coefficients
for k ← 0 to n do
    t_i = mid (a_i);
    if t_i is not a FP number then
        t_i = inf(a_i);
    end
    b_i = a_i − t_i;
end
θ ← EvaluateChebyshevRangeIA(b_0, ..., b_n, [a, b], [a, b]);
Δ ← Δ + θ;
return (t_0, ..., t_n, Δ) ;
```

Algorithm 4.5.3: Computation of a polynomial with FP coefficients and a rigorous error bound from a CM.

Proof of Algorithm 4.5.3. Let us set $t_i \in \boldsymbol{a_i}$, one common choice is $t_i = \operatorname{mid}(\boldsymbol{a_i})$, if this is not a floating-point number, then $\inf(\boldsymbol{a_i})$ can be taken, for example. Let $\widetilde{P}(x)$ be a polynomial of degree n,

$$\widetilde{P}(x) = \sum_{i=0}^{n} t_i T_i^{[a,b]}(x).$$

We have from Definition 4.5.2:

$$\exists \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \forall x \in [a,b], \exists \delta \in \boldsymbol{\Delta}, f(x) - \widetilde{P}(x) = \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) - \widetilde{P}(x) + \delta.$$

The difference between $\sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x)$ and $\widetilde{P}(x)$ can easily be bounded using Equation (4.38) and interval arithmetic. We have $\alpha_i \in \boldsymbol{a_i}$, and thus $\alpha_i - t_i \in [\inf(\boldsymbol{a_i}) - t_i, \sup(\boldsymbol{a_i}) - t_i]$, which leads to

$$\forall x \in [a,b], \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n},$$
$$\sum_{i=0}^{n} (\alpha_i - t_i) T_i^{[a,b]}(x) \in [\inf(\boldsymbol{a_0}) - t_0, \sup(\boldsymbol{a_0}) - t_0] + \sum_{i=1}^{n} [\inf(\boldsymbol{a_i}) - t_i, \sup(\boldsymbol{a_i}) - t_i] \cdot [-1, 1] = \boldsymbol{\theta}.$$

Finally, the error between f and $\widetilde{P}$ is bounded by $\boldsymbol{\theta} + \boldsymbol{\Delta}$. □

4.5.1 Chebyshev Models for basic functions

We now give algorithms for computing Chebyshev Models for basic functions. We write a special procedure for each basic function f like identity, $\sin, \cos, \exp$, etc. We tune case by case the

best choice among using Chebyshev interpolators (CI) or truncated Chebyshev series (TCS). We give below the example of constant function, identity function and then we describe the process to follow for other functions. We exemplify on $\sin$ for using CI.

Algorithm: $\texttt{CMConst}(\boldsymbol{c}, n)$
Input: a (usually small) interval $\boldsymbol{c}$, $n \in \mathbb{N}^\star$
Output: a Chebyshev Model for $x \mapsto \gamma$ for any $\gamma \in \boldsymbol{c}$
$\boldsymbol{a_0} \leftarrow \boldsymbol{c}$;
$\boldsymbol{a_1} \dots \boldsymbol{a_n} \leftarrow [0, 0]$;
$\boldsymbol{\Delta} \leftarrow [0, 0]$;
$M \leftarrow (\boldsymbol{a_0}, \dots, \boldsymbol{a_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 4.5.4: Computation of a Chebyshev Model of a constant function

Constant function

Proof of Algorithm 4.5.4. Similar to that of Algorithm 2.2.1. □

Algorithm: $\texttt{CMVar}(\boldsymbol{I}, n)$
Input: $n \in \mathbb{N}^\star$
Output: a Chebyshev Model M of the identity function $x \mapsto x$
$\boldsymbol{a_0} \leftarrow [0, 0]$;
$\boldsymbol{a_1} \leftarrow [1, 1]$;
$\boldsymbol{a_2} \dots \boldsymbol{a_n} \leftarrow [0, 0]$;
$\boldsymbol{\Delta} \leftarrow [0, 0]$;
$M \leftarrow (\boldsymbol{a_0}, \dots, \boldsymbol{a_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 4.5.5: Computation of a Chebyshev Model of the identity function

Identity function

Proof of Algorithm 4.5.5. It suffices to notice that $T_1(x) = x$ and to choose $\alpha_1 = 1$. □

Computing the coefficients and the error bound for basic functions. We have two ways of computing Chebyshev models for basic functions:

- We use truncated Chebyshev series for functions whose Chebyshev coefficients have a simple explicit form (e.g. $\arccos$, see Example 4.2.13) when this can be obtained as explained in Section 4.2.2.
- We use Chebyshev interpolants described in Section 4.3. The summary of formulas used is given in Section 4.4. Both kinds of Chebyshev nodes can be used. For simplicity we chose to give the algorithms using Chebyshev nodes of first kind. We also use like in the case of Taylor Models an adaptation of Lemma 5.12 of [176]. Lemma 4.3.1 gives us a way to obtain an exact error bound for Chebyshev interpolants in some cases:

Proposition 4.5.5 (Adaptation of Lemma 5.12 of [176]). *Let $f \in \mathcal{C}^{n+2}[a,b]$ the function to be approximated over an interval $\boldsymbol{I} = [a,\, b]$; let $n \geq 1$ be an integer, $p_{1,n}$ be the interpolant of f at Chebyshev nodes of first kind and $I_n(x) = \dfrac{f(x) - p_{1,n}(x)}{T_{n+1}^{[a,b]}(x)}$.*
If the sign of $f^{(n+2)}$ is constant over $\boldsymbol{I}$, then $I_n(x)$ is monotonic over $[a,\, b]$ and letting $B = \max\{|f(a) - p_{1,n}(a)|,\, |f(b) - p_{1,n}(b)|\}$, we have : $f(x) - p_{1,n}(x) \in [-B, B]$.

Proof. If $f^{(n+2)}$ has constant sign over $\boldsymbol{I}$, then $f^{(n+1)}$ is monotonic over $\boldsymbol{I}$, so Lemma 4.3.1 implies that $I_n(x)$ is monotonic over $[a,\, b]$. It follows that we can bound it by the tightest interval that contains $I_n(a)$ and $I_n(b)$. Furthermore, observing that $T_{n+1}^{[a,b]}(a) = T_{n+1}(-1) = (-1)^{n+1}$, $T_{n+1}^{[a,b]}(b) = T_{n+1}(1) = 1$ and $T_{n+1}^{[a,b]}([a,b]) = [-1,1]$ and letting $B = \max\{|f(a) - p_{1,n}(a)|,\, |f(b) - p_{1,n}(b)|\}$, we have : $f(x) - p_{1,n}(x) \in [-B, B]$. □

Remark 4.5.6. *Automatically checking whether $f^{(n+2)}$ has constant sign over $[a,b]$ is simple to achieve for basic functions using interval arithmetic evaluation: if the resulting interval contains numbers of different signs, we cannot apply Prop. 4.5.5 and fall back to the classical bound given by Equation (4.47). Otherwise, we can obtain the "tightest" error bound for the error $f(x) - p_{1,n}(x)$ using four simple evaluations $f(a), f(b), p_{1,n}(a), p_{1,n}(b)$.*
This remark makes it possible to obtain smaller error bounds and strengthens the effectiveness of the "basic bricks" approach. Examples will be given in Section 4.6.

Let us now describe and prove the algorithm used for computing a Chebyshev Model for $\sin$.

The $\sin$ function

Proof of `CMSin`. In lines $4 - 11$ of the algorithm, enclosures of the coefficients c_k given by interpolation formula (4.43) are computed, so for all $k = 0, \ldots, n$, we have $c_k \in \boldsymbol{c_k}$. In lines $12 - 17$ an interval enclosure of the $n+2$ derivative of sine is computed. Based on its sign, Proposition 4.5.5 applies or not. If it does not apply, we compute an enclosure of the error given in formula (4.47) in a similar manner to what we have seen in the Taylor Models algorithms.

If, on the contrary, the hypothesis of Remark 4.5.6 holds, we compute an enclosure of the error based on Proposition 4.5.5 in lines $18 - 24$. Obviously, we supposed that `EvaluateClenshawIA` computes a valid polynomial bound. □

Remark 4.5.7. *The complexity of the algorithm is $\mathcal{O}(n^2)$ operations. It is known [131] that the usage of Fast Fourier Transform, can speed-up this computation of the coefficients to $O(n \log n)$ operations. However, note that in this case, an interval arithmetic adaptation of FFT should be considered. It is a future work to use [163] for that. The correction of the present algorithm should not be affected as long as enclosures of the real coefficients of the interpolant are computed.*

Reciprocal function $x \mapsto 1/x$. Similarly to Taylor Models, here, divisions are performed also by computing a reciprocal followed by a multiplication. The reciprocal $x \mapsto 1/x$ is handled like any other unary basic function.

4.5.2 Operations with Chebyshev models

For an informal simplified description of these operations, we consider two couples *(polynomial, error bound)* for two functions f_1 and f_2, over the interval $[a,b]$, of degree n: $(\boldsymbol{P_1}, \boldsymbol{\Delta}_1)$ and $(\boldsymbol{P_2}, \boldsymbol{\Delta}_2)$. Similarly to TM arithmetic, we discuss first informally the operations defined on these couples. Then we give and prove the algorithms in detail.

Algorithm: $\texttt{CMSin}(\boldsymbol{I}, n)$
Input: $\boldsymbol{I} = [a, b]$ an interval, $a \neq b$, $n \in \mathbb{N}^\star$
Output: a Chebyshev Model M of the $\sin$ function in the basis $\left(T_i^{[a,b]}\right)_{i \geqslant 0}$

$a \leftarrow [\inf(\boldsymbol{I})]$;
$b \leftarrow [\sup(\boldsymbol{I})]$;
$\boldsymbol{a} \leftarrow [a, a]$;
$\boldsymbol{b} \leftarrow [b, b]$;
for $i \leftarrow 0$ **to** n **do**
  $\boldsymbol{\mu_i} \leftarrow \texttt{eval}\left(\cos\left(\frac{(i+1/2)\,x}{n+1}\right), \texttt{eval}(\pi)\right)$;
  $\boldsymbol{\mu_i^{[a,b]}} \leftarrow \dfrac{\boldsymbol{a}+\boldsymbol{b}}{2} + \dfrac{\boldsymbol{b}-\boldsymbol{a}}{2}\,\boldsymbol{\mu_i}, [-1, 1]$;
  $\boldsymbol{f_i} \leftarrow \texttt{eval}\left(\sin, \boldsymbol{\mu_i^{[a,b]}}\right)$;
end
for $k \leftarrow 0$ **to** n **do**
  $\boldsymbol{c_k} \leftarrow \texttt{EvaluateClenshawIA}(\boldsymbol{f_0}, \ldots, \boldsymbol{f_n}, \boldsymbol{\mu_k})$;
end
switch $(n+2) \bmod 4$ **do**
  case *0:* $\boldsymbol{\Gamma} \leftarrow \texttt{eval}\,(\sin(x)/i!,\ \boldsymbol{I})$;
  case *1:* $\boldsymbol{\Gamma} \leftarrow \texttt{eval}\,(\cos(x)/i!,\ \boldsymbol{I})$;
  case *2:* $\boldsymbol{\Gamma} \leftarrow \texttt{eval}\,(-\sin(x)/i!,\ \boldsymbol{I})$;
  case *3:* $\boldsymbol{\Gamma} \leftarrow \texttt{eval}\,(-\cos(x)/i!,\ \boldsymbol{I})$;
endsw
if $(\sup(\boldsymbol{\Gamma}) \leq 0)$ **or** $(\inf(\boldsymbol{\Gamma}) \geq 0)$ **then**
  $\boldsymbol{\Delta_a} \leftarrow \texttt{eval}\,(\sin(x), \boldsymbol{a}) - \texttt{EvaluateClenshawIA}(\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}, \boldsymbol{a}, [a, b])$;
  $\boldsymbol{\Delta_b} \leftarrow \texttt{eval}\,(\sin(x), \boldsymbol{b}) - \texttt{EvaluateClenshawIA}(\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}, \boldsymbol{b}, [a, b])$;
  $B \leftarrow \max(|\sup(\boldsymbol{\Delta_a})|, |\inf(\boldsymbol{\Delta_a})|, |\sup(\boldsymbol{\Delta_b})|, |\inf(\boldsymbol{\Delta_b})|)$;
  $\boldsymbol{\Delta} \leftarrow [-B, B]$;
else
  switch $(n+1) \bmod 4$ **do**
    case *0:* $\boldsymbol{\Gamma} \leftarrow \texttt{eval}\,(\sin(x)/i!,\ \boldsymbol{I})$;
    case *1:* $\boldsymbol{\Gamma} \leftarrow \texttt{eval}\,(\cos(x)/i!,\ \boldsymbol{I})$;
    case *2:* $\boldsymbol{\Gamma} \leftarrow \texttt{eval}\,(-\sin(x)/i!,\ \boldsymbol{I})$;
    case *3:* $\boldsymbol{\Gamma} \leftarrow \texttt{eval}\,(-\cos(x)/i!,\ \boldsymbol{I})$;
  endsw
  $\boldsymbol{\Delta} \leftarrow [-|\sup(\boldsymbol{\Gamma})|, |\sup(\boldsymbol{\Gamma})|]$;
  $\boldsymbol{V} \leftarrow \texttt{eval}\left((x-y)^{n+1}/2^{2n+1}, (\boldsymbol{b}, \boldsymbol{a})\right)$;
  $\boldsymbol{\Delta} \leftarrow \boldsymbol{V} \cdot \boldsymbol{\Delta}$;
end
$M \leftarrow (\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 4.5.6: Computation of a Chebyshev Model of the sine function

4.5.3 Addition

Addition of two such models is done straightforwardly by adding the two polynomials and the remainder bounds as:

$$(\boldsymbol{P_1}, \boldsymbol{\Delta}_1) + (\boldsymbol{P_2}, \boldsymbol{\Delta}_2) = (\boldsymbol{P_1} + \boldsymbol{P_2}, \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2).$$

It is obvious that since $f_1(x) \in \boldsymbol{P_1} + \boldsymbol{\Delta}_1$ and $f_2(x) \in \boldsymbol{P_2} + \boldsymbol{\Delta}_2$, $f_1(x) + f_2(x) \in \boldsymbol{P_1} + \boldsymbol{P_2} + \boldsymbol{\Delta}_1 + \boldsymbol{\Delta}_2$.

The algorithm and its proof are given below.

Algorithm: `CMAdd`(M_f, M_g, n)
Input: M_f, M_g two Chebyshev Models corresponding to two functions f and g (over the same interval $\boldsymbol{I} = [a, b]$),
$n \in \mathbb{N}^\star$ the common degree of polynomials involved in M_f and M_g
Output: a Chebyshev Model M corresponding to $f + g$
$(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta_f}) \leftarrow M_f$;
$(\boldsymbol{b_0}, \ldots, \boldsymbol{b_n}, \boldsymbol{\Delta_g}) \leftarrow M_g$;
for $i \leftarrow 0$ **to** n **do**
  $\boldsymbol{c_i} \leftarrow \boldsymbol{a_i} + \boldsymbol{b_i}$;
end
$\boldsymbol{\Delta} \leftarrow \boldsymbol{\Delta_f} + \boldsymbol{\Delta_g}$;
$M \leftarrow (\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 4.5.7: Addition of Chebyshev Models

Proof of `CMAdd`. Since M_f is a Chebyshev model of f, we have that

$$\exists \alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}, \forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) - \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) = \delta_f.$$

We consider such values α_i. The same holds for M_g and g with some $\beta_i \in \boldsymbol{b_i}$. We shall show

$$\exists \gamma_0 \in \boldsymbol{c_0}, \ldots, \gamma_n \in \boldsymbol{c_n}, \forall x \in \boldsymbol{I}, \exists \delta \in \boldsymbol{\Delta},\ (f(x) + g(x)) - \sum_{i=0}^{n} \gamma_i T_i^{[a,b]}(x) = \delta.$$

We choose $\gamma_i = \alpha_i + \beta_i \in \boldsymbol{c_i}$ and $\delta = \delta_f + \delta_g \in \boldsymbol{\Delta}$. Assume that $x \in \boldsymbol{I}$. We have

$$\begin{aligned} f(x) + g(x) - \sum_{i=0}^{n} \gamma_i T_i^{[a,b]}(x) &= \left(f(x) - \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) \right) + \left(g(x) - \sum_{i=0}^{n} \beta_i T_i^{[a,b]}(x) \right) \\ &= \delta_f + \delta_g = \delta. \end{aligned}$$

□

Remark 4.5.8. *The algorithm has a linear complexity.*

4.5.4 Multiplication

Consider $f_1(x) \in \boldsymbol{P_1} + \boldsymbol{\Delta}_1$ and $f_2(x) \in \boldsymbol{P_2} + \boldsymbol{\Delta}_2$. We have

$$f_1(x) \cdot f_2(x) \in \boldsymbol{P_1} \cdot \boldsymbol{P_2} + \boldsymbol{P_2} \cdot \boldsymbol{\Delta}_1 + \boldsymbol{P_1} \cdot \boldsymbol{\Delta}_2 + \boldsymbol{\Delta}_1 \cdot \boldsymbol{\Delta}_2.$$

We observe that $\boldsymbol{P_1} \cdot \boldsymbol{P_2}$ is a polynomial of degree $2n$. Depending on the basis used, we split it into two parts: the polynomial consisting of the terms that "do not exceed n", $(\boldsymbol{P_1} \cdot \boldsymbol{P_2})_{0\ldots n}$ and respectively the upper part $(\boldsymbol{P_1} \cdot \boldsymbol{P_2})_{n+1\ldots 2n}$, for the terms of the product $\boldsymbol{P_1} \cdot \boldsymbol{P_2}$ whose "order exceeds n".

Now, a CM for $f_1 \cdot f_2$ can be obtained by finding an interval bound for all the terms except $\boldsymbol{P} = (\boldsymbol{P_1} \cdot \boldsymbol{P_2})_{0\ldots n}$. Hence we have,

$$\boldsymbol{\Delta} = B((\boldsymbol{P_1} \cdot \boldsymbol{P_2})_{n+1\ldots 2n}) + B(\boldsymbol{P_2}) \cdot \boldsymbol{\Delta}_1 + B(\boldsymbol{P_1}) \cdot \boldsymbol{\Delta}_2 + \boldsymbol{\Delta}_1 \cdot \boldsymbol{\Delta}_2.$$

The interval bound for the polynomials involved can be computed as discussed in 4.5.

```
Algorithm: CMMul(M_f, M_g, I, x_0, n)
Input: M_f, M_g two Chebyshev Models corresponding to two functions f and g (over the
       same interval I = [a, b],
       n ∈ ℕ* the common degree of polynomials involved in M_f and M_g
Output: a Chebyshev Model M corresponding to f · g
(a_0, ..., a_n, Δ_f) ← M_f;
(b_0, ..., b_n, Δ_g) ← M_g;
for k ← 0 to 2n do
    c_k ← [0; 0];
end
for i ← 0 to n do
    for j ← 0 to n do
        c_{i+j} ← (c_{i+j} + a_i · b_j)/2;
        c_{|i−j|} ← (c_{|i−j|} + a_i · b_j)/2;
    end
end
for k ← 0 to n do
    d_k ← [0; 0];
end
for k ← n + 1 to 2n do
    d_k ← c_k;
end
B ← EvaluateChebyshevRangeIA(d_0, ..., d_{2n}, [a, b], [a, b]);
B_f ← EvaluateChebyshevRangeIA(a_0, ..., a_n, [a, b], [a, b]);
B_g ← EvaluateChebyshevRangeIA(b_0, ..., b_n, [a, b], [a, b]);
Δ ← B + (Δ_f · B_g) + (Δ_g · B_f) + (Δ_f · Δ_g);
M ← (c_0, ..., c_n, Δ);
return M ;
```

Algorithm 4.5.8: Multiplication of Chebyshev Models

Proof of `CMMul`. We assume that M_f and M_g are Chebyshev Models of two functions f and g, over the same interval $\boldsymbol{I}$. We shall prove that `CMMul` returns a Chebyshev Model of fg over $\boldsymbol{I}$.

By definition, since M_f is a Chebyshev model of f, we have $\alpha_i \in \boldsymbol{a_i}$ ($i = 0, \ldots, n$) such that

$$\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) - \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) = \delta_f.$$

The same holds for M_g and g with $\beta_i \in \boldsymbol{b_i}$. We have

$$\left(\sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x)\right) \cdot \left(\sum_{i=0}^{n} \beta_i T_i^{[a,b]}(x)\right) = \sum_{i=0}^{n}\sum_{j=0}^{n} \frac{\alpha_i\,\beta_j}{2}\left(T_{i+j}^{[a,b]} + T_{|i-j|}^{[a,b]}\right) = \sum_{k=0}^{2n} \gamma_k\, T_k^{[a,b]}(x).$$

Hence we have $\gamma_k \in \boldsymbol{c_k}$ for all k. Assume now that $x \in \boldsymbol{I}$. We have values δ_f and δ_g such that

$$\begin{aligned} f(x)\,g(x) &= \left(\sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) + \delta_f\right) \cdot \left(\sum_{i=0}^{n} \beta_i T_i^{[a,b]}(x) + \delta_g\right) \\ &= \sum_{i=0}^{n} \gamma_i T_i^{[a,b]}(x) + \underbrace{\sum_{i=0}^{n} 0\, T_i^{[a,b]} + \sum_{i=n+1}^{2n} \gamma_i T_i^{[a,b]}(x)}_{\in \boldsymbol{B}} \\ &+ \delta_f \underbrace{\left(\sum_{i=0}^{n} \beta_i T_i^{[a,b]}(x)\right)}_{\in \boldsymbol{B_g}} + \delta_g \underbrace{\left(\sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x)\right)}_{\in \boldsymbol{B_f}} + \delta_f\,\delta_g. \end{aligned}$$

The inclusions in $\boldsymbol{B}$, $\boldsymbol{B_f}$ and $\boldsymbol{B_g}$ are given by the correctness of `EvaluateChebyshevRangeIA`. The conclusion follows. □

In this current setting the number of operations necessary to multiply two such models is $O(n^2)$.

4.5.5 Composition

Let f_1 and f_2 be two basic functions. When the model for $f_1 \circ f_2$ is needed, we can consider $(f_1 \circ f_2)(x)$ as function f_1 evaluated at point $y = f_2(x)$. Hence, we have to take into account the additional constraint that the image of f_2 has to be included in the definition range of f_1. This can be checked by a simple interval bound computation of $B(\boldsymbol{P_2}) + \boldsymbol{\Delta_2}$. Then we have:

$$(f_1 \circ f_2)(x) \in \boldsymbol{P_1}(f_2(x)) + \boldsymbol{\Delta_1} \in \boldsymbol{P_1}(\boldsymbol{P_2}(x) + \boldsymbol{\Delta_2}) + \boldsymbol{\Delta_1} \tag{4.50}$$

In this formula, the only polynomial coefficients and remainders involved are those of the CMs of f_1 and f_2 which are basic functions. As we have seen above, fairly simple formulæ exist for computing the coefficients and remainders of such functions. However, when using formula (4.50), it is not obvious how to extract a polynomial and a final remainder bound from it. In fact, we have to reduce this extraction process to performing just multiplications and additions of CMs. A similar idea is used for composing TMs.

In our case, the difference is that $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ are polynomials represented in Chebyshev basis, and not in the monomial basis. In consequence, for computing the composition, we had to use a different algorithm. It is worth mentioning that a simple change of basis back and forth to monomial basis will not be successful. The problem is that the multiplications and additions used in such a composition process do not have to add too much overestimation to the final remainder. As we discussed in Section 4.2.3, for Taylor expansions of most of the functions we address, the

size of the coefficients for the representation in the monomial basis is bounded by a decreasing sequence (see Theorem 4.2.16). Hence the contributions of the remainders in such a recursive algorithm are smaller and smaller. On the contrary, for interpolation polynomials, the coefficients represented in monomial basis oscillate too much and have poor numerical properties. Hence, a direct application of the principle of composing TMs will not be successful.

When using Chebyshev basis, we perform the composition using an adaptation of Clenshaw Algorithm 4.3.1. This algorithm is used for efficient evaluation of a Chebyshev sum $\sum p_i T_i^{[a,b]}(x)$. In our case, the variable x where the sum is to be evaluated is a CM, the multiplications and additions are operations between CMs. Moreover, using this algorithm, we perform a linear number of such operations between models.

Algorithm: `PolynomialEvaluationOfCM`($\boldsymbol{b_0}$, ..., $\boldsymbol{b_n}$, $[c,d]$, M_f, $[a,b]$, n)
Input: $n \in \mathbb{N}^\star$;
$\boldsymbol{b_0}$, ..., $\boldsymbol{b_n}$ tight intervals;
$[c,d]$, $[a,b]$ intervals, $a \neq b$, $c \neq d$;
$M_f = (\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta}_f)$ a Chebyshev Model of a function f over $[a,b]$ such that $\sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) + \delta_f \in [c,d]$, for any $x \in [a,b]$ and $(\alpha_0, \ldots, \alpha_n, \delta_f)$ such that $\alpha_i \in \boldsymbol{a_i}$ for each i and $\delta_f \in \boldsymbol{\Delta}_f$.
Output: a Chebyshev Model of $x \mapsto \sum_{i=0}^{n} \beta_i T_i^{[c,d]}(f(x))$ over $[a,b]$, for any $(\beta_0, \ldots, \beta_n)$ such that $\beta_i \in \boldsymbol{b_i}$ for each i.
$M_{n+1} \leftarrow ([0;0], \ldots, [0;0], [0;0])$;
$M_{n+2} \leftarrow ([0;0], \ldots, [0;0], [0;0])$;
$\boldsymbol{c_1} \leftarrow \texttt{eval}\left(-\frac{2(c+d)}{d-c}\right)$;
$\boldsymbol{c_2} \leftarrow \texttt{eval}\left(\frac{4}{d-c}\right)$;
$N \leftarrow \texttt{CMAdd}(\texttt{CMMul}(M_f, \texttt{CMConst}(\boldsymbol{c_2}, n), [a,b], n), \texttt{CMConst}(\boldsymbol{c_1}, n), n)$;
for $i \leftarrow n$ **downto** 0 **do**
  $M_i \leftarrow \texttt{CMMul}(N, M_{i+1}, [a,b], n)$;
  $M_i \leftarrow \texttt{CMAdd}(M_i, \texttt{CMMul}(M_{i+2}, \texttt{CMConst}([-1,-1], n), [a,b], n), n)$;
  $M_i \leftarrow \texttt{CMAdd}(M_i, \texttt{CMConst}(\boldsymbol{b_i}, n), n)$;
end
$M \leftarrow \texttt{CMMul}(N, M_1, [a,b], n)$;
$M \leftarrow \texttt{CMAdd}(M_0, \texttt{CMMul}(M, \texttt{CMConst}([-1,-1], n), [a,b], n), n)$;
return M ;

Algorithm 4.5.9: Composition of a polynomial with a Chebyshev Model

Proof of `PolynomialEvaluationOfCM`. The proof is the same as the proof of correctness of the algorithm `EvaluateClenshawIA` and based on the correctness of `CMConst`, `CMAdd` and `CMMul`. □

Proof of `CMComp`. Since M_f is a Chebyshev model of f, we have values $\alpha_0 \in \boldsymbol{a_0}, \ldots, \alpha_n \in \boldsymbol{a_n}$ such that

$$\forall x \in [a,b], \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) - \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) = \delta_f. \tag{4.51}$$

```
1  Algorithm: CMComp(M_f, [a, b], g, n)
   Input: M_f a Chebyshev Model corresponding to a function f over [a, b], a ≠ b;
   g : ℝ → ℝ a basic function;
   n ∈ ℕ*.
   Output: a Chebyshev Model M corresponding to g ∘ f over [a, b]
2  (a_0, ..., a_n, Δ_f) ← M_f;
3  B_f ← EvaluateChebyshevRangeIA(a_0, ..., a_n, [a, b], [a, b]);
4  [c, d] ← B_f + Δ_f;
5  switch g do
6      case sin: M_g ← CMSin([c, d], n);
7      case x ↦ 1/x: M_g ← CMInv([c, d], n);
8      ⋮
9  endsw
10 (b_0, ..., b_n, Δ_g) ← M_g;
11 (c_0, ..., c_n, Δ) ← PolynomialEvaluationOfCM(b_0, ..., b_n, [c, d], M_f, [a, b], n);
12 M ← (c_0, ..., c_n, Δ + Δ_g);
13 return M ;
```

Algorithm 4.5.10: Composition of a Chebyshev Model with a Function

Moreover, since M_f is defined over $[a, b]$ it is correct to call `EvaluateChebyshevRangeIA` over $[a, b]$ and hence, $[c, d]$ represents a valid bound for

$$f(x) = \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) + \delta_f \in [c, d]. \tag{4.52}$$

Furthermore, by correction of `CMSin`, `CMInv`, etc. M_g is a Chebyshev Model of g over $[c, d]$. So, we have values $\beta_0 \in \boldsymbol{b_0}, \ldots, \beta_n \in \boldsymbol{b_n}$ such that

$$\forall y \in [c, d], \exists \delta_g \in \boldsymbol{\Delta_g},\ g(y) - \sum_{i=0}^{n} \beta_i T_i^{[c,d]}(y) = \delta_g. \tag{4.53}$$

Based on Equations (4.52) and (4.53) we have

$$\forall x \in [a, b], \exists \delta_g \in \boldsymbol{\Delta_g},\ (g \circ f)(x) = \sum_{i=0}^{n} \beta_i T_i^{[c,d]}(f(x)) + \delta_g. \tag{4.54}$$

The call to `PolynomialEvaluationOfCM` is correct as per Equations (4.52) and based on its correction, we have that $(\boldsymbol{c_0},\ \ldots,\ \boldsymbol{c_n},\ \boldsymbol{\Delta})$ is a Chebyshev Model of $x \mapsto \sum_{i=0}^{n} \beta_i\, T_i^{[c,d]}(f(x))$ over $[a, b]$. Hence we have values $\gamma_i \in \boldsymbol{c_i}$ ($i \in [\![0,\ n]\!]$) such that

$$\forall x \in [a, b], \exists \delta \in \boldsymbol{\Delta},\ \left(\sum_{i=0}^{n} \beta_i\, T_i^{[c,d]}(f(x))\right) - \left(\sum_{i=0}^{n} \gamma_i\, T_i^{[a,b]}(x)\right) = \delta.$$

Combining this last result with Equation (4.54), we get

$$\forall x \in [a, b], \exists \delta \in \boldsymbol{\Delta} + \boldsymbol{\Delta_g},\ (g \circ f)(x) - \left(\sum_{i=0}^{n} \gamma_i\, T_i^{[a,b]}(x)\right) = \delta,$$

which is the property that we wanted to prove. □

Division of Chebyshev Models Computing a Chebyshev Model for $\frac{f}{g}$ reduces to computing $f \cdot \left(\frac{1}{x} \circ g\right)$. This implies multiplication and composition of Chebyshev Models as well as computing a Chebyshev Model for the basic function $x \mapsto 1/x$, which was already explained before in this section. We give below the algorithm for completeness.

1 **Algorithm:** `CMDiv`$(M_f, M_g, [a,b], n)$
Input: $n \in \mathbb{N}^\star$,
$[a,b]$ an interval, $a \neq b$,
M_f, M_g two Chebyshev Models of degree n corresponding to two functions f and g over $[a,b]$,
Output: a Chebyshev Model M corresponding to f/g
2 $M \leftarrow$ `CMMul`$(M_f,$ `CMComp`$(M_g, [a,b], x \mapsto 1/x, n), [a,b], n)$;
3 **return** M ;

Algorithm 4.5.11: Division of Chebyshev Models

Finally, we can write the complete algorithm that computes a Chebyshev Model for any expression, by induction on this expression.

Computing Chebyshev Models for any function given by an expression

Proof of `CM`. We use structural induction on the expression tree for function h and we have proven above the correction of the sub-algorithms used. □

Remark 4.5.9. *Growth of the coefficients and overestimation.*
The overestimation does not grow too much during the recursion process. This is due to the nice convergence properties already seen in Section 4.2.2 of the series expansions in Chebyshev polynomial basis. As with TMs, when composing two such models, the intervals contributing to the final remainder become smaller for higher coefficients, which yields a reduced overestimation in the final remainder.

Integral of a Chebyshev Model. Finally, we give a simple linear time algorithm for computing a CM for the integral of a function.

Proof of `CMIntegrate`. We assume the M_f is Chebyshev Model of f over $\boldsymbol{I}$. We shall prove that `CMIntegrate` returns a Chebyshev Model of $g(t,\xi_0) = \int_{\xi_0}^{t} f(x)\mathrm{d}x$ over $\boldsymbol{I}$, for all $\xi_0 \in \boldsymbol{x_0}$.

By definition, since M_f is a Chebyshev model of f, we have $\alpha_i \in \boldsymbol{a_i}$ $(i = 0, \ldots, n)$ such that

$$\forall x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ f(x) - \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x) = \delta_f.$$

Then,

$$\forall \xi_0 \in \boldsymbol{x_0},\ x \in \boldsymbol{I}, \exists \delta_f \in \boldsymbol{\Delta_f},\ g(t,\xi_0) = \int_{\xi_0}^{t} \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x)\mathrm{d}x + \int_{\xi_0}^{t} \delta_f \mathrm{d}x.$$

Let $I(t) = \sum_{i=1}^{n+1} \frac{b-a}{2} \cdot \frac{\alpha_{i-1} - \alpha_{i+1}}{2} T_i^{[a,b]}(t)$, with $\alpha_{n+1} = \alpha_{n+2} = 0$. Now, using equation (4.40), we have:

```
1  Algorithm: CM([a, b], h, n)
   Input: [a, b] an interval, a ≠ b,
   h the expression of a function,
   n ∈ ℕ*
   Output: a Chebyshev Model M corresponding to h
2  switch h do
3      case h = x ↦ c: M ← CMConst(c, [a, b], n);
4      case h = x ↦ x: M ← CMVar([a, b], n);
5      case h = f + g:
6          M_f ← CM(n, [a, b], f, n);
7          M_g ← CM(n, [a, b], g, n);
8          M ← CMAdd(M_f, M_g, n);
9      endsw
10     case h = f · g:
11         M_f ← CM(n, [a, b], f, n);
12         M_g ← CM(n, [a, b], g, n);
13         M ← CMMul(M_f, M_g, [a, b], n);
14     endsw
15     case h = f/g: M ← CMDiv(n, [a, b], f, g, n);
16     case h = g ∘ f:
17         M_f ← CM(n, [a, b], f, n);
18         M ← CMComp([a, b], M_f, g, n);
19     endsw
20 endsw
21 return M ;
```

Algorithm 4.5.12: Computation of Chebyshev Models

Algorithm: `CMIntegrate`$(M_f, \boldsymbol{I}, \boldsymbol{x_0}, n)$

Input: M_f a Chebyshev Model corresponding to a function f over the interval $\boldsymbol{I} = [a, b]$, $n \in \mathbb{N}^\star$ the degree of polynomial involved in M_f

Output: a Chebyshev Model M corresponding to $g(t, \xi_0) = \int_{\xi_0}^{t} f(x)\mathrm{d}x$, for all $\xi_0 \in \boldsymbol{x_0}$

$(\boldsymbol{a_0}, \ldots, \boldsymbol{a_n}, \boldsymbol{\Delta_f}) \leftarrow M_f$;
$\boldsymbol{a_{n+1}} \leftarrow [0; 0]$;
$\boldsymbol{a_{n+2}} \leftarrow [0; 0]$;
for $k \leftarrow 0$ **to** $n+1$ **do**
  $\boldsymbol{c_k} \leftarrow [0; 0]$;
end
for $i \leftarrow 1$ **to** $n+1$ **do**
  $\boldsymbol{c_i} \leftarrow \dfrac{b-a}{2} \cdot \dfrac{\boldsymbol{a_{i-1}} - \boldsymbol{a_{i+1}}}{2i}$;
end
$\boldsymbol{c_0} \leftarrow [-1, -1] \cdot$ `EvaluateClenshawIA`$(\boldsymbol{c_0}, \ldots, \boldsymbol{c_{n+1}}, \boldsymbol{x_0}, [a, b])$;
$\boldsymbol{B} \leftarrow \boldsymbol{\Delta} \cdot ([a, b] - \boldsymbol{x_0})$;
$B \leftarrow \max(|\sup(\boldsymbol{B})|, |\inf(\boldsymbol{B})|$;
$\boldsymbol{\Delta} \leftarrow [-B, B] + \boldsymbol{c_{n+1}} \cdot [\boldsymbol{-1}, \boldsymbol{1}]$;
$M \leftarrow (\boldsymbol{c_0}, \ldots, \boldsymbol{c_n}, \boldsymbol{\Delta})$;
return M ;

Algorithm 4.5.13: Integration of Chebyshev Models

$$\int_{\xi_0}^{t} \sum_{i=0}^{n} \alpha_i T_i^{[a,b]}(x)\mathrm{d}x = I(t) - I(\xi_0) = \sum_{k=0}^{n+1} \gamma_k \, T_k^{[a,b]}(t),$$

and

$$\left| \int_{\xi_0}^{t} \delta_f \mathrm{d}x \right| \leqslant |\delta_f| \, |x - \xi_0| \leqslant B.$$

Now based on the correction of `EvaluateClenshawIA`, we have $-I(\boldsymbol{x_0}) \subseteq \boldsymbol{c_0}$ and hence $\gamma_k \in \boldsymbol{c_k}$ for all $k = 0 \ldots n+1$.

This means that we have $\gamma_i \in \boldsymbol{c_i}$ $(i = 0, \ldots, n+1)$ such that

$$\forall \xi_0 \in \boldsymbol{x_0},\, t \in \boldsymbol{I}, \exists \beta \in [-B, B],\, g(t, \xi_0) = \sum_{k=0}^{n} \gamma_k \, T_k^{[a,b]}(t) + \underbrace{\gamma_{n+1} \, T_{n+1}^{[a,b]}(t) + \beta}_{\boldsymbol{\Delta}}.$$

□

4.6 Experimental results and discussion

We implemented a prototype of both TMs and CMs methods in Maple, using the IntpakX* package. They are available at http://www.ens-lyon.fr/LIP/Arenaire/Ware/

*. http://www.math.uni-wuppertal.de/~xsc/software/intpakX/

ChebModels/. A tuned C implementation is available for TMs in Sollya and a C implementation for CMs will soon be included also therein. The timings are given based on these C implementations. They were obtained on a system having a hardware configuration featuring an Intel Core2 Duo processor running at 2.4GHz, and 4GB of memory, and running Ubuntu 10.04 LTS.

Table 4.2 shows the quality of some absolute error bounds obtained with CMs vs. TMs. Each row of the table represents one example. The function f, the interval I and the degree n of the approximation polynomial are given in the first column. In the second column we give the validated upper-bound obtained using a CM. The exact error between f and the polynomial from CM is provided in the third column such that one can observe the overestimation of the validated bound. We also give the remainder bounds and the exact error obtained when an interpolating polynomial is directly used (directly means that the remainder is computed using (1.9) and automatic differentiation as explained in Section 1.4.3). Finally we present the remainder obtained using a TM and the exact error. The Taylor polynomial was developed in the midpoint of I and the necessary polynomials bounds were computed using a Horner scheme as explained in Chapter 2.

The examples presented are representative for several situations, and we will detail them in what follows. The first five were analyzed in Section 1.4.3 for comparing the bounds obtained with "interpolation + AD" vs. TMs. There, we observed that in some cases the overestimation in the interpolation remainder is so big, that we can not benefit from using such a polynomial. We used them in order to highlight that CMs do not have this drawback and the remainders obtained with our methods have better quality than the TMs in all situations.

The first example presents 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. It has singularities in the complex plane (in $\pi/2+\mathbb{Z}\pi$), but the interval I is relatively far from these singularities. All the methods present a relatively small overestimation. The third example is the same function but over a larger interval. In these case, the singularities are closer and Taylor polynomials are not very good approximations. The fourth and fifth examples are composite functions on larger intervals. The overestimation in the interpolation method becomes very large, rendering this method useless, while it stays reasonable with TMs and CMs.

The following examples $(6-8)$ are similar to some presented in [10]. There, the authors computed the minimax polynomials for these functions. Evidently, the polynomials obtained with CMs have a higher approximation error than the minimax, however, it is important to notice that in these tricky cases the remainder bound obtained for the CMs stays fairly good and it is much better than the one obtained from TMs.

Examples $8-9$ present the case when the definition domain of the function is close to a singularity. As seen in these examples, when a direct interpolation process is used for a composite function, unfortunately, one can not apply Lemma 4.3.1 for bounding the remainder. Consequently, the bound obtained for the remainder is highly overestimated. However, when using the approach based on "basic bricks" both TMs and CMs benefit from it, yielding a much better remainder bound.

Example 10 deals with a function which is associated to the classical Runge phenomenon. Firstly, since the complex singularities of the function f defined by $f(x) = 1/(1+4x^2)$ are close to the definition interval I, the Taylor polynomial is not a good approximation. Then, the interpolation method gives unfeasible interval remainder bounds due to the overestimation of the $n+1$th derivative of the function f. On the contrary, CM bound is feasible in this case.

Example 11 is presented to emphasize the fact that similarly to TMs, the CMs reduce the dependency problem and can convey informations about the properties of functions. The CM for $f = \sin^2 + \cos^2$ over $[-1,1]$, without performing any expression simplification, is in fact $(0.99999999997 + \tilde{P}(x), [-3.91 \cdot 10^{-9}, 3.91 \cdot 10^{-9}])$, where $\tilde{P}(x)$ is a polynomial with a tiny supremum norm. Hence, the information that f is in fact 1, is conveyed when using such models, while

when using interval arithmetic directly, one would obtain an interval range of $[0.29, 1.71]$.

For the same examples, timings are given in Table 4.3. We observe that in practice, for the current C implementation, TMs are 2 times faster than CMs for the same fixed degree n for the polynomial. We have already seen that the advantage of Taylor polynomials consists of considering splitting the input interval. So, we give in Table 4.4 the comparison of the error bounds between the same CM of degree n over the interval $[a, b]$ and 2 TMs with the same degree n over $[a, (a+b)/2]$ and $[(a+b)/2, b]$ (the maximum of the two bounds obtained for the 2 TMs is given in the table). Consequently, we observe that for *basic analytic functions TMs* are currently more efficient. Taylor approximations are better in the sense that they can be obtained faster, given the same target error bound. For these functions, if several polynomial approximations are allowed, i.e. we are not interested in obtaining a compact representation of the function over the whole interval, it seems to be more efficient to split the interval an compute several Taylor approximations. On the other hand, for more complicated functions, CMs seem to be more performant.

No	$f(x), I, n$	CM	Exact bound	Interpolation	Exact bound	TM	Exact bound
1	$\sin(x), [3,\ 4], 10$	$\mathbf{1.19 \cdot 10^{-14}}$	$1.13 \cdot 10^{-14}$	$\mathbf{1.19 \cdot 10^{-14}}$	$1.13 \cdot 10^{-14}$	$1.22 \cdot 10^{-11}$	$1.16 \cdot 10^{-11}$
2	$\arctan(x), [-0.25,\ 0.25], 15$	$\mathbf{7.89 \cdot 10^{-15}}$	$7.95 \cdot 10^{-17}$	$\mathbf{7.89 \cdot 10^{-15}}$	$7.95 \cdot 10^{-17}$	$2.58 \cdot 10^{-10}$	$3.24 \cdot 10^{-12}$
3	$\arctan(x), [-0.9,\ 0.9], 15$	$\mathbf{5.10 \cdot 10^{-3}}$	$1.76 \cdot 10^{-8}$	$\mathbf{5.10 \cdot 10^{-3}}$	$1.76 \cdot 10^{-8}$	$1.67 \cdot 10^{2}$	$5.70 \cdot 10^{-3}$
4	$\exp(1/\cos(x)), [0,\ 1], 14$	$\mathbf{5.22 \cdot 10^{-7}}$	$4.95 \cdot 10^{-7}$	0.11	$6.10 \cdot 10^{-7}$	$9.06 \cdot 10^{-3}$	$2.59 \cdot 10^{-3}$
5	$\frac{\exp(x)}{\log(2+x)\ \cos(x)}, [0,\ 1], 15$	$\mathbf{4.86 \cdot 10^{-9}}$	$2.21 \cdot 10^{-9}$	0.18	$2.68 \cdot 10^{-9}$	$1.18 \cdot 10^{-3}$	$3.38 \cdot 10^{-5}$
6	$\sin(\exp(x)), [-1,\ 1], 10$	$\mathbf{2.56 \cdot 10^{-5}}$	$3.72 \cdot 10^{-6}$	$4.42 \cdot 10^{-3}$	$3.72 \cdot 10^{-6}$	$2.96 \cdot 10^{-2}$	$1.55 \cdot 10^{-3}$
7	$\tanh(x+0.5) - \tanh(x-0.5), [-1, 1], 10$	$\mathbf{1.75 \cdot 10^{-3}}$	$4.88 \cdot 10^{-7}$	$8.48 \cdot 10^{-3}$	$4.88 \cdot 10^{-7}$	8.68	$2.96 \cdot 10^{-3}$
8	$\sqrt{x+1.0001}, [-1, 0], 10$	$\mathbf{3.64 \cdot 10^{-2}}$	$3.64 \cdot 10^{-2}$	$3.64 \cdot 10^{-2}$	$3.64 \cdot 10^{-2}$	0.11	0.11
9	$\sqrt{x+1.0001} \cdot \sin(x), [-1, 0], 10$	$\mathbf{3.32 \cdot 10^{-2}}$	$3.08 \cdot 10^{-2}$	$3.21 \cdot 10^{33}$	$3.08 \cdot 10^{-2}$	0.12	$9.83 \cdot 10^{-2}$
10	$\frac{1}{1+4x^2}, [-1, 1], 10$	$\mathbf{1.13 \cdot 10^{-2}}$	$6.17 \cdot 10^{-3}$	$1.50 \cdot 10^{7}$	$4.95 \cdot 10^{-3}$	$+\infty$	$8.20 \cdot 10^{2}$
11	$\sin^2(x) + \cos^2(x), [-1, 1], 10$	$\mathbf{3.91 \cdot 10^{-9}}$	$2.1 \cdot 10^{-11}$	$8.44 \cdot 10^{-8}$	$2.29 \cdot 10^{-50}$	$8.74 \cdot 10^{-6}$	$5.57 \cdot 10^{-52}$

Table 4.2: Examples of bounds obtained by several methods

No	$f(x), I, n$	Timing CM (ms)	Timing TM (ms)
1	$\sin(x), [3,\ 4], 10$	4	2
2	$\arctan(x), [-0.25,\ 0.25], 15$	10	4
3	$\arctan(x), [-0.9,\ 0.9], 15$	14	7
4	$\exp(1/\cos(x)), [0,\ 1], 14$	31	14
5	$\frac{\exp(x)}{\log(2+x)\ \cos(x)}, [0,\ 1], 15$	38	19
6	$\sin(\exp(x)), [-1,\ 1], 10$	7	4
7	$\tanh(x+0.5) - \tanh(x-0.5), [-1, 1], 10$	5	3
8	$\sqrt{x+1.0001}, [-1, 0], 10$	10	4
9	$\sqrt{x+1.0001} \cdot \sin(x), [-1, 0], 10$	20	8
10	$\frac{1}{1+4x^2}, [-1, 1], 10$	10	4
11	$\sin^2(x) + \cos^2(x), [-1, 1], 10$	18	10

Table 4.3: Timings in miliseconds for results given in Table 4.2

Examples regarding rigorous quadrature

We have seen in the introduction that we are interested in rigorously computing some definite integrals. We consider in the following examples comparisons between TMs and CMs regarding the number of correct digits obtained in these integrals. The results obtained with TMs are based on the classical algorithm of integrating TMs taken from [13]. In the case of the CMs, the algorithm

No	$f(x)$, I, n	CM bound	TM bound
1	$\sin(x)$, $[3,\ 4]$, 10	$1.19 \cdot 10^{-14}$	$\mathbf{5.95 \cdot 10^{-15}}$
2	$\arctan(x)$, $[-0.25,\ 0.25]$, 15	$7.89 \cdot 10^{-15}$	$\mathbf{1.06 \cdot 10^{-15}}$
3	$\arctan(x)$, $[-0.9,\ 0.9]$, 15	$5.10 \cdot 10^{-3}$	$\mathbf{5.81 \cdot 10^{-4}}$
4	$\exp(1/\cos(x))$, $[0,\ 1]$, 14	$\mathbf{5.22 \cdot 10^{-7}}$	$1.10 \cdot 10^{-5}$
5	$\frac{\exp(x)}{\log(2+x)\ \cos(x)}$, $[0,\ 1]$, 15	$\mathbf{4.86 \cdot 10^{-9}}$	$4.60 \cdot 10^{-8}$
6	$\sin(\exp(x))$,$[-1,\ 1]$, 10	$\mathbf{2.56 \cdot 10^{-5}}$	$1.01 \cdot 10^{-4}$
7	$\tanh(x+0.5) - \tanh(x-0.5)$, $[-1, 1]$, 10	$1.75 \cdot 10^{-3}$	$\mathbf{7.28 \cdot 10^{-4}}$
8	$\sqrt{x+1.0001}$, $[-1, 0]$, 10	$\mathbf{3.64 \cdot 10^{-2}}$	0.11
9	$\sqrt{x+1.0001} \cdot \sin(x)$, $[-1, 0]$, 10	$\mathbf{3.32 \cdot 10^{-2}}$	$7.06 \cdot 10^{-2}$
10	$\frac{1}{1+4x^2}$, $[-1, 1]$, 10	$\mathbf{1.13 \cdot 10^{-2}}$	$1.39 \cdot 10^{2}$
11	$\sin^2(x) + \cos^2(x)$, $[-1, 1]$, 10	$\mathbf{3.91 \cdot 10^{-9}}$	$2.23 \cdot 10^{-8}$

Table 4.4: Examples of bounds *1 CM vs. 2 TMs*.

Order	Subdiv.	Bound TM	Bound CM
5	1	[3.0231893333333, 8.5807786666666]	[3.0986941190195, 3.1859962140742]
	4	[3.1415363229415, 3.1416629536292]	[3.1415907717769, 3.1415943610772]
	16	[3.1415926101614, 3.1415926980786]	[3.1415926531269, 3.1415926539131]
10	1	[-2.1984010266006, 3.2113963175267]	[3.1411981994969, 3.1419909934525]
	4	[3.1415926519535, 3.1415926546870]	[3.1415926535805, 3.1415926535990]
	16	[3.1415926535897, 3.1415926535897]	[3.1415926535897932, 3.1415926535897932]

Table 4.5: Computation of digits of π using TMs vs. CMs

is very similar: we integrate the polynomial part and bound straightforwardly the remainder as shown in Algorithm 4.5.13. Then we bound the polynomial part using Algorithm 4.5.1 for CMs and Algorithm 2.1.1 for TMs. We can choose to subdivide the interval over which we consider the definite integral, in order to have better approximations for the same fixed order of the model.

Example 4.6.1 (Toy example: digits of π.)**.** *For a first toy example, let us consider the comparison between TMs and CMs regarding the computation of the digits of π using*

$$\pi = \int_0^1 \frac{4}{1+x^2} \mathrm{d}x.$$

The results obtained using a TM are taken from [13]. We show in Table 4.5 the number of correct digits obtained with both methods. In the first column we give the degree of the polynomial used; in the second column, the number of subintervals we split the initial interval.

Let us now give a rigorous answer to the integral in Example 1.0.1. In 0.2 seconds in our Sollya implementation, we have all the exact digits given in the introduction plus a rigorous enclosure of the result: $0.43060610312069060491237735 52[4, 6]$. We used a CM of degree 30 over the whole interval $[0, 1]$.

For the second example 1.0.2, we face the case of a very oscillatory function. In this case, we had to make a tuned splitting in subintervals in order to obtain the enclosure of the solution. We note that our purpose is to show the potential of these tools in developing rigorous methods. Of

course, several methods exist in literature [109, 62] that were not analyzed here, for computing numerical integrals, with guarantees on the quality of output.

Certified plots

Finally, let us analyze the inequality given in Example 1.0.4, that is necessary in the formal proof of Kepler's conjecture. We want to prove that $f(x) = 2\pi - 2x\,\mathrm{asin}((\cos 0.797)\sin(\pi/x)) + 0.0331x - 2.097 > 0$ for $x \in \boldsymbol{I} = [3, 64]$. Firstly, we provide the *certified plot* of $f(x)$. We use the following method: first, Algorithm 4.5.12 is used to compute an RPA for f of the form (p, B), then we consider a partition of the function domain $[3, 64] = \bigcup_{\substack{i=[\![0,n-1]\!],\\ x_0=3,\, x_n=64}} [x_i, x_{i+1}]$ and evaluate using interval arithmetic $\tilde{p}([x_i, x_{i+1}]) + B \supseteq f([x_i, x_{i+1}])$, for each i. These intervals provide a piecewise enclosure of the image of f. This is shown in Figure 4.4, where we used a CM of degree 81 for the whole interval, which provides us with an error bound of $2 \cdot 10^{-11}$. This gives us a first level of confidence in the result.

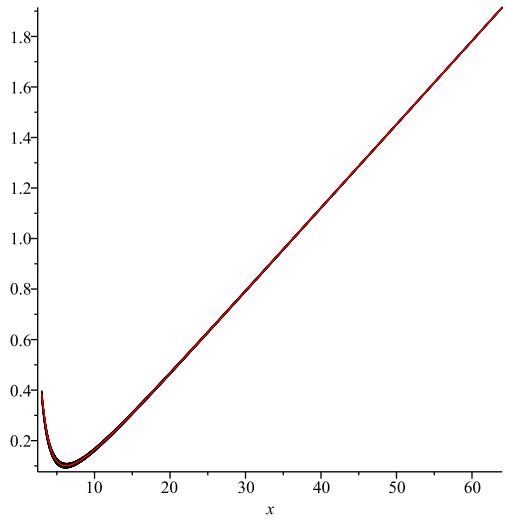


Figure 4.4: Certified plot of $f(x) = 2\pi - 2x\,\mathrm{asin}((\cos 0.797)\sin(\pi/x)) + 0.0331x - 2.097$ for $x \in \boldsymbol{I} = [3, 64]$.

In order to formally prove this kind of inequalities, we use a method similar to the one presented in Chapter 3. Specifically, once we have computed a formally proven RPA (TM or CM) such that $|f(x) - p(x)| \leqslant B$, for all $x \in \boldsymbol{I}$, the positivity of $p(x) - B$ implies the positivity of f, since $f(x) \geqslant p(x) - B$ for all $x \in \boldsymbol{I}$. We have shown in Chapter 3 that showing the positivity of a polynomial relies on the sum-of-squares method that is already formalized.

In this example, using a partitioning in intervals of length 1 and computing one TM for each interval and obtaining the SOS instance for each polynomial using the same tools as in Chapter 3, took us 3.5 seconds.

For instance, on $I_0 = [3, 4]$ we obtain:

$$
\begin{aligned}
f(x) \geqslant (4-x)*(&95709837/1048576000*(x^2-11/2*x+6)^2+95709837/1048576000*(-2*x+5)^2 \\
&+40842491377/536870912000*(x^2-8*x+16)^2+2180258663/16777216000*(-x^2+7*x-12)^2 \\
&+35497951/2097152000*(x^2-6*x+9)^2+95709837/2097152000*(-1/2*x^2+4*x-8)^2 \\
&+95709837/1048576000*(-x^2+7*x-12)^2+95709837/2097152000*(-1/2*x^2+7/2*x-6)^2 \\
&+95709837/1048576000*(x^2-6*x+9)^2)+(x-3)*(95709837/1048576000*(3/2*x-5)^2 \\
&+95709837/1048576000*(3/2*x^2-11*x+20)^2+6004769207/134217728000*(x^2-8*x+16)^2 \\
&+28278727/838860800*(-x^2+7*x-12)^2+95709837/1048576000*(x^2-6*x+9)^2 \\
&+95709837/2097152000*(x^2-8*x+16)^2+95709837/1048576000*(1/2*x^2-4*x+8)^2 \\
&+95709837/2097152000*(-x^2+7*x-12)^2+95709837/1048576000*(1/2*x^2-7/2*x+6)^2).
\end{aligned}
$$

The only part that is still under development is a formally proven TM.

4.7 Conclusion and future work

We presented a new tool which is potentially useful in various rigorous computing applications. Currently, CMs for univariate composite functions are more efficient than TMs. This observation seems promising for replacing Taylor Models for multivariate functions with models based on better approximations. As we already mentioned, another future work consists in improving both the complexity of computation of coefficients of CI for basic functions, and the complexity of multiplication of CMs using an adaptation of Fast Fourier Transform to interval arithmetic. Longer term goals are the complete formalization of TMs and CMs in a formal proof checker and the extension of CMs to other families of orthogonal polynomials, like Legendre polynomials for example.

Another potential application of CMs is the rigorous solving of ODE using validated enclosure methods. In the next chapter, we begin the study of the complexity of such methods for the case of D-finite functions.

CHAPTER 5

Rigorous Uniform Approximation of D-finite Functions

două culori ce nu s-au văzut niciodată,
una foarte de jos, intoarsă spre pământ,
una foarte de sus, aproape ruptă
în infrigurata, neasemuita luptă

Nichita Stănescu, Ce bine că eşti.

In the previous chapter we studied rigorous polynomial approximations based on truncated Chebyshev series expansions or Chebyshev interpolants for functions given explicitly by an expression. In this chapter we obtain this kind of RPAs for functions that are given as solutions of ordinary differential equations with polynomial coefficients (D-finite functions). This case was presented in Example 1.1.6.

The order-n truncation of the Chebyshev expansion of a function over a given interval is a near-best uniform polynomial approximation of the function on that interval and a wide range of numerical methods exist for obtaining these approximations. Moreover, in the case of D-finite functions, the coefficients of the expansions obey linear recurrence relations with polynomial coefficients. However, these do not lend themselves to a direct recursive computation of the coefficients, owing chiefly to the lack of initial conditions.

In this work, we show how these recurrences can nonetheless be used, as part of a validated process, to compute good uniform approximations of D-finite functions, together with rigorous error bounds. Our approach is based on a classical numerical method going back to Clenshaw, revisited in the light of properties of the recurrence relations we consider, and combined with an enclosure method for ODE. One more important contribution of this work is that the algorithms for obtaining these RPAs have a linear arithmetic complexity.

This is part of an undergoing joint work [7] with Alexandre Benoit and Marc Mezzarobba.

5.1 Introduction

In the previous chapter we have seen how to obtain *near-minimax rigorous polynomial approximations* for elementary functions, that is, functions that were given by an expression containing only basic functions like constants, identity function, $\exp, \sin, \cos$, etc. and operations like addition, multiplication, division, composition. In this work, we extend the possibility of having Chebyshev Models (Definition 4.5.2) for D-finite functions, that is, solutions of linear ordinary differential equations (LODE) with polynomial coefficients [153]. This property allows to develop

a uniform theoretic and algorithmic treatment of these functions, an idea that has led to many applications in recent years in the context of Symbolic Computation [173, 149].

Many of the special functions commonly used in areas such as mathematical physics are *D-finite*, so, besides their exact representation by differential equations, their repeated evaluation is often needed. We evaluate a function y at many points lying on an interval, usually with moderate precision for plotting, numerical integration, or the computation of minimax approximation polynomials. A standard approach to address this need resorts to polynomial approximations of y. We deem it useful to support working with arbitrary D-finite functions in a computer algebra system. Hence, it makes sense to ask for good uniform polynomial approximations of these functions on intervals. Rigorous error bounds are necessary in order for the whole computation to yield a rigorous result.

We recall that besides easy numerical evaluation, polynomial approximations provide a convenient representation of continuous functions on which comprehensive arithmetics including addition, multiplication, composition and integration may be defined. For a variety of reasons, it is natural to write the polynomials on the Chebyshev basis rather than the monomial basis. In particular, the truncations that occur during most arithmetic operations then maintain good approximation on the whole interval. In this sense, previous works already mentioned in Chapter 4 include Trefethen *et al.*'s Chebfun [159, 54] or Epstein, Miranker and Rivlin's "ultra-arithmetic" [60, 61, 84].

We already explained in previous chapters the need of an algebra with RPAs. This work allows us to use arbitrary D-finite functions as "basic functions" at the leaves and nodes of expression trees to be evaluated using Chebyshev Models.

Finally, perhaps the main appeal of RPAs and related techniques is the ability to solve functional equations rigorously using enclosure methods [114, 84, 99, 124]. LODE with polynomial coefficients are among the simplest equations to which these tools apply. A third goal of this work is to begin the study of the complexity of validated enclosure methods, from a computer algebra point of view, using this family of problems as a prototype.

5.1.1 Setting

We fix a linear homogeneous differential equation of order r with polynomial coefficients

$$L \cdot y = a_r y^{(r)} + a_{r-1} y^{(r-1)} + \cdots + a_0 y = 0, \quad a_i \in \mathbb{Q}[x]. \tag{5.1}$$

Up to a change of variable, we assume that we seek a polynomial approximation of a solution y of (5.1) over the interval $[-1, 1]$. We also assume that $a_r(x) \neq 0$ for $x \in [-1, 1]$, so that all solutions of (5.1) are analytic on $[-1, 1]$. Besides the operator L, we are given r boundary conditions

$$\lambda_i(y) = \ell_i, \quad 1 \leqslant i \leqslant r, \tag{5.2}$$

each of the form $\lambda_i(y) = \sum_{j=1}^{q} \mu_j y^{(r_j)}(x_j)$ with $x_j \in [-1; 1]$ and $r_j \leqslant r - 1$. The boundary conditions are chosen such that the function y of interest is the unique solution of (5.1) satisfying (5.2). They are independent in the sense that the $\lambda_i : \ker L \to \mathbb{C}$ are linearly independent. Note that the case of initial values given outside the domain of expansion may be reduced to our setting using numerical analytic continuation [107].

Complexity model Unless otherwise noted, we assume for simplicity that all computations are carried out in exact (rational) arithmetic. The rigor of the computation is unaffected if exact arithmetic is replaced by floating-point arithmetic in Algorithm 5.3.1 and by interval arithmetic in Algorithm 5.5.2. (In the case of Algorithm 5.4.1, switching to interval arithmetic requires small adjustments.) However, we do not analyze the effect of rounding errors on the quality of the approximation polynomial p and error bound B when the computations are done in floating-point

arithmetic. In simple cases at least, we expect that Algorithm 5.3.1 exhibits comparable stability to similar methods based on backward recurrence [168]. Our experiments (reported in Section 5.7) show a satisfactory numerical behavior.

To account for this variability in the underlying arithmetic, we assess the complexity of the algorithms in the *arithmetic model*. In other words, we only count basic operations in $\mathbb{Q}$, while neglecting both the size of their operands and the cost of accessory control operations.

Notations We use double brackets to denote integer intervals $[\![i, j]\!]$. Related to Chebyshev expansions, in this chapter we use for simplifying some proofs a slightly different notation detailed in §5.2.1 below. Notations from Theorem 5.2.1 are also repeatedly used in the subsequent discussion.

5.1.2 Outline

In this work, Chebyshev Models of D-finite functions are obtained in two stages. We first compute a candidate approximation polynomial, based on the Chebyshev expansion of the function y. No attempt is made to control the errors rigorously at this point. We then validate the output using enclosure methods.

This chapter is organized as follows. In Section 5.2, we review and study the recurrence relation satisfied by the coefficients of Chebyshev series of D-finite functions. The use of this recurrence is the key to the linear time complexity. The algorithm we use to actually compute the coefficients, described in Section 5.3, is essentially Fox and Parker's variant [65, Chap. 5] of Clenshaw's algorithm [42]. In general, its output is an approximation of unspecified quality. Under some simplifying assumptions (*i.e.* Hypothesis H, page 156), we prove that the computed approximated truncated Chebyshev series converges at least exponentially to the true TCS (see Proposition 5.3.2). In Section 5.4, we study Chebyshev series expansions of rational functions. Most importantly, we state remainder bounds that are then used in Section 5.5, along with an enclosure method for differential equations, to validate the output of the first stage and obtain the bound B. Section 5.7 presents a prototype implementation of our approach and experimental results.

5.2 Chebyshev Expansions of D-finite Functions

5.2.1 Chebyshev Series

A brief overview of Chebyshev series was given in Section 4.2. Let f be a solution of Equation (5.1). As such, it may be analytically continued to any domain $U \subseteq \mathbb{C}$ that contains no singular point of the equation. Since there are no singularities on the segment $[-1, 1]$, there exists an elliptic disk $\bar{\varepsilon}_r \subseteq U$, with foci in ± 1 and $r \geq 1$ (cf. Definition 4.2.19), for which Theorem 4.2.22 applies. Instead of the more common

$$\sum_n{}' a_n T_n = \frac{a_0}{2} T_0 + a_1 T_1 + a_2 T_2 + \cdots , \tag{5.3}$$

we write the Chebyshev series of f as

$$f(x) = \sum_{n=-\infty}^{\infty} c_n T_n(x), \qquad c_{-n} = c_n, \tag{5.4}$$

where the *Chebyshev coefficients* $c_n = \frac{1}{2} a_n$ are given by

$$c_n = \frac{1}{\pi} \int_{-1}^{1} \frac{f(x) T_n(x)}{\sqrt{1 - x^2}} \mathrm{d}x, \qquad n \in \mathbb{Z}. \tag{5.5}$$

This choice makes the link between Chebyshev and Laurent expansions (Remark 4.2.21) as well as the action of recurrence operators (both discussed below) more transparent.

From Theorem 4.2.22, the coefficients c_n then satisfy $c_n = O(\alpha^n)$ for all $\alpha > r^{-1}$; and the Chebyshev expansion (5.4) converges uniformly to f on $\bar{\varepsilon}_r$. From Remark 4.2.21, c_n are also the coefficients of the (doubly infinite) Laurent expansion of the function $\tilde{f}(z) = f(\frac{z+z^{-1}}{2})$ around the unit circle.

Let $C \subseteq \mathbb{C}^{\mathbb{Z}}$ be the vector space of doubly infinite sequences $(c_n)_{n\in\mathbb{Z}}$ such that

$$(\forall n \in \mathbb{N})(c_n = c_{-n}) \qquad \text{and} \qquad (\exists \alpha < 1)(c_n = O(\alpha^n)).$$

Thus the sequence of Chebyshev coefficients of a function f that is analytic on some complex neighborhood of $[-1; 1]$ belongs to C. Conversely, for all $c \in C$, the function series $\sum_{n=-\infty}^{\infty} c_n T_n(x)$ converges uniformly on (some neighborhood of) $[-1; 1]$ to an analytic function $f(x)$.

We denote by $\pi_d : f \mapsto \sum_{n=-d}^{d} c_n T_n$ the associated orthogonal projection on the subspace of polynomials of degree at most d. It is well-known (Equation (4.18)) that $\pi_d(y)$ is a *near-minimax* uniform approximation. Our choice of truncated Chebyshev expansions over other near-minimax approximations with nice analytical properties is motivated primarily by the existence of a recurrence relation on the coefficients (c_n) when f is a D-finite function.

5.2.2 The Chebyshev Recurrence Relation

The polynomials T_n satisfy the recurrence

$$2xT_n(x) = T_{n-1}(x) + T_{n+1}(x), \tag{5.6}$$

as well as the mixed differential-difference relation

$$2(1-x^2)T_n'(x) = n(T_{n-1}(x) - T_{n+1}(x)) \tag{5.7}$$

which translates into the integration formula $2nc_n = c'_{n-1} - c'_{n+1}$ where $\sum c'_n T_n = (\sum c_n T_n)'$. From these equalities follows the key ingredient of the approach developed in this work, namely that the sequence of Chebyshev coefficients of a D-finite function obeys a linear recurrence with polynomial coefficients. This fact was observed by Fox and Parker [64, 65] in special cases and later proved in general by Paszkowski [130]. Properties of this recurrence and generalizations to other orthogonal polynomial bases were explored in a series of papers by Lewanowicz starting 1976 (see in particular [95, 96]). The automatic determination of this recurrence in a symbolic computation system was first studied by Geddes [68].

The following theorem summarizes results regarding this recurrence extracted from existing work [130, 95, 96, 138, 8] and slightly extended to fit our purposes. Here and in the sequel, we denote by $\mathbb{Q}(n)\langle S, S^{-1}\rangle$ the skew Laurent polynomial ring over $\mathbb{Q}(n)$ in the indeterminate S, subject to the commutation rules

$$S\lambda = \lambda S \quad (\lambda \in \mathbb{Q}), \qquad Sn = (n+1)S. \tag{5.8}$$

Likewise, $\mathbb{Q}[n]\langle S, S^{-1}\rangle \subseteq \mathbb{Q}(n)\langle S, S^{-1}\rangle$ is the subring of noncommutative Laurent polynomials in S themselves with polynomial coefficients. The elements of $\mathbb{Q}[n]\langle S, S^{-1}\rangle$ identify naturally with linear recurrence operators through the left action of $\mathbb{Q}[n]\langle S, S^{-1}\rangle$ on $\mathbb{C}^{\mathbb{Z}}$ defined by $(n \cdot u)_n = nu_n$ and $(S \cdot u)_n = u_{n+1}$. Recall that L denotes the differential operator appearing in Equation (5.1).

Theorem 5.2.1. *[130, 95, 96, 138, 8] Let u, v be analytic functions on some complex neighborhood of the segment $[-1, 1]$, with Chebyshev expansions*

$$u(x) = \sum_{n=-\infty}^{\infty} u_n T_n(x), \quad v(x) = \sum_{n=-\infty}^{\infty} v_n T_n(x).$$

There exist difference operators $P, Q \in \mathbb{Q}[n]\langle S, S^{-1}\rangle$ with the following properties.

1. *The differential equation $L \cdot u(x) = v(x)$ holds if and only if*
$$P \cdot (u_n) = Q \cdot (v_n). \tag{5.9}$$
2. *The left-hand side operator P is of the form $P = \sum_{k=-s}^{s} b_k(n)S^k$ where $s = r + \max_i(\deg a_i)$ and $b_{-k}(-n) = -b_k(n)$ for all k.*
3. *Letting*
$$\delta_r(n) = 2^r \prod_{i=-r+1}^{r-1} (n-i), \qquad I = \frac{1}{2n}(S^{-1} - S), \tag{5.10}$$
we have $Q = Q_r = \delta_r(n)I^r$ (formally, in $\mathbb{Q}(n)\langle S, S^{-1}\rangle$). In particular, Q depends only on r and satisfies the same symmetry property as P.

We note that I as defined in Eq. (5.10) makes sense as an operator from the *symmetric* sequences $(u_{|n|})_{n\in\mathbb{Z}}$ to $\{(u_n)_{n\in\mathbb{Z}\setminus\{0\}}\}$ and corresponds to the integration of Chebyshev series. A sloppy but perhaps more intuitive statement of the main point of Theorem 5.2.1 would be: "$(\int)^r L \cdot u = w$ if and only if $\delta_r(n)P \cdot u = w$, up to some integration constants".

Proof. Assume $L \cdot u = v$. Benoit and Salvy [8, Theorem 1] give a simple proof that (5.9) holds for some $P, Q \in \mathbb{Q}(n)\langle S, S^{-1}\rangle$. That P and Q can actually be taken to have polynomial coefficients and satisfy the properties listed in the last two items follows from the explicit construction discussed in §4.1 of the same article, based on Paszkowski's algorithm [130, 95], and from the observation that $\delta_r(n)I^r \in \mathbb{Q}[n]\langle S, S^{-1}\rangle$. Rebillard's thesis [138, §4.1] contains detailed proofs of this last point as well as the assertions of Item 2. Several of these results actually go back to [130, 95, 96].

There remains to prove the "if" direction. Consider sequences $u, v \in C$ such that $P \cdot u = Q \cdot v$, and let $y \in C$ be defined by $L \cdot u(x) = \sum_{n=-\infty}^{\infty} y_n T_n(x)$. Then, from what we already proved, we have $P \cdot u = Q \cdot y$, and hence, $Q \cdot y = Q \cdot v$ and we conclude that $y = v$ using Lemma 5.2.2 below. □

Lemma 5.2.2. *The restriction to C of the operator Q of Theorem 5.2.1 is injective.*

Proof. With the notations of Theorem 5.2.1, we show by induction on $r \geqslant 1$ that
$$(v \in C) \wedge \big(|n| \geqslant r \implies (Q_r \cdot v)_n = 0\big) \implies v = 0. \tag{5.11}$$
First, we have $(\ker Q_1) \cap C = \{0\}$ since any sequence belonging to C converges to zero as $n \to \pm\infty$. Now assume that (5.11) holds, and let $v \in C$ be such that $(Q_{r+1} \cdot v)_n = 0$ for $|n| \geqslant r+1$. We observe that w, defined by:
$$w := Q_r \cdot v,$$
is an element of C. Since $r \geqslant 1$, we have
$$\begin{aligned} nQ_{r+1} &= \delta_{r+1}(n)(S^{-1} - S)I^r \\ &= ((n+r)(n+r-1)S^{-1}\delta_r(n) - (n-r)(n-r+1)S\delta_r(n))I^r \\ &= ((n+r)(n+r-1)S^{-1} - (n-r)(n-r+1)S)Q_r \end{aligned}$$
hence for $|n| \geqslant r+1$,
$$(n+r)(n+r-1)w_{n-1} = (n-r)(n-r+1)w_{n+1}. \tag{5.12}$$
Unless w_n is ultimately zero, this implies that $w_{n+1}/w_{n-1} \to 1$ as $n \to \infty$, which is incompatible with the fact that $w \in C$. It follows that $w_n = 0$ for $|n|$ large enough, and using (5.12) again that $w_n = 0$ as soon as $|n| \geqslant r$. Applying the hypothesis (5.11) concludes the induction. □

Following Rebillard, we call (5.9) the Chebyshev recurrence relation associated to (5.1). An easy-to-explain way to compute it,which curiously does not seem to have appeared in print in this form, although it is undoubtedly known to specialists, is to first perform the change of variable $x = \frac{1}{2}(z + z^{-1})$ in the differential equation (5.1), and then obtain a recurrence on the Laurent coefficients of $\tilde{u}(z) = u(x)$ by the classical "Frobenius" method. Benoit and Salvy [8] give a unified presentation of several other algorithms by interpreting them as various ways to perform the substitution $x \mapsto \frac{1}{2}(S + S^{-1})$, $\frac{\mathrm{d}}{\mathrm{d}x} \mapsto (S - S^{-1})^{-1}(2n)$ in a suitable non-commutative division algebra, and show that they compute the same operator P, if the recurrence has no singularities in $[-1, 1]$, which is our case.

Remark 5.2.3. *Using Theorem 5.2.12 and the same notations, for any sequence (u_n) we have:*

$$\forall n, \sum_k b_k(n)u_{n+k} = -\sum_k b_{-k}(-n)u_{n+k} = -\sum_k b_k(-n)u_{-n-k},$$

i.e., $P \cdot (u_n)_{n\in\mathbb{Z}} = -P \cdot (u_{-n})_{n\in\mathbb{Z}}$. Specifically, if (u_n) is a solution of $P \cdot (u_n) = 0$ then so it is (u_{-n}). Hence, from any solution (u_n), we get the symmetric solution $(u_n + u_{-n})$.

5.2.3 Solutions of the Chebyshev Recurrence

Several difficulties arise when trying to use the Chebyshev recurrence to compute the Chebyshev coefficients. The first one is related to initial conditions. Here it may be worth contrasting the situation with the more familiar case of the solution of differential equations in power series. Unlike the *Taylor* coefficients of y, the Chebyshev coefficients $c_0, c_1, \ldots$ that could serve as initial conditions for the recurrence are not related to initial or boundary conditions of the differential equation in any simple way. In particular, as can be seen from Theorem 5.2.1 above, the order $2s$ of the recurrence is larger than that of the differential equation (except in degenerate cases), meaning that we need to somehow obtain more initial values for the recurrence than we "naturally" have at hand. Nevertheless, the recurrence (5.9) shows that the Chebyshev coefficients of a D-finite function are rational linear combinations of a finite number of integrals of the form (5.5). Computing these coefficients efficiently to high accuracy is an interesting problem to which we hope to come back in future work.

Next, also in contrast with the case of power series, the leading and trailing coefficients $b_{\pm s}$ of the recurrence (5.9) may vanish for arbitrarily large values of n even though the differential equation (5.1) is nonsingular. The zeroes of b_s are called the *leading singularities* of (5.9), those of b_{-s}, its *trailing singularities*. In the case of Chebyshev recurrences, leading and trailing singularity sets are opposite of each other.

Example 5.2.4. *For all $k \in \mathbb{Z}$, the Chebyshev recurrence relation associated to the differential equation $y''(x) + xy'(x) + ky(x) = 0$, namely*

$$(n+1)(n+k-2)\, u_{n-2} - 2n(-2n^2 - k + 1)\, u_n - (n-1)(n-k+2)\, u_{n+2} = 0,$$

admits the leading singularity $n = k - 2$.

We do however have some control over the singularities.

Proposition 5.2.5. *With the notations of Theorem 5.2.1, the coefficients of the Chebyshev recurrence (as computed by one of the equivalent algorithms mentioned above) satisfy the relations*

$$b_{j-i}(-j) = -b_{j+i}(-j), \qquad |j| \leqslant r - 1, \quad i \in \mathbb{N}, \tag{5.13}$$

with $b_k = 0$ for $|k| > s$.

Proof. We proceed by induction on r. When $j = 0$, the assertion (5.13) reduces to $b_{-i}(0) = -b_i(0)$, which follows from the second item of Theorem 5.2.1. In particular, this proves the result for $r = 1$. Now let $r \geqslant 2$ and assume that the proposition holds when L has order $r-1$. Write $L = \hat{L} + \partial^r p_r(x)$ where $p_r \in \mathbb{Q}[x]$ and $\hat{L}$ is a differential operator of order $r-1$. Letting $\hat{P} = \sum_{k\in\mathbb{Z}} \hat{b}_k(n) S^k$ be the Chebyshev recurrence operator associated to $\hat{L}$, we then have [8]

$$\delta_r(n)^{-1} P = I\delta_{r-1}(n)^{-1}\hat{P} + p_r(\tfrac{1}{2}(S + S^{-1})) \tag{5.14}$$

where the last term denotes the evaluation of p_r at $x = \frac{1}{2}(S + S^{-1})$. Since

$$I\delta_{r-1}(n)^{-1} = (n\delta_r(n))^{-1}((n-r+2)(n-r+1)S^{-1} - (n+r-2)(n+r-1)S)$$

by the commutation rule (5.8), the relation (5.14) rewrites as

$$\begin{aligned} P =& \frac{1}{n}\sum_k \big((n-r+2)(n-r+1)\hat{b}_{k+1}(n-1) - (n+r-2)(n+r-1)\hat{b}_{k-1}(n+1)\big)S^k \\ &+ \delta_r(n) p_r\left(\frac{1}{2}(S+S^{-1})\right). \end{aligned}$$

The case $j = 0$ having already been dealt with, assume $0 < |j| < r$. Since $\delta_r(-j) = 0$ and p_r is a polynomial, it follows by extracting the coefficient of S^k in the last equality and evaluating at $n = -j$ that

$$-jb_k(-j) = (j+r-2)(j+r-1)\hat{b}_{k+1}(-j-1) - (j-r+2)(j-r+1)\hat{b}_{k-1}(-j+1). \tag{5.15}$$

Now $\hat{b}_{j-i}(-j) = -\hat{b}_{j+i}(-j)$ for $|j| < r-1$ by the induction hypothesis, and the term involving $\hat{b}_{k\pm1}$ vanishes for $j = \mp(r-1)$ and $j = \mp(r-2)$. In each case, we obtain $b_{j-i}(-j) = -b_{j+i}(-j)$. □

Corollary 5.2.6. *If $(u_{|n|})_{n\in\mathbb{Z}}$ is a symmetric sequence, then $(P \cdot u)_n = 0$, for all $|n| < r$.*

Proof. We have

$$(P \cdot u)_n = \sum_{k\in\mathbb{Z}} b_k(n) u_{n+k} = \sum_{i\in\mathbb{Z}} b_{i-n}(n) u_i.$$

Using Proposition 5.2.5 with $j = -n$,

$$\sum_{i\in\mathbb{Z}} b_{i-n}(n) u_i = -\sum_{i\in\mathbb{Z}} b_{-i-n}(n) u_i = -\sum_{i\in\mathbb{Z}} b_{i-n}(n) u_i,$$

whence, $(P \cdot u)_n = -(P \cdot u)_n$, for all $|n| < r$. □

Corollary 5.2.7. *The dimension of the space of symmetric solutions of a Chebyshev recurrence is at least $s + r$.*

Proof. Let P be a Chebyshev operator recurrence of order $2s$ in relation to a differential equation of order r.

Let $(u_0, \cdots, u_{s+r-1})$ be a tuple. Let $(y_n)_{n\in\mathbb{Z}}$ be a sequence which satisfies $y_i = u_i$ for all $0 \leq |i| < s + r$ and $P \cdot y_n = 0$ for all $|n| \geq r$. Since P is of order $2s$, there exists always a sequence y_n which satisfies these properties. Proving the corollary is equivalent to prove that this sequence is solution of the recurrence relation P and is symmetric. By the Theorem 5.2.1, we have $P \cdot y_n = -P \cdot y_{-n}$. By equality of the initial condition of the recurrence, we deduce $y_n = y_{-n}$. To prove that y_n is solution of the recurrence relation, we need to prove the equality $P \cdot y_n = 0$ for all $|n| < r$, which follows directly from Corollary 5.2.6.

□

5.2.4 Convergent and Divergent Solutions

We briefly recall a powerful result regarding asymptotic behavior of solutions of linear recurrences. For simplicity, the exact statement of the following theorem was taken from [108], while other formulations and proofs can be found in references therein, the actual results going back to Poincaré (1885), Perron (1910), Kreuser (1914).

Perron-Kreuser theorem

Let a linear recurrence relation

$$p_m(n)u_{n+m} + \ldots + p_1(n)u_{n+1} + p_0(n)u_n = 0, \tag{5.16}$$

and the corresponding operator $\sum_{k=0}^{m} p_k S^k$. Assume that the coefficients $p_k(n) \sim_{n\to\infty} c_k n^{d_k}$ for some $c_k \in \mathbb{C}$, $d_k \in \mathbb{Z}$. If (u_n) is a solution of (5.16) with $u_{n+1}/u_n \sim_{n\to\infty} \alpha n^\kappa$, then for the recurrence equation to hold asymptotically, the maximum value of $d_k + k\kappa$ must be reached at least twice, so that the corresponding terms can cancel. This means that $-\kappa$ must be among the slopes of the edges of the Newton polygon of the equation. The Newton polygon of (5.16) is the upper convex hull of the points $A_k = (k, d_k) \in \mathbb{R}^2, k = 0, \ldots, m$. For each edge $e = [A_i, A_j]$ $(i < j)$, we denote by $-\kappa(e)$ its slope and we attach the algebraic equation $\chi_e(\alpha) = \sum_{A_k \in e} c_k \alpha^{k-i} = 0$ called the characteristic equation of e. Observe that the degrees of the characteristic equations sum up to the order m of the recurrence.

Theorem 5.2.8 (Poincaré, Perron, Kreuser)**.** *For each edge e of the Newton polygon of (5.16), let $\alpha_{e,1}, \alpha_{e,2}, \ldots$ be the solutions of the characteristic equation χ_e, counted with multiplicities.*

(a) If for each e, the moduli $|\alpha_{e,1}|, |\alpha_{e,2}|, \ldots$ are pairwise distinct, then any solution of (u_n) that is not ultimately 0 satisfies $u_{n+1}/u_n \sim_{n\to\infty} \alpha_{e,i} n^{\kappa(e)}$ for some e and i.

(b) If moreover, (5.16) is reversible (i.e., $p_0(n) \neq 0, \forall n \in \mathbb{N}$) then it admits a basis of solutions $\left(u^{[e,i]}\right)_{e,1\leqslant i\leqslant \deg \chi_e}$ such that

$$\frac{u_{n+1}^{[e,i]}}{u_n^{[e,i]}} \sim_{n\to\infty} \alpha_{e,i} n^{\kappa(e)}. \tag{5.17}$$

(c) If there exists e and $i \neq j$ such that $|\alpha_{e,i}| = |\alpha_{e,j}|$ results analogous to (a) and (b) hold with the weaker conclusion

$$\limsup_{n\to\infty} \left| \frac{u_n^{[e,i]}}{n!^{\kappa(e)}} \right|^{1/n} = \alpha_{e,i}. \tag{5.18}$$

Let us go back to the operator P of the Chebyshev recurrence given in Theorem 5.2.12. The symmetry properties of the coefficients of P can be translated to its Newton polygon, as defined above and drawn in Figure 5.1. Denote by $\alpha_s, \ldots, \alpha_1, \alpha_{-1}, \ldots, \alpha_{-s}$ the roots of the corresponding characteristic equations, the edges being traversed from left to right and the roots attached to a same edge in order of nondecreasing modulus, and by $-\kappa_s, \ldots, -\kappa_1, -\kappa_{-1}, \ldots, -\kappa_{-s}$ the slopes of the corresponding edges.

Proposition 5.2.9. *The slopes and the roots of the characteristic equations of Newton polygon of P verify: $\kappa_{-i} = -\kappa_i$ and $|\alpha_{-i}| = |\alpha_i|^{-1}$ for all i. Moreover, the characteristic equation associated to the horizontal edge (if it exists) does not have any root of modulus 1.*

Proof. From Theorem 5.2.1, item 2, we have $b_{-k}(n) = -b_k(-n)$, so the Newton polygon is symmetric with respect to the vertical axis, and $-\kappa_i = \kappa_{-i}$ for all i. Let $A_k = (k, \deg b_k)$ and the edge

$e_i = [A_{l(i)}, A_{r(i)}]$ the edge of slope κ_i. We denote by $\mathrm{lc}(b)$ the leading coefficient of b. For proving that $|\alpha_{-i}| = |\alpha_i|^{-1}$ we write characteristic equation of e_i:

$$\begin{aligned} \chi_i(\alpha) &= \textstyle\sum_{A_k \in e_i} \mathrm{lc}(b_k)\alpha^{k-l(i)} = \sum_{A_k \in e_i} (-1)^{1+\deg b_k} \mathrm{lc}(b_{-k})\alpha^{k-l(i)} \\ &= \pm \textstyle\sum_{A_k \in e_{-i}} \mathrm{lc}(b_k)((-1)^{\kappa_i}\alpha)^{-k-l(i)} = \pm\alpha^{l(i)-l(-i)}\chi_{-i}((-1)^{\kappa_i}\alpha^{-1}), \end{aligned} \tag{5.19}$$

since we have $\deg b_k - \deg b_{l(i)} = \kappa_i(k - l(i))$.

Finally, for proving that the characteristic equation associated to the horizontal edge (if it exists) does not have any root of modulus 1, it suffices to observe that it is given by $a_r(\frac{1}{2}(\alpha + \alpha^{-1}))$, where a_r is the leading coefficient in equation (5.1) and we supposed that $a_r(x) \neq 1$, for all $x \in [-1, 1]$. □

Hence, using Theorem 5.2.8, and Proposition 5.2.9, in some neighborhood of infinity the Chebyshev recurrence admits a basis of $2s$ solutions of the form given in Theorem 5.2.8. Moreover, s of these solutions are *convergent* and s are *divergent*. We call these solutions in some neighborhood at infinity *germs of solutions at infinity*. Specifically, we define:

Definition 5.2.10 (Germs of solutions at infinity.)**.** *A* germ of sequence *in the neighborhood of $+\infty$ is given by a definition domain $[\![N, \infty]\!]$ and a sequence $(u_n)_{n \geqslant N}$ modulo the identification of sequences which coincide on the definition domain. We call* germ of solution at infinity *(of a linear recurrence) a germ of sequence at $+\infty$ whose representatives satisfy the recurrence relation on their definition domain.*

We note for completeness that in the case of linear recurrences with polynomial coefficients, the germs of solutions at infinity coincide with the solutions of the recurrence when N is taken to be greater than the largest singularity of the recurrence, the singularities being in a finite number. So, the dimension of the space of the germs of solutions at infinity is equal to the order of the recurrence, i.e. $2s$ in our case.

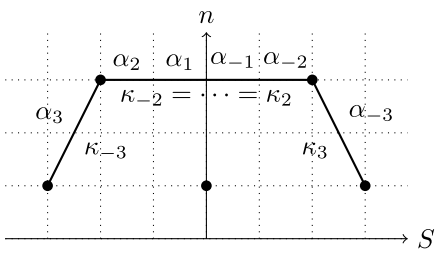


Figure 5.1: Newton polygon for Chebyshev recurrence.

Proposition 5.2.11. *The dimension of the space of symmetric solutions of a Chebyshev recurrence is $s + r$.*

Proof. In the Corollary 5.2.7, we have shown that this dimension is at least $s + r$. It remains to prove that this dimension is at most $s + r$. Let L be a differential operator of degree r and let P be the recurrence operator of order $2s$ in relation to L.

Let $\boldsymbol{S} = \{n \geqslant s : b_{-s}(n) = 0\}$ be the set of trailing singularities of P greater than s and let k be their cardinality.

We consider $(u_{|n|})_{n \in \mathbb{Z}}$ a symmetric sequence such that it verifies $(P \cdot u)_n = 0$ for all $n \in \mathbb{N} \setminus \boldsymbol{S}$. Using Remark 5.2.3, $(P \cdot u)_n = 0$ for $-n \in \mathbb{N} \setminus \boldsymbol{S}$. Then this sequence is completely characterized by a germ of solution at infinity of P together with the terms u_n where $n + s \in \boldsymbol{S}$. Specifically, taking $2s$ initial conditions $\{u_N, \ldots, u_{N+2s-1}\}$ for N sufficiently large, together with $\{u_{\sigma-s} : \sigma \in \boldsymbol{S}\}$, the remaining terms of nonnegative index are determined from the recurrence and the negative index ones by the symmetry property. Let E be the set of such sequences. It follows that the dimension

of E is $2s+k$. If we consider only the convergent sequences among these, i.e. $E \cap C$, then the dimension of this space is $s+k$ (since there are s convergent and s divergent germs at infinity).

In order for u to be a (symmetric) solution of the Chebyshev recurrence, we need also that $(P \cdot u)_\sigma = 0$, for $\sigma \in \boldsymbol{S}$ and $(P \cdot u)_n = 0$, for all $|n| < s$. From Corollary 5.2.6, since u is symmetric, $(P \cdot u)_n = 0$ for all $|n| < r$. So, u is a symmetric solution of the recurrence if and only if:

$$(P \cdot u)_n = 0, \quad n \in [\![r, s-1]\!] \cup \boldsymbol{S}. \tag{5.20}$$

According to the Theorem 5.2.1, if a symmetric convergent sequence y_n is canceled by P, the Chebyshev series $\sum y_n T_n(x)$ is canceled by L. The dimension of the space of solutions of L is r, then the dimension of the space of symmetric convergent solutions is r.

This implies that the rank of the system (5.20) taken over the space $E \cap C$ is $s+k-r$. Then the rank of the same system taken over E is at least $s+k-r$. So, the dimension of the space of symmetric solutions of the Chebyshev recurrence is at most $2s+k-(s+k-r) = s+r$. □

So, the structure of solutions of the Chebyshev recurrence is the following:
- The dimension of the space of germs of solutions at infinity is $2s$.
- The dimension of the space of germs of convergent solutions at infinity is s.
- The dimension of the space of symmetric convergent solutions is r.
- The dimension of the space of symmetric solutions is $s+r$.

5.3 Computing the Coefficients

5.3.1 Clenshaw's Algorithm Revisited

Using the properties of the Chebyshev recurrence given above, we revisit now a classical algorithm of Clenshaw [42] for numerically computing the coefficients of Chebyshev series solutions of LODE. We use ideas based on Miller's algorithm for the computation of a minimal solution of a linear recurrence using a "backward" unrolling of the recurrence as we already showed in Section 4.2.2. Now, that we have briefly seen the asymptotic behavior of solutions of linear recurrences, and in particular of Chebyshev recurrences, we can explain better Miller's algorithm. Let u_n be a sequence that verifies a linear recurrence, like (5.16) for instance. The idea is to compute "backwardly" the coefficients $u_N, u_{N-1}, \ldots, u_0$ starting with arbitrarily chosen initial conditions for N sufficiently big. In this way, roughly speaking, the behavior of the minimal ("most convergent") solution of the recurrence will be "caught" when having computed $u_0, u_1, \ldots, u_n$, for $n \ll N$. We have seen one advantage in Section 4.2.2: this computation method is much more stable than the "forward" unrolling of the recurrence. Moreover, and more interestingly, by computing s linearly independent test solutions we expect that their restrictions to $[\![0, n]\!]$ will form a vector space "close" to the one we would obtain with the "s most convergent" solutions.

Our algorithm takes as input both the degree d of the searched polynomial and a parameter N as a "high" index where we set the initial conditions. In practice, it was sufficient to take simply $N = d+s$.

5.3.2 Convergence

We now prove that our algorithm converges, under some simplifying assumptions. The proof is inspired by the analysis of the generalized Miller algorithm [172, 168].

These hypothesis are (Hypothesis **H**):

H1. The complex roots of the characteristic equations of each edge of the Newton polygon are simple and of distinct modulus (see Item (a) of Theorem 5.2.8).

1 **Algorithm:** `ComputeCoeffsClenshaw`$(L, \{\lambda_i(y) = l_i\}_{i=1}^r, d, N)$

Input: a differential operator L of order r, boundary conditions $\lambda_1(y) = \ell_1, \ldots, \lambda_r(y) = \ell_r$ as in (5.2),
a degree $d > s$, where $2s$ is the order of the Chebyshev recurrence associated to L,
an integer $N \geq d$.

Output: an approximation $\tilde{y}(x) = \sum_{n=-d}^{d} \tilde{y}_n T_n(x)$ of the corresponding solution y of $L \cdot y = 0$.

2 **Procedure:**

1 compute the Paszkowski operator $P = \sum_{k=-s}^{s} b_k(n) S^k$ associated to L

2 set $\mathbf{S} = \{n \in \mathbb{N} \mid (s \leqslant n \leqslant N) \wedge (b_{-s}(n) = 0)\}$ and $\mathbf{I} = \mathbf{S} \cup [\![N, N+s-1]\!]$

3 for $i \in \mathbf{I}$

4 for n from $N+s-1$ downto s

5 compute the coefficients $t_{i,n-s}$ using the recurrence relation if $n \notin \mathbf{I}$ with the initial conditions

$$\begin{cases} t_{i,i-s} &= 1 \\ t_{i,n-s} &= 0, \quad n \in \mathbf{I} \setminus \{i\} \end{cases}$$

6 set $\tilde{y}_n = \sum_{i \in \mathbf{I}} \eta_i t_{i,|n|}$ for $|n| \leqslant N$, and $\tilde{y}_n = 0$ for $|n| > N$ (hence $\eta_i = \tilde{y}_i$), and $\tilde{y}(x) = \sum_{n=-N}^{N} \tilde{y}_n T_n(x)$

7 solve for $(\eta_i)_{i \in \mathbf{I}}$ the linear system

$$\begin{cases} \lambda_k(\tilde{y}) = \ell_k, & 1 \leqslant k \leqslant r \\ b_{-s}(n)\tilde{y}_{n-s} + \cdots + b_s(n)\tilde{y}_{n+s} = 0, & n \in [\![r, s-1]\!] \cup \mathbf{S} \end{cases}$$

8 return $\sum_{n=-d}^{d} \tilde{y}_n T_n(x)$

Algorithm 5.3.1: Computation of the coefficients of Chebyshev series of D-finite functions based on Clenshaw's Algorithm.

H2. The operator P does not have trailing singularities at indexes $n \geqslant s$, that is, $S = \varnothing$ (see Item (b) of Theorem 5.2.8).

With these assumptions we can apply the strong form of Theorem 5.2.8 (Item (b)). We note that Hypothesis **H2** implies that any germ of solution at infinity extends to a solution defined on the whole set of nonnegative integers (but not necessarily a priori, a symmetric solution, or a solution on the whole $\mathbb{Z}$). This hypothesis also implies that the equation $L \cdot y$ does not have polynomial solutions.

If these assumptions are not fulfilled, Algorithm 5.3.1 still computes *some* approximation of $\pi_d(y)$, and its output may still be validated using the algorithm of Section 5.5 if it happens to be satisfactory.

We will prove in Proposition 5.3.2 that for a fixed degree d and as N tends to infinity, the computed polynomial converges at least exponentially fast towards the truncated Chebyshev series $\pi_d(y)$ of degree d of y.

The proof makes use of the following lemma:

Lemma 5.3.1. *Assume that $(e_{0,n})_n, \dots, (e_{s-1,n})_n$ are sequences such that*

$$\frac{e_{i,n+1}}{e_{i,n}} \sim_{n\to+\infty} \alpha_i n^{\kappa_i} \quad (\alpha_i \in \mathbb{C} \setminus \{0\}, \kappa_i \in \mathbb{Q})$$

with $\kappa_0 \leqslant \kappa_1 \leqslant \cdots \leqslant \kappa_{s-1}$ and $\kappa_i = \kappa_j \to \alpha_i \neq \alpha_j$. Then the Casorati determinant

$$C(n) = \begin{vmatrix} e_{0,n} & e_{1,n} & \cdots & e_{s-1,n} \\ e_{0,n+1} & & & e_{s-1,n+1} \\ \vdots & & & \vdots \\ e_{0,n+s-1} & e_{1,n+s-1} & \cdots & e_{s-1,n+s-1} \end{vmatrix}$$

satisfies

$$C(n) \sim_{n\to\infty} e_{0,n}e_{1,n+1}\cdots e_{s-1,n+s-1} \prod_{\substack{i<j \\ \kappa_i=\kappa_j}} \Big(\frac{\alpha_i}{\alpha_j} - 1\Big).$$

Proof. Write $C(n) = e_{0,n}e_{1,n+1}\cdots e_{s-1,n+s-1}C'(n)$. Then

$$C'(n) = \det\Big(\frac{e_{j,n+i}}{e_{j,n+j}}\Big)_{0\leqslant i,j<s} = \sum_{\sigma\in\mathfrak{S}_s} \varepsilon(\sigma) \prod_{j=0}^{s-1} \frac{e_{j,n+\sigma(j)}}{e_{j,n+j}},$$

where the term of index σ of the sum grows like $n^{\sum_{j=0}^{s-1}(\sigma(j)-j)\kappa_j}$. The dominant terms are those such that $\sum_{j=0}^{s-1}\sigma(j)\kappa_j = \sum_{j=0}^{s-1} j\kappa_{\sigma(j)}$ is maximal, that is, such that the $\kappa_{\sigma(j)}$ are in nondecreasing order. Using the Iverson bracket, *i.e.* $[P] = 1$ if P is true and $[P] = 0$ otherwise, this translates into

$$\begin{aligned} C'(n) &= \sum_{\sigma\in\mathfrak{S}_s} \varepsilon(\sigma) \prod_{j=0}^{s-1} [\kappa_{\sigma(j)} = \kappa_j] \frac{e_{j,n+\sigma(j)}}{e_{j,n+j}} + o(1) \\ &= \det\Big([\kappa_i = \kappa_j]\frac{e_{j,n+i}}{e_{j,n+j}}\Big)_{0\leq i,j<s} + o(1) \\ &= \prod_{\kappa\in\{\kappa_i\}} \det\Big(\frac{e_{j,n+i}}{e_{j,n+j}}\Big)_{\kappa_i=\kappa_j=\kappa} + o(1). \end{aligned}$$

Each determinant in the last product has the form

$$\det\Big(\frac{e_{j,n+i}}{e_{j,n+j}}\Big) = \det\Big(n^{(j-i)\kappa}\frac{e_{j,n+i}}{e_{j,n+j}}\Big) \to_{n\to\infty} \det(\alpha_j^{i-j}) = \prod_{\substack{i<j \\ \kappa_i=\kappa_j=\kappa}} \Big(\frac{\alpha_i}{\alpha_j} - 1\Big) \neq 0$$

whence the result. □

Proposition 5.3.2. *Assume Hypothesis (**H**). In the notations of Algorithm 5.3.1, fix L as well as the boundary conditions $\lambda_i(y) = \ell_i$, write $y(x) = \sum_{n=-\infty}^{\infty} y_n T_n(x)$ and let $y_n^{(N)} = \tilde{y}_n$, $|n| \leqslant N$, be the approximate Chebyshev coefficients computed by the algorithm (run in exact arithmetic), as a function of the remaining input parameter N. As $N \to \infty$ it holds that*

$$\max_{n=-N}^{N} (y_n^{(N)} - y_n) = O(N^t e_{1,N})$$

for some t independent of N.

Proof. Let us first describe the output of Algorithm 5.3.1. The sequence $(y_n^{(N)})_{n=-N}^{N}$ extends to a solution $(y_n^{(N)})_{n\in\mathbb{Z}}$ of $P \cdot y^{(N)} = 0$ characterized by the conditions $y_N^{(N)} = \cdots = y_{N+s-1}^{(N)} = 0$ from Step 5, and the linear system solved in Step 7. By writing the linear forms $\lambda_1, \ldots, \lambda_r : C \to \mathbb{C}$ that express the boundary conditions (5.2) as $\lambda_i(y) = \sum_{n=-\infty}^{\infty} \lambda_{i,n} y_n$, we may define "truncations" $\lambda_i^{(N)}(y) = \sum_{n=-N}^{N} \lambda_{i,n} y_n$ that make sense even for divergent series. Abusing notation slightly, we apply the λ_i and $\lambda_i^{(N)}$ indifferently to functions, formal Chebyshev series or their coefficient sequences. We also introduce linear forms $\lambda_{r+1} = \lambda_{r+1}^{(N)}, \ldots, \lambda_s = \lambda_s^{(N)}$ to write the last $s - r$ equations (*i.e.*, the symmetry constraints on Chebyshev expansions) in the same form as the first r, so that the system (7) translates into

$$\lambda_i^{(N)}(y^{(N)}) = \sum_{n=-N}^{N} \lambda_{i,n} y_n^{(N)} = \ell_i, \quad 1 \leqslant i \leqslant s. \tag{5.21}$$

Now let

$$\Delta^{(N)} = \begin{vmatrix} e_{1,N} & \cdots & e_{s,N} & e_{-1,N} & \cdots & e_{-s,N} \\ \vdots & & \vdots & \vdots & & \vdots \\ e_{1,N+s-1} & \cdots & e_{s,N+s-1} & e_{-1,N+s-1} & \cdots & e_{-s,N+s-1} \\ \lambda_1^{(N)}(e_1) & \cdots & \lambda_1^{(N)}(e_s) & \lambda_1^{(N)}(e_{-1}) & \cdots & \lambda_1^{(N)}(e_{-s}) \\ \vdots & & \vdots & \vdots & & \vdots \\ \lambda_s^{(N)}(e_1) & \cdots & \lambda_s^{(N)}(e_s) & \lambda_s^{(N)}(e_{-1}) & \cdots & \lambda_s^{(N)}(e_{-s}) \end{vmatrix}, \tag{5.22}$$

and let $\Delta_j^{(N)}$ be the same determinant with the column involving e_j replaced by $(0, \ldots, 0, \ell_1, \ldots, \ell_s)^{\mathrm{T}}$. By Cramer's rule, the sequence $(y_n^{(N)})_n$ decomposes on the basis $(e_j)_{j=-s}^{s}$ of $\ker P \subseteq \mathbb{C}^{\mathbb{Z}}$ as

$$y^{(N)} = \sum_{k=-s}^{s} \gamma_k^{(N)} e_k, \qquad \gamma_k^{(N)} = \frac{\Delta_k^{(N)}}{\Delta^{(N)}}. \tag{5.23}$$

The "exact" sequence of Chebyshev coefficients of the function y defined by the input is likewise given by

$$y = \sum_{k=1}^{s} \gamma_k e_k, \qquad \gamma_k = \frac{\Delta_k}{\Delta}, \tag{5.24}$$

where $\Delta = \det(\lambda_i(e_j))_{1\leqslant i,j\leqslant s}$ and Δ_j denotes the determinant Δ with the j-th column replaced by $(\ell_1, \ldots, \ell_s)^{\mathrm{T}}$.

Our goal is now to prove that $\gamma_k^{(N)} \to \gamma_k$ fast as $N \to \infty$. We decompose $\Delta^{(N)}$ into four blocks as follows:

$$\Delta^{(N)} = \begin{vmatrix} A & B \\ C & D \end{vmatrix}.$$

The corresponding modified blocks in $\Delta_k^{(N)}$ are denoted A_k, B_k, C_k, D_k. Notice that these matrices depend on N, although we dropped the explicit index for readability.

The blocks A and B (by Lemma 5.3.1) as well as C (since $\det C \to \Delta \neq 0$ when $n \to \infty$) are nonsingular for large N. The Schur complement formula implies that

$$\Delta^{(N)} = -\det(B)\det(C)\det(I - C^{-1}DB^{-1}A).$$

Setting $\mathbf{e}_j = (e_{j,N}, \dots, e_{j,N+s-1})^{\mathrm{T}}$, the entry at position (i,j) in the matrix $B^{-1}A$ satisfies

$$\begin{aligned}
(B^{-1}A)_{i,j} &= \frac{\det(\mathbf{e}_{-1}, \dots, \mathbf{e}_{-i+1}, \mathbf{e}_j, \mathbf{e}_{-i-1}, \dots, \mathbf{e}_{-s})}{\det B} \\
&= \frac{(-1)^{i-1}\det(\mathbf{e}_j, \mathbf{e}_{-1}, \dots, \widehat{\mathbf{e}_{-i}}, \dots, \mathbf{e}_{-s})}{\det(\mathbf{e}_{-1}, \dots, \mathbf{e}_{-s})} \\
&= O\left(\frac{e_{j,N}e_{-1,N+1}\dots e_{-i+1,N+i-1}e_{-i-1,N+i}\dots e_{-s,N+s-1}}{e_{-1,N}e_{-2,N+1}\dots e_{-s,N+s-1}}\right) \\
&= O\left(N^{\kappa_{-1}+\dots+\kappa_{-i+1}-(i-1)\kappa_i}\frac{e_{j,N}}{e_{-i,N}}\right) \\
&= O\left(\frac{e_{j,N}}{e_{-i,N}}\right)
\end{aligned}$$

as $N \to \infty$ by Lemma 5.3.1. Given our assumptions on the boundary conditions (5.2), we have $\lambda_{i,n} = O_{n\to\pm\infty}(n^r)$ in (5.21), so that the entries of D satisfy $D_{i,j} = \lambda_i^{(N)}(e_{-j}) = O(N^r e_{-j,N})$. This yields the estimate $(DB^{-1}A)_{i,j} = O(N^r e_{j,N})$ for the j-th column of $DB^{-1}A$. Since

$$C_{i,j} = \lambda_i^{(N)}(e_j) = \lambda_i(e_j) + O(N^r e_{j,N}),$$

we get $(C^{-1}DB^{-1}A)_{i,j} = O(N^r e_{j,N})$ as well, and

$$\begin{aligned}
\Delta^{(N)} &= -\det(B)\det(C)(1 - \operatorname{tr}(C^{-1}DB^{-1}A) + O(\|C^{-1}DB^{-1}A\|^2)) \\
&= -\det(B)(\Delta + O(N^r e_{1,N})).
\end{aligned}$$

We turn to the modified determinants Δ_k. For $k > 0$, the same reasoning as above (except that C_k may now be singular) leads to *

$$\begin{aligned}
\Delta_k^{(N)} &= -\det(B)\det(C_k - DB^{-1}A_k) \\
&= -\det(B)(\det(C_k) + O(N^r e_{1,N})) \\
&= -\det(B)(\Delta_k + O(N^r e_{1,N})),
\end{aligned}$$

hence

$$\gamma_k^{(N)} = \frac{\Delta_k^{(N)}}{\Delta^{(N)}} = \gamma_k + O(N^r e_{1,N}), \qquad k > 0. \tag{5.25}$$

In the case $k < 0$, write

$$\Delta_k^{(N)} = -\det(C)\det(B_k - AC^{-1}D_k).$$

The natural entrywise bounds on A and D yield $(C^{-1}D_k)_{i,j} = O(N^r e_{-j,N+s-1})$ and from there

$$(AC^{-1}D_k)_{i,j} = O(N^r e_{1,N}e_{-j,N+s-1}) = o(e_{-j,N}),$$

so that

$$(B_k + AC^{-1}D_k)_{i,j} \sim e_{-j,N+i-1}, \qquad j \neq -k.$$

*. we even have $\Delta_1^{(N)} = \Delta_1 + O(N^r e_{2,N})$

For $j = -k$ however, the j-th column of B_k is zero and that of D_k is constant, hence

$$(B_k + AC^{-1}D_k)_{i,j} = O(e_{1,N}), \qquad j = -k.$$

It follows that

$$\begin{aligned}\det(B_k + AC^{-1}D_k) &= O(e_{-1,N+s-1} \cdots \widehat{e_{k,N+s-1}} \cdots e_{-s,N+s-1}\, e_{1,N}) \\ &= O\left(\frac{N^\tau \det(B)}{e_{k,N}} e_{1,N}\right)\end{aligned}$$

with $\tau \leqslant \sum_{j\neq k}(s-j)\kappa_{-j}$, and

$$\begin{aligned}\gamma_k^{(N)} = \frac{\Delta_k^{(N)}}{\Delta^{(N)}} &= \frac{-\det(B)\det(C)N^\tau}{-\det(B)\det(C)(1+O(e_{1,N}))}\frac{e_{1,N}}{e_{k,N}} \\ &= O\left(N^\tau \frac{e_{1,N}}{e_{k,N}}\right), \qquad k < 0.\end{aligned} \tag{5.26}$$

Combining (5.23), (5.24) with (5.25), (5.26) finally yields

$$y_n^{(N)} = y_n + O\left(N^{\max(r,\tau)} e_{1,N} \sum_{k=1}^{s}\left(e_{k,n} + \frac{e_{-k,n}}{e_{-k,N}}\right)\right)$$

as $N \to \infty$, uniformly in n. □

5.3.3 Variants

Another popular method for the approximate computation of Chebyshev expansions is Lánczos' tau method [89, 90]. It has been observed by Fox [64] and later in greater generality (and different language) by El Daou, Ortiz and Samara [58] that both methods are in fact equivalent, in the sense that they may be cast into a common framework and tweaked to give exactly the same result. We now outline how the use of the Chebyshev recurrence fits into the picture. This sheds another light on Algorithm 5.3.1 and indicates how the Chebyshev recurrence may be used in the context of the tau method.

As in the previous sections, consider a differential equation $L \cdot y = 0$ of order r, with polynomial coefficients, to some solution of which a polynomial approximation of degree d is seeked. Assume for simplicity that there are no nontrivial polynomial solutions, *i.e.*, $(\ker L) \cap \mathbb{C}[x] = \{0\}$.

In a nutshell, the tau method works as follows. The first step is to compute $L \cdot p$ where p is a polynomial of degree d with indeterminate coefficients. Since $(\ker L) \cap \mathbb{C}[x] = \{0\}$, the result has degree greater than d. One then introduces additional unknowns $\tau_{d+1}, \ldots, \tau_{d+m}$ in such number that the system

$$\begin{aligned} &L \cdot p = \tau_{d+1}T_{d+1} + \cdots + \tau_{d+m}T_{d+m}, \\ &\lambda_i(p) = \ell_i, \qquad (1 \leqslant i \leqslant r),\end{aligned} \tag{5.27}$$

has a (preferably unique) solution. The output is the value of p obtained by solving this system; it is an exact solution of the projection $\pi_d(L \cdot y) = 0$ of the original differential equation.

Now let $p = \sum_{n=-d}^{d} p_n T_n$ and extend the sequence (τ_n) by putting $\tau_n = 0$ for $n \notin [\![d+1, d+m]\!]$ and $\tau_{-n} = \tau_n$. It follows from (5.27) that $P \cdot (p_n) = \frac{1}{2}Q \cdot (\tau_n)$ where P and Q are the recurrence operators given by Theorem 5.2.1. Denoting $\operatorname{Supp} u = \{n | u_n \neq 0\}$, we also see from the explicit expression of Q that $\operatorname{Supp}(Q \cdot \tau) \subseteq [\![d, d+m+1]\!]$. Hence the coefficients p_n of the result of the tau method are given by the Chebyshev recurrence, starting from a small number of initial conditions given near the index $|n| = d$.

"Conversely," consider the polynomial $\tilde{y}$ computed in Algorithm 5.3.1 and let $v = \sum_n v_n T_n = L \cdot \tilde{y}$. We have $P \cdot \tilde{y} = Q \cdot v$ by Theorem 5.2.1. But the definition of $\tilde{y}$ in the algorithm also implies that $(P \cdot \tilde{y})_n = 0$ when $|n| \leqslant N - s$ (since the $\tilde{y}_n$, $|n| \leqslant N$ are linear combinations of sequences $(t_{i,n})_{|n|\leqslant N}$ recursively computed using the recurrence $P \cdot t_i = 0$) or $|n| > N+s$ (since $\tilde{y}_n = 0$ for $|n| > N$), so that $\mathrm{Supp}(Q \cdot v) \subseteq [\![N - s, N + s - 1]\!]$. It can be checked that the recurrence operator P associated to $L = (\frac{\mathrm{d}}{\mathrm{d}x})^r$ by Paszkowski's algorithm is $P = \delta_r(n)$: indeed, in the language of [8], it must satisfy $Q^{-1}P = I^{-r}$. Thus $\delta_r(n) \cdot u = Q \cdot v$ is equivalent to $u^{(r)} = v$, whence

$$v(x) = \frac{\mathrm{d}^r}{\mathrm{d}x^r} \sum_{|n|>r} \frac{(P \cdot \tilde{y})_n}{\delta_r(n)} T_n(x) = \sum_{N-s\leqslant|n|<N+s} \frac{(P \cdot \tilde{y})_n}{\delta_r(n)} T_n^{(r)}(x). \tag{5.28}$$

We see that the output $\tilde{y}(x)$ of Algorithm 5.3.1 satisfies an inhomogeneous differential equation of the form $L \cdot \tilde{y} = \tau_{N-s} T_n^{(r)}(x) + \cdots + \tau_{N+s-1} T_n^{(r)}(x)$. (However, the support of the sequence (v_n) itself is not sparse in general.)

This point of view also leads us to the following observation. In comparison, the best known arithmetic complexity bound for the conversion of arbitrary polynomials of degree d from the Chebyshev basis to the monomial basis is $O(M(n))$, where M stands for the cost of polynomial multiplication [128, 20].

Proposition 5.3.3. *The expression on the monomial basis of the polynomial $\tilde{y}(x)$ returned by Algorithm 5.3.1 may be computed in $O(d)$ arithmetic operations.*

Proof. As already mentioned, the Taylor series expansion of a function that satisfies a LODE with polynomial coefficients obeys a linear recurrence relation with polynomial coefficients. In the case of an inhomogeneous equation $L \cdot u = v$, the recurrence operator does not depend on v, and the right-hand side of the recurrence is the coefficient sequence of v. Now $\tilde{y}$ satisfies $L \cdot \tilde{y} = v$ where v is given by (5.28). The coefficients $(P \cdot \tilde{y})_n/\delta_r(n)$ of (5.28) are easy to compute from the last few Chebyshev coefficients of $\tilde{y}$. One deduces the coefficients v_n in linear time by applying repeatedly the non-homogeneous recurrence relation

$$T'_{n-1}(x) = -T'_{n+1}(x) + 2xT'_n(x) + 2T_n(x) \tag{5.29}$$

obtained by differentiation of the equation (5.6), and finally those of the expansion of $\tilde{y}$ on the monomial basis using the recurrence relation they satisfy. □

5.4 Chebyshev Expansions of Rational Functions

This section is devoted to the same problems as the remainder of the chapter, only restricted to the case where $y(x)$ is a rational function. We are interested in computing a recurrence relation on the coefficients y_n of the Chebyshev expansion of a function y, using this recurrence to obtain a good uniform polynomial approximation of $y(x)$ on $[-1, 1]$, and certifying the accuracy of this approximation. All this will be useful in the validation part of our main algorithm.

Our primary tool is the change of variable $x = \frac{1}{2}(z + z^{-1})$ followed by partial fraction decomposition. Similar ideas have been used in the past with goals only slightly different from ours, like the computation of y_n in closed form [57, 103]. Indeed, the sequence $(y_n)_{n\in\mathbb{N}}$ turns out to obey a recurrence with *constant* coefficients. Finding this recurrence or a closed form of y_n are essentially equivalent problems. However, we need results regarding the cost of the algorithms that do not seem to appear in the literature. Our main concern in this respect is to avoid conversions from polynomial and series from the monomial to the Chebyshev basis and back. We also require simple error bounds on the result. To summarize, in fact, we are interested in computing a Chebyshev model for a rational function in a linear time (see Proposition 5.4.6 for the exact formulation of the result).

5.4.1 Recurrence and Explicit Expression

Let $y(x) = a(x)/b(x) \in \mathbb{Q}[x]$ be a rational function without any pole in $[-1, 1]$. As usual, we denote by $(y_n)_{n\in\mathbb{Z}}$, $(a_n)_{n\in\mathbb{Z}}$ and $(b_n)_{n\in\mathbb{Z}}$ the symmetric Chebyshev coefficient sequences of y, a and b.

Proposition 5.4.1. *The Chebyshev coefficient sequence $(y_n)_{n\in\mathbb{Z}}$ obeys the recurrence relation with constant coefficients $b(\frac{1}{2}(S + S^{-1})) \cdot (y_n) = (a_n)$.*

Proof. This is actually the limit case $r = 0$ of Theorem 5.2.1, but a direct proof is very easy: just write

$$\sum_{i=-\infty}^{\infty} b_i z^i \sum_{n=-\infty}^{\infty} y_n z^n = \sum_{n=-\infty}^{\infty} \Big(\sum_{i=-\infty}^{\infty} b_i y_{n-i} \Big) z^n = \sum_{n=-\infty}^{\infty} a_n z^n, \quad x = \frac{z + z^{-1}}{2},$$

and identify the coefficients of like powers of z. □

The recurrence suffers from the existence of divergent solutions and lack of easily accessible initial values discussed in §5.2. However, we can explicitly separate the positive powers of z from the negative ones in the Laurent series expansion

$$\hat{y}(z) = y\Big(\frac{z + z^{-1}}{2}\Big) = \sum_{n=-\infty}^{\infty} y_n z^n, \qquad \rho^{-1} < |z| < \rho, \tag{5.30}$$

using partial fraction decomposition. From the computational point of view, it is better to start with the square-free factorization of the denominator of $\tilde{y}$:

$$\beta(z) = z^{\deg b} b\left(\frac{z + z^{-1}}{2}\right) = \beta_1(z)\beta_2(z)^2 \cdots \beta_k(z)^k \tag{5.31}$$

and write the full partial fraction decomposition of $\hat{y}(z)$ in the form

$$\hat{y}(z) = q(z) + \sum_{i=1}^{k} \sum_{\beta_i(\zeta)=0} \sum_{j=1}^{i} \frac{h_{i,j}(\zeta)}{(\zeta - z)^j}, \qquad q(z) = \sum_n q_n z^n \in \mathbb{Q}[z], \quad h_{i,j} \in \mathbb{Q}(z). \tag{5.32}$$

The $h_{i,j}$ may be computed efficiently using the Bronstein-Salvy algorithm [30] (see also [72]).

We obtain an identity of the form (5.30) by expanding the partial fractions corresponding to poles ζ with $|\zeta| > 1$ in power series about the origin, and those with $|\zeta| < 1$ about infinity. The expansion at infinity of

$$\frac{h_{i,j}(\zeta)}{(\zeta - z)^j} = \frac{(-1)^j z^{-j} h_{i,j}(\zeta)}{(1 - \zeta z^{-1})^j}$$

does not contribute to the coefficients of z^n, $n \geqslant 0$ in the complete Laurent series. It follows from the uniqueness of the Laurent expansion of $\hat{y}$ on the annulus $\rho^{-1} < |z| < \rho$ that *

$$\sum_{n=0}^{\infty} y_n z^n = q(z) + \sum_{i=1}^{k} \sum_{\substack{\beta_i(\zeta) = 0 \\ |\zeta| > 1}} \sum_{j=1}^{i} \frac{h_{i,j}(\zeta)}{(\zeta - z)^j}. \tag{5.33}$$

*. To prevent confusion, it may be worth pointing out that in the expression

$$\tilde{y}(z) = q(z) + \sum_{i=1}^{k} \sum_{\substack{\beta_i(\zeta) = 0 \\ |\zeta| > 1}} \sum_{j=1}^{i} \Big(\frac{h_{i,j}(\zeta)}{(\zeta - z)^j} + \frac{h_{i,j}(\zeta^{-1})}{(\zeta^{-1} - z)^j} \Big)$$

the Laurent expansion of a single term of the form $\frac{h_{i,j}(\zeta)}{(\zeta-z)^j} + \frac{h_{i,j}(\zeta^{-1})}{(\zeta^{-1}-z)^j}$ is *not* symmetric for $j > 1$, even if $q(z) = 0$.

We now extract the coefficient of z^n in (5.33) and use the symmetry of $(y_n)_{n\in\mathbb{Z}}$ to get an explicit expression of y_n in terms of the roots of $b(\frac{1}{2}(z+z^{-1}))$.

Proposition 5.4.2. *The coefficients of the Chebyshev expansion $y(x) = \sum_{n=-\infty}^{\infty} y_{|n|}T_n(x)$ are*

$$y_n = q_n + \sum_{i=1}^{k}\sum_{j=1}^{i} \sum_{\substack{\beta_i(\zeta)=0\\ |\zeta|>1}} \binom{n+j-1}{j-1} h_{i,j}(\zeta)\zeta^{-n-j} \qquad (n \geqslant 0) \tag{5.34}$$

where the $q_n \in \mathbb{Q}$, $\beta_i \in \mathbb{Q}[z]$ and $h_{i,j} \in \mathbb{Q}(z)$ are defined in Equations (5.31) and (5.32).

Note that (5.33) also yields a recurrence of order $\deg b$ on $(y_n)_{n\in\mathbb{N}}$, instead of $2\deg b$ for that from Proposition 5.4.1, but now with algebraic rather than rational coefficients in general. Besides, we can now explicitly bound the error in truncating the Chebyshev expansion of y.

5.4.2 Bounding the truncation error

Proposition 5.4.3. *Let $y \in \mathbb{Q}(x)$ have no pole within the elliptic disk $\bar{\varepsilon}_\rho$ (cf. Definition 4.2.19). Assume again the notations from (5.31) and (5.32). For all $d \geqslant \deg q$, it holds that*

$$\|\sum_{n>d} y_n T_n\|_\infty \leqslant \sum_{i=1}^{k}\sum_{j=1}^{i} \sum_{\substack{\beta_i(\zeta)=0\\ |\zeta|>1}} \frac{|h_{i,j}(\zeta)|(d+2)^{j-1}}{(|\zeta|-1)^j}|\zeta|^{-d-1} = O(d^{\deg b}\rho^{-d}).$$

Proof. We have $\|\sum_{n>d} y_n T_n\|_\infty \leqslant \sum_{n>d}|y_n|$ because $\|T_n\|_\infty \leqslant 1$ for all n. Using the inequality

$$\sum_{n>d}\binom{n+j-1}{j-1}t^{n+j} \leqslant (d+2)^{j-1}t^{d+1}\sum_{n=0}^{\infty}\binom{n+j-1}{j-1}t^{n+j} = \frac{(d+2)^{j-1}t^{d+j+1}}{(1-t)^j}$$

for $t<1$, the explicit expression from Proposition 5.4.2 yields

$$\sum_{n>d}|y_n| \leqslant \sum_{n>d}\sum_{i=1}^{k}\sum_{j=1}^{i}\sum_{\substack{\beta_i(\zeta)=0\\ |\zeta|>1}}\binom{n+j-1}{j-1}|h_{i,j}(\zeta)||\zeta|^{-n-j} \tag{5.35}$$

$$\leqslant \sum_{i=1}^{k}\sum_{j=1}^{i}\sum_{\substack{\beta_i(\zeta)=0\\ |\zeta|>1}}\frac{|h_{i,j}(\zeta)|(d+2)^{j-1}}{(|\zeta|-1)^j}|\zeta|^{-d-1}. \tag{5.36}$$

The asymptotic estimate follows since $|\zeta|>1$ actually implies $|\zeta|>\rho$ when $b(\frac{1}{2}(\zeta+\zeta^{-1}))=0$. □

5.4.3 Computation

There remains to check that the previous results really translate into a linear time algorithm. We first state a lemma regarding polynomial division with remainder. The naïve algorithm for this task [166, Algorithm 2.5] runs in linear time with respect to the degree of the dividend when the divisor is fixed. Its input and output are usually represented by their coefficients on the monomial basis, but the algorithm is easily adapted to work on other polynomial bases.

Lemma 5.4.4. *The division with remainder $a = bq + r$ ($\deg r < \deg b$) where $a, b, q, r \in \mathbb{Q}[x]$ are represented on the Chebyshev basis may be performed in $O(\deg a)$ operations for fixed b.*

Proof Assume $n = \deg a > \deg b = m$. The naïve polynomial division algorithm mainly relies on the fact that $\deg(a - b_m^{-1} a_n x^{n-m} b) < n$ where $a = \sum_i a_i x^i$ and $b = \sum_i b_i x^i$. From the multiplication formula $2T_n T_m = T_{n+m} + T_{n-m}$ follows the analogous inequality $\deg(a - 2b_m^{-1} a_n T_{n-m} b) < n$ where a_k, b_k now denote the coefficients of a and b on the Chebyshev basis. Performing the whole computation on that basis amounts to replace each of the assignments $a \leftarrow a - b_m^{-1} a_n x^{n-m} b$ repeatedly done by the classical algorithm by $a \leftarrow a - 2b_m^{-1} a_n T_{n-m} b$. Since the polynomial $T_{n-m} b$ has at most $2m$ nonzero coefficients, each of these steps takes constant time with respect to n. We do at most $n - m$ such assignments, hence the overall complexity is $O(n)$. □

Remark 5.4.5. *For completeness, we recall that the "naive" product represented in Chebyshev basis, of two polynomials in Chebyshev basis $a, b \in \mathbb{Q}[x]$ can be performed in $O((\deg a)(\deg b))$ operations (see (4.37)).*

Algorithm

1 **Algorithm:** `ChebExpandRatFrac`(y, f, ε)

Input: Coefficients of the polynomial $f = \sum_{i=-d}^{d} \tilde{f}_i T_i(x)$ of degree d in Chebyshev basis,
a rational fraction $y(x) = a(x)/b(x)$,
an error bound ε.

Output: Coefficients of an approximation $\tilde{y}(x) = \sum_{i=-k}^{k} \tilde{y}_i T_i(x)$ of fy, such that
$\|\tilde{y} - fy\|_\infty \leqslant \varepsilon$

2 **Procedure:**

1 convert a and b to Chebyshev basis;

2 compute the polynomial $g = af$ in Chebyshev basis;

3 compute the quotient q and the remainder r with euclidean division of g by b;

4 compute the partial fraction decomposition of $\hat{w}(z) = w(x) = r(x)/b(x)$, where $x = \frac{z+z^{-1}}{2}$ using Bronstein-Salvy Algorithm and obtain that of $\hat{y}(z) = q(x) + w(x)$ (cf. (5.32));

5 find $d' \geqslant \deg q$ such that $\left\| \sum_{n>d'} y_n T_n \right\|_\infty \leqslant \varepsilon/4$ using Proposition 5.4.3;

6 compute ρ_- and ρ_+ such that $\beta_i(\zeta) = 0 \wedge |\zeta| > 1 \Rightarrow 1 < \rho_- \leqslant |\zeta| \leqslant \rho_+$;

7 compute $M \geqslant \sum_{i=1}^{k} \sum_{j=1}^{i} j(\deg \beta_i) \sup_{\rho_- \leqslant |\zeta| \leqslant \rho_+} \left(|h'_{i,j}(\zeta)| + |\zeta^{-1} h_{i,j}(\zeta)| \right) \rho_-^{-j}$;

8 $\varepsilon' := \min\left(\rho_- - 1, M^{-1}\left(1 - \rho_-^{-1}\right)^{D+1} \frac{\varepsilon}{4}\right)$, with $D = \deg b$;

9 compute approximations $\tilde{\zeta} \in \mathbb{Q}[i]$ of the roots ζ of β_i such that $\left|\tilde{\zeta} - \zeta\right| < \varepsilon'$;

10 for $0 \leqslant n \leqslant d'$

11 $$\tilde{y}_n = q_n + \mathrm{Re}\left(\sum_{i=1}^{k} \sum_{j=1}^{i} \sum_{\substack{\beta_i(\zeta) = 0 \\ |\zeta| > 1}} \binom{n+j-1}{j-1} h_{i,j}(\tilde{\zeta}) \tilde{\zeta}^{-n-j} \right);$$

12 return

$$\tilde{y}(x) = \sum_{n=-d'}^{d'} \tilde{y}_n T_n(x)$$

Algorithm 5.4.1: Linear time computation of Chebyshev models for rational functions

Proposition 5.4.6. *Algorithm 5.4.1 is correct. Its arithmetic complexity is $O(d + \log(\varepsilon^{-1}))$, all the other parameters being fixed.*

Proof. Firstly, we prove that $\|\tilde{y} - y\|_\infty \leqslant \varepsilon$.

Let $A = \{\zeta : \rho_- \leqslant |\zeta| \leqslant \rho_+\}$ and $M_0 = \sup_{\zeta \in A} |h'_{i,j}(\zeta)|$, $M_1 = \sup_{\zeta \in A} |\zeta^{-1} h_{i,j}(\zeta)|$.

For all $\zeta \in A$,

$$\left|\left(h_{i,j}(\zeta)\zeta^{-n-j}\right)'\right| \leqslant (M_0 + (n+j)M_1)\,|\zeta|^{-n-j} \leqslant (n+j)(M_0 + M_1)\rho_-^{-n-j}. \tag{5.37}$$

From Proposition 5.4.2, observing that condition $|\zeta - \tilde{\zeta}| < \rho_- - 1$ implies that $\left[\zeta, \tilde{\zeta}\right] \subseteq A$, and using (5.37), we have:

$$\begin{aligned}
|y_n - \tilde{y}_n| &\leqslant \sum_{i=1}^{k}\sum_{j=1}^{i} \sum_{\substack{\beta_i(\zeta) = 0 \\ |\zeta| > 1}} \binom{n+j-1}{j-1} \left|h_{i,j}(\zeta)\zeta^{-n-j} - h_{i,j}(\tilde{\zeta})\tilde{\zeta}^{-n-j}\right| \\
&\leqslant \sum_{i=1}^{k}\sum_{j=1}^{i} j(\deg \beta_i)\binom{n+j}{j}\,(M_0 + M_1)\,{\rho_-}^{-n-j}\varepsilon' \\
&\leqslant M\binom{n+D}{D}{\rho_-}^{-n}\varepsilon'.
\end{aligned}$$

Whence,

$$\|y_n - \tilde{y}_n\|_\infty \leqslant \sum_{n=-d'}^{d'} |y_n - \tilde{y}_n| + 2\left\|\sum_{n>d'} y_n T_n\right\|_\infty \leqslant \frac{2M\varepsilon'}{(1-\rho_-^{-1})^{D+1}} + 2\frac{\varepsilon}{4} \leqslant \varepsilon.$$

Complexity analysis. It is easy to see that steps 1, 4-8 have a constant cost. Steps 2 and 3 are in $O(d)$, using Lemma 5.4.4 and Remark 5.4.5. For step 9, it is known [127, Theorem 1.1(d)] that the roots of a polynomial with integer coefficients can be approximated with of absolute accuracy η in $O(\eta^{-1})$ arithmetic operations. We note that the binary complexity of the algorithm is also almost linear with respect to ε^{-1}. Since M does not depend neither on ε nor on d, we have $\varepsilon' = \Omega(\varepsilon)$ and hence step 9 is in $O(\varepsilon^{-1})$. Finally each iteration of the loop has a constant cost by keeping track of the values $\tilde{\zeta}^{-n}$ from one iteration to another. □

5.5 Error Bounds / Validation

In this section we assume that we have already computed (see Algorithm 5.3.1) a polynomial $p(x) = \sum_{n=0}^{d} \tilde{y}_n T_n(x)$ of degree d, which is an approximate solution for a D-finite function y, specified by a LODE and suitable boundary conditions. As stated in our goal in Introduction, we are now interested in computing a "small" bound B such that $|y(x) - p(x)| \leqslant B$ for all $x \in [-1, 1]$. The main idea for computing this bound, is to use an interval method, also called validated or verified method for ODE (see the reviews given in [122, 45] for example). We note that in general these methods are based on interval Taylor series expansions, but here we have a simple adaptation to validate solutions in Chebyshev basis. We use the Picard-Lindelöf iteration map, defined in the space of continuous functions $u : [-1, 1] \to \mathbb{R}$ by:

$$\tau(u)(t) = y_0 + \int_{t_0}^{t} f(u(s), s)\mathrm{d}s. \tag{5.38}$$

It is classical (see [Chap. 15] [151]) that if f satisfies a Lipschitz condition in its first variable, then τ has a unique fixed point y, solution of the associated ODE, with the initial condition $y(t_0) = y_0$.

For the sake of simplicity, we initially present our method for the case of an order 1 linear differential equation with polynomial coefficients, then we show how we can use this method to give a more general algorithm for linear differential equation of order r.

1 **Algorithm:** `ValidateOrder-1`$(L, y(t_0) = y_0, p, M)$
Input: an order 1 differential operator L (written in suitable form $L \cdot y = y' - a \cdot y$, with $a \in \mathbb{Q}(x)$),
an initial condition $y(t_0) = y_0$,
an approximation polynomial $p(x) = \sum_{n=0}^{d} \tilde{y}_n T_n(x)$ for y,
an error bound M that will be used in Algorithm 5.4.1.
Output: an error bound B s.t. $\|y - p\|_\infty \leqslant B$
2 **Procedure:**
1 $P_0 := p$;
2 Find j such that $\gamma_j(t) := \|a\|_\infty{}^j \cdot \frac{|t-t_0|^j}{j!} < 1$;
3 For $i = 1 \dots j$
4 use Algorithm 5.4.1 to compute an approximation polynomial $P_{a,i}$ for $a \cdot P_{i-1}$ such that $\|P_{a,i} - a \cdot P_{i-1}\|_\infty \leqslant M$;
5 compute $P_i(t) := y_0 + \int_{t_0}^{t} P_{a,i}(x)\mathrm{d}x$;
6 Compute $\alpha_j(t) := M \cdot \sum_{k=1}^{j} \|a\|_\infty{}^{k-1} \frac{|t-t_0|^k}{k!}$, $\beta_j(t) := P_j(t) - P_0(t)$;
7 Return $R_j^* = \frac{\|\alpha_j\|_\infty + \|\beta_j\|_\infty}{1 - \|\gamma_j\|_\infty}$.

Algorithm 5.5.1: Linear time validation of a numerical polynomial solution in Chebyshev basis for an order 1 LODE.

Order 1 LODE with polynomial coefficients

Proposition 5.5.1. *Given a linear differential equation with polynomial coefficients of order 1, a boundary condition and a polynomial approximation p of degree d of the unique solution y of the differential equation, Algorithm 5.5.1 computes an upper-bound B of $\|y - p\|_\infty$ in $O(d)$ arithmetic operations.*

We first prove the correctness of the algorithm. The idea of this algorithm is to iterate, starting with the initial approximation p, the unique fixed point of the operator

$$\tau(u)(t) = y_0 + \int_{t_0}^{t} a(s) \cdot u(s)\mathrm{d}s, \text{with } a \in \mathbb{Q}(s), \tag{5.39}$$

which is also the solution of the differential operator L described in the input of the algorithm, in order to construct a convergent sequence of valid upper-bounds for $\|y - p\|_\infty$, whose limit is the returned bound B.

Lemma 5.5.2. *Using the notations of Algorithm 5.5.1, for all upper-bounds $R > 0$, $\|y - p\|_\infty \leq R$, $S_{k,i}(R)$ defined, for all integers i, by*

$$\begin{aligned} S_{0,i}(R) &= \|\alpha_i\|_\infty + \|\beta_i\|_\infty + \|\gamma_i\|_\infty R, \\ S_{k,i}(R) &= \|\alpha_i\|_\infty + \|\beta_i\|_\infty + \|\gamma_i\|_\infty S_{k-1,i}(R),\ k \in \mathbb{N}^*, \end{aligned} \tag{5.40}$$

is also an upper-bound of $\|y-p\|_\infty$.

Proof. Let p be a polynomial and R an upper-bound for $\|y-p\|_\infty$. Using (5.39) we construct successive approximations $P_i(t)$ as described in lines $4-5$ of Algorithm 5.5.1 with the initialization $P_0 = p$. We also compute through simple integration valid bounds of $|y(t) - P_i(t)|$:

$$R_i(t) := M \sum_{k=1}^{i} \|a\|_\infty{}^{k-1} \frac{|t-t_0|^k}{k!} + \|a\|_\infty{}^{i} R \frac{|t-t_0|^i}{i!},\ t \in [-1,1],\ i \geq 1. \tag{5.41}$$

We verify this bound with an induction and the following triangular inequality:

$$\begin{aligned} |y(t) - P_{i+1}(t)| &\leq |y(t) - \tau(P_i)(t)| + |\tau(P_i)(t) - P_{i+1}(t)| \\ &\leq \left| \int_{t_0}^{t} a(s) R_i(s) ds \right| + \left| \int_{t_0}^{t} M ds \right| \\ &\leq R_{i+1}(t). \end{aligned}$$

Hence, for each $t \in [-1,1]$ and each integer i, we have:

$$|y(t) - p(t)| \leq |y(t) - P_i(t)| + |P_i(t) - p(t)|.$$

It follows that $\|y-p\|_\infty \leqslant \|\alpha_i\|_\infty + \|\beta_i\|_\infty + \|\gamma_i\|_\infty R$. With an induction over k, we deduce the lemma. □

This lemma is the main idea of our algorithm, since it allows us to obtain sequences of upper-bounds for $\|y-p\|_\infty$.

Lemma 5.5.3. *For all integer j such that $\|\gamma_j\|_\infty < 1$, we have the following limit:*

$$\lim_{k\to\infty} S_{k,j}(R) = \frac{\|\alpha_j\|_\infty + \|\beta_j\|_\infty}{1 - \|\gamma_j\|_\infty}.$$

Proof. Using Lemma 5.5.2, an induction over k and the equation 5.40, we prove the equality:

$$S_{k,j}(R) = \sum_{i=0}^{k-1} ((\|\alpha_j\|_\infty + \|\beta_j\|_\infty)\|\gamma_j\|_\infty{}^{i}) + \|\gamma_j\|_\infty{}^{k} R.$$

Knowing the convergence of a geometric series and the inequality $\|\gamma_j\|_\infty < 1$, we deduce:

$$\lim_{k\to\infty} S_{k,j}(R) = \frac{\|\alpha_j\|_\infty + \|\beta_j\|_\infty}{1 - \|\gamma_j\|_\infty}.$$

□

***Proof** of Proposition 5.5.1.* Algorithm 5.5.1 computes an integer j such that $\|\gamma_j\|_\infty < 1$, from previous lemmata, it follows that $\|y-p\|_\infty \leqslant R_j^*$ and hence the value returned by Algorithm 5.5.1 is correct.

Dependencies of this algorithm in d are given by the call of Algorithm 5.4.1. Moreover, each loop iterations takes $O(d)$ operations and the number of loop iterations does not depend on d. Hence, the complexity analysis of this algorithm is well $O(d)$ arithmetic operations. □

Quality of the validated bound.

Lemma 5.5.4. *Let* $\varepsilon^* = \|y-p\|_\infty$, $B = \frac{\|\alpha_j\|_\infty + \|\beta_j\|_\infty}{1-\|\gamma_j\|_\infty}$. *We have:*

$$\varepsilon^* \leqslant B \leqslant \frac{2\,\|\alpha_j\|_\infty + (1+\|\gamma_j\|_\infty)\varepsilon^*}{1-\|\gamma_j\|_\infty}.$$

Proof. We have $\|\beta_j\|_\infty = \|P_j - p\|_\infty \leqslant \|y-P_j\|_\infty + \|y-p\|_\infty$. Using (5.41) with $R=\varepsilon^*$ we have: $\|\beta_j\|_\infty \leqslant \|\alpha_j\|_\infty + (1+\|\gamma_j\|_\infty)\varepsilon^*$.

□

Remark 5.5.5. *Since* $\|\alpha_j\|_\infty$ *can be made as small as desired, suppose that* $\|\alpha_j\|_\infty \leqslant \mu_M\varepsilon^*$. *We can compute the uniform norm with a relative error:*

$$\left|\frac{\varepsilon^* - B}{\varepsilon^*}\right| \leqslant \frac{2(\mu_M + \|\gamma_j\|_\infty)}{1-\|\gamma_j\|_\infty}.$$

The generalization to order r LODE is pretty straightforward. We give below the algorithm and its correction.

1 **Algorithm:** `ValidateOrder-r`$(L, \{y^{(i)}(t_0) = y_0^{(i)}\}_{i=0}^{r-1}, p, M)$
Input: an order r differential operator L (written in suitable form
$L\cdot y = y^{(r)} - \left(a_{r-1}y^{(r-1)} + \cdots + a_0 y\right)$, with $a_i \in \mathbb{Q}(x)$),
and suitable initial values conditions $\{y^{(i)}(t_0) = y_0^{(i)}\}_{i=0}^{r-1}$,
an approximation polynomial $p(x) = \sum_{n=0}^{d} \tilde{y}_n T_n(x)$ for y,
an error bound M that will be used in Algorithm 5.4.1.
Output: an error bound B s.t. $\|y-p\|_\infty \leqslant B$
2 **Procedure:**
1 $P_{r-1,0} := p^{(r-1)}$; $A := \max\{\|a_i\|_\infty, i = r-1\ldots 0\}$; $Q := \left\|\sum_{i=0}^{r-1} \frac{|t-t_0|^i}{i!}\right\|_\infty$;
2 Find j such that $\gamma_j(t) := A^j \cdot Q^j \cdot \frac{|t-t_0|^j}{j!} < 1$.
3 For $i = 0\ldots j-1$
4 For $k = r-2\ldots 0$ compute $P_{k,i}(t) := y^{(k)}(t_0) + \int_{t_0}^{t} P_{k+1,i}(u)\mathrm{d}u$
5 For $k = r-1\ldots 0$ use Algorithm 5.4.1 to compute approximations polynomials $P_{a_k,i}$ for $a_k \cdot P_{k,i}$ such that $\|P_{a_k,i} - a_k\cdot P_{a_k,i}\|_\infty \leq M$
6 compute $P_{r-1,i+1}(t) := y^{(r-1)}(t_0) + \int_{t_0}^{t}\sum_{k=0}^{r-1} P_{a_k,i}(u)\mathrm{d}u$;
7 Compute $\alpha_j(t) := rM \cdot \sum_{k=1}^{j} A^{k-1}Q^{k-1}\frac{|t-t_0|^k}{k!}$, $\beta_j(t) := P_{r-1,j}(t) - P_{r-1,0}(t)$;
8 Compute $R_j^* = \frac{\|\alpha_j\|_\infty + \|\beta_j\|_\infty}{1-\|\gamma_j\|_\infty}$ (R_j^* is already a valid error bound for $\left\|y^{(r-1)} - P_{r-1,0}\right\|_\infty$);
9 Return $B = R_j^* \cdot \frac{|t-t_0|^{r-1}}{(r-1)!} + \|P_{0,0} - p\|_\infty$.

Algorithm 5.5.2: Linear time validation of a numerical polynomial solution in Chebyshev basis for an order r LODE.

Order r LODE with polynomial coefficients

Proposition 5.5.6. *Let L an order r differential operator, written in suitable form $L \cdot y = y^{(r)} - \left(a_{r-1}y^{(r-1)} + \cdots + a_0 y\right)$, with $a_i \in \mathbb{Q}(x)$, and suitable initial values conditions $y^{(i)}(t_0) = y_0^{(i)}$, y the unique solution of $L \cdot y = 0$ and $p(x) = \sum_{n=0}^{d} \tilde{y}_n T_n(x)$ an approximation polynomial of degree d for y. Algorithm 5.5.1 computes a "small" bound B such that $\|y - p\|_\infty \leqslant B$ in $O(d)$ arithmetic operations.*

Proof. Correction of the algorithm.

The generalization of Algorithm 5.5.1 is based on applying (5.38), for validating an approximation of $y^{(r-1)}$:

$$\tau(u^{r-1})(t) = y_0^{(n-1)} + \int_{t_0}^{t} \left(a_{r-1}(s)u^{(r-1)}(s) + \cdots + a_0(s)u(s)\right) \mathrm{d}s, \tag{5.42}$$

where $a_i \in \mathbb{Q}(x)$.

Since we are given a numerical approximation polynomial p for y, we can compute numerically its $r-1$ derivative $P_{r-1,0} = p^{(r-1)}$. We initially compute in lines $1-8$ a validated error bound $R_j^* \geq \left\|y^{(r-1)} - P_{r-1,0}\right\|_\infty$ (this is discussed below). From that, we can easily deduce, by $r-1$ successive validated integrations of $P_{r-1,0}$, the polynomial $P_{0,0}$ and validated error bound $\|P_{0,0} - y\|_\infty \leq R_j^* \cdot \frac{|t-t_0|^{r-1}}{(r-1)!}$.

Hence, we can bound the error $\|y - p\|_\infty \leq \|y - P_{0,0}\|_\infty + \|p - P_{0,0}\|_\infty$.

Now, let us detail the lines $1-8$. The correction proof is similar to that of Algorithm 5.5.1, noting that when the order of the equation is greater than 1, we have to work not only with a polynomial approximation of y, but also with polynomial approximations of derivatives $y', \ldots, y^{(r-1)}$. The only significant change is that one can show by computation that in this case, (5.41) becomes:

$$R_{r-1,j}(t) := rM \cdot \sum_{k=1}^{j} \left(A^{k-1}Q^{k-1} \cdot \frac{|t-t_0|^k}{k!}\right) + A^j \cdot Q^j \cdot R_{r-1} \cdot \frac{|t-t_0|^j}{j!},\ j \geq 1 \tag{5.43}$$

where A, Q, M are defined in Algorithm 5.5.2, and $R_{r-1} > 0$ is a bound for the error between $y^{(r-1)}$ and $p^{(r-1)}$.

□

5.6 Discussion and future work.

The algorithms we presented validate only initial value $LODE$. Our initial setting included also boundary values problems. From a theoretical point of view, we could extend these algorithms to handle boundary values problems in two ways:

- Since the dimension of the space of solutions of L is r, we can compute and validate r linearly independent solutions $y_0 = (p_0, B_0), \ldots, y_{r-1} = (p_{r-1}, B_{r-1})$ of initial values problems using Algorithm 5.3.1 and 5.5.2. Then, our solution is a linear combination of these r solutions $y = \sum_{i=0}^{r-1} c_i y_i$. We can rigorously find c_i by solving in interval arithmetic the linear system given by the boundary conditions $\{\lambda_i(y) = \ell_i\}_{i=1}^{r}$. The complexity of this approach would still be linear in function of the degree d of the polynomial searched.
- Another possibility would be to construct fixed point operators based directly on the boundary conditions. One issue we should consider in this case is whether we can prove that these operators are contracting. We are currently exploring these two methods.

In this setting also, we intend to extend this work to other families of orthogonal polynomials. Another exploratory idea would be to try to use this work as a basic brick in a process of solving non-linear ODE using a linearization method such as Newton's method.

Now, we give below experimental results obtained with our prototype implementation (written in Maple). Our implementation is not yet tuned and it is highly probable that timings will be improved.

5.7 Experiments

First, Algorithm 5.3.1 is used to compute a numerical solution, then Algorithm 5.5.2 computes the validated bound. For the following three examples, we plot in the respective figures the error between the exact solution and the polynomial computed using Algorithm 5.3.1 for degrees $d \in 30, 60, 90$. In Table 5.1 we give timings and validated bounds using Algorithm 5.5.1 and 5.5.2.

Example 5.7.1. *Example adapted from [84]:*

$$(-x-15)y(x) + (32+2x)y'(x) = 0, \quad y(0) = 1/\sqrt{16} \tag{5.44}$$

with the exact solution:

$$y(x) = \frac{e^{x/2}}{\sqrt{x+16}} \tag{5.45}$$

Example 5.7.2. *Fourth order initial value problem taken from [68].*

$$y^{(4)}(x) - y(x) = 0, \quad y(0) = 3/2,\ y'(0) = -1/2, y''(0) = -3/2,\ y'''(0) = 1/2, \tag{5.46}$$

with the exact solution:

$$y(x) = 3/2\cos(x) - 1/2\sin(x). \tag{5.47}$$

Example 5.7.3. *Initial value problem for a function that has complex singularities in $z = \pm\sqrt{2}/2$, note also that the initial condition is not rational.*

$$4xy(x) + (1+4x^2+4x^4)y'(x) = 0, \quad y(0) = \exp(1), \tag{5.48}$$

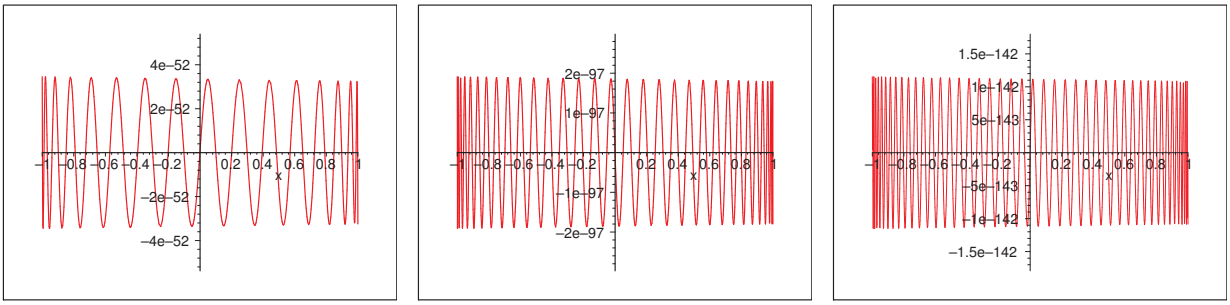
with the exact solution:

$$y(x) = \exp(1/(1+2x^2)). \tag{5.49}$$

Ex.	Deg. d	Timings (s) Algorithm 5.3.1	Validated Bound	Timings (s) Validation Algorithm
5.7.1	30	0.09	$0.20 \cdot 10^{-46}$	0.57
	60	0.09	$0.73 \cdot 10^{-96}$	1.17
	90	0.096	$0.44 \cdot 10^{-141}$	1.74
5.7.2	30	0.10	$0.78 \cdot 10^{-40}$	0.76
	60	0.116	$0.45 \cdot 10^{-98}$	1.8
	90	0.116	$0.35 \cdot 10^{-163}$	3.9
5.7.3	30	0.10	$0.15 \cdot 10^{-6}$	2.2
	60	0.104	$0.26 \cdot 10^{-14}$	13.3
	90	0.108	$0.32 \cdot 10^{-22}$	20.9

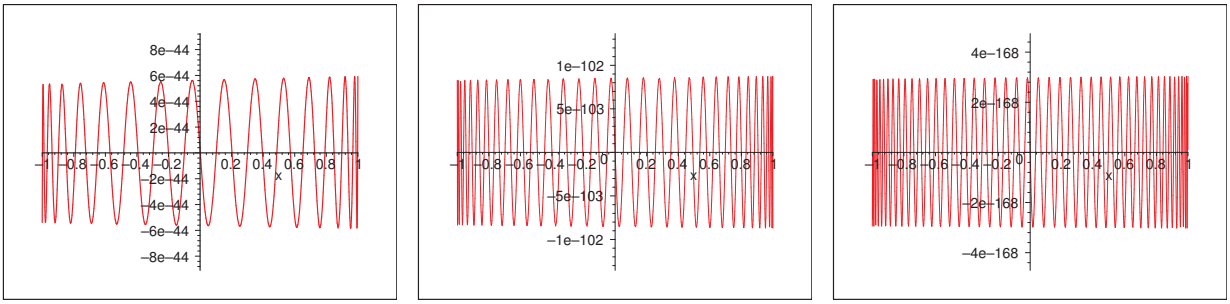
Table 5.1: Timings and validated bounds for Examples 5.7.1, 5.7.2, 5.7.3.

In general, we do not know the closed form solution of LODEs with polynomial coefficients, so we deem it interesting to provide as an example the *certified plot* of the solution using the following



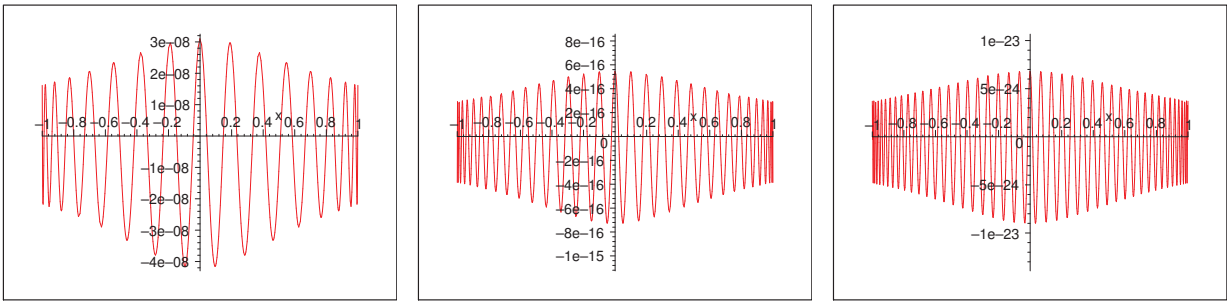
(a) $d = 30$ (b) $d = 60$ (c) $d = 90$

Figure 5.2: Approximation errors for Example 5.7.1.



(a) $d = 30$ (b) $d = 60$ (c) $d = 90$

Figure 5.3: Approximation errors for Example 5.7.2.



(a) $d = 30$ (b) $d = 60$ (c) $d = 90$

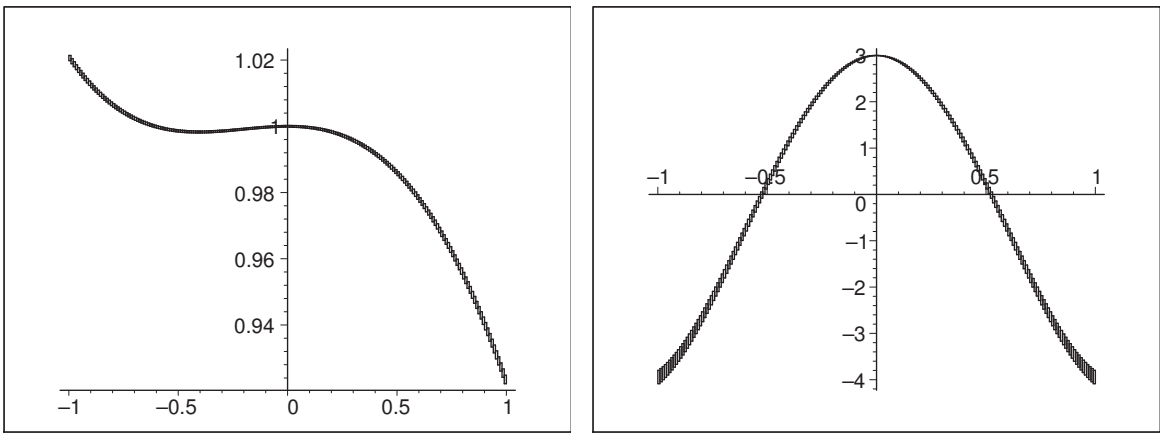
Figure 5.4: Approximation errors for Example 5.7.3.

method: first, Algorithm 5.3.1 is used to compute a numerical solution $\tilde{y}$, then Algorithm 5.5.2 computes the validated bound B. Finally, we consider a partition of the function domain $[-1,1] = \bigcup_{\substack{i=[\![0,n-1]\!],\\ x_0=-1,\, x_n=1}} [x_i, x_{i+1}]$ and evaluate using interval arithmetic $\tilde{y}([x_i, x_{i+1}]) + B \supseteq y([x_i, x_{i+1}])$, for each i. These intervals provide a piecewise enclosure of the image of y.

Example 5.7.4. *Certified plots of D-finite functions.*

$$(x^2+16)y''(x) + (3x^2 - x + 2)y'(x) + (5x+1)y(x) = 0, \quad y(0) = 1, \quad y'(0) = 0. \tag{5.50}$$

$$(60x^2 + 75 + 9x^4)y(x) + (-4x^3 - 12x)y'(x) + (x^4 + 6x^2 + 9)y''(x),\ y(0) = 3,\ y'(0) = 0. \tag{5.51}$$



(a) (5.50) (b) (5.51)

Figure 5.5: Certified plots for Example 5.7.4.

CHAPTER 6

Automatic Generation of Polynomial-based Hardware Architectures for Function Evaluation

Look engineers. All we're asking for is an infinite number of transistors on a finite-sized chip. You can't even do that? *

We present a practical application of RPAs to the synthesis of elementary functions in hardware. The main motivation of this work is to facilitate the implementation of a full hardware mathematical library (libm) in FloPoCo †, a core generator for high-performance computing on Field Programmable Gate Arrays (FPGAs).

This chapter details an architecture generator that inputs the specification of a univariate function and outputs a synthesizable description of an architecture evaluating this function with guaranteed accuracy. It improves upon the literature in two aspects. Firstly, it uses better polynomials, thanks to recent advances related to constrained-coefficient polynomial approximation. Secondly, it refines the error analysis of polynomial evaluation to reduce the size of the multipliers used.

An open-source implementation is provided in the FloPoCo project, including architecture exploration heuristics designed to use efficiently the embedded memories and multipliers of high-end FPGAs. High-performance pipelined architectures for precisions up to 64 bits can be obtained in seconds.

This is a joint work [46] with Florent de Dinechin and Bogdan Pasca.

6.1 Introduction and motivation

In this work, we deal also with univariate real functions $f : [a, b] \to \mathbb{R}$ which are assumed to be *sufficiently smooth*, i.e. continuously differentiable on the interval $[a, b]$ up to a certain order. We are interested in a hardware fixed-point implementation of this function over this interval. The literature provides many examples of such functions for which a hardware implementation is required.

- Fixed-point sine, cosine, exponential and logarithms are routinely used in signal processing algorithms.
- Random number generators with a Gaussian distribution may be built using the Box-Muller method, which requires logarithm, square root, sine and cosine [92]. Arbitrary distributions may be obtained by the inversion method, in which case one needs a fixed-point evaluator

*. Dark Discussions at Cafe Infinity, using calculators as proof, `http://www.mathisfunforum.com/`

†. `www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/`

for the inverse cumulative distribution function (ICDF) of the required distribution [34]. There are as many ICDF as there are statistical distributions.
- Approximations of the inverse $1/x$ and inverse square root $1/\sqrt{x}$ functions are used in recent floating-point units to bootstrap division and square root computation [101].
- $f_{\log}(x) = \log(x + 1/2)/(x - 1/2)$ over $[0, 1]$, and $f_{\exp}(x) = e^x - 1 - x$ over $[0, 2^{-k}]$ for some small k, are used to build hardware floating-point logarithm and exponential in [53].
- $f_{\cos}(x) = 1 - \cos\left(\frac{\pi}{4}x\right)$, and $f_{\sin}(x) = \frac{\pi}{4} - \frac{\sin\left(\frac{\pi}{4}x\right)}{x}$ over $[0, 1]$, are used to build hardware floating-point trigonometric functions in [52].
- $s_2(x) = \log_2(1 + 2^x)$ and $d_2(x) = \log_2(1 + 2^x)$ are used to build adders and subtracters in the Logarithm Number System (LNS), and many more functions are needed for Complex LNS [4].

Many function-specific algorithms exist, for example variations on the CORDIC algorithm provide low-area, long-latency evaluation of most elementary functions [117]. Our purpose here is to provide a generic method, that is a method that works for a very large class of functions. The main motivation of this work is to facilitate the implementation of a full hardware mathematical library (libm) in FloPoCo, a core generator for high-performance computing on FPGAs *. We present a complete implementation in this context, however, most of the methodology is independent of the FPGA target and could apply to other hardware targets such as ASIC circuits.

6.1.1 Related work and contributions

Articles describing specific polynomial evaluators are too numerous to be mentioned here, and we just review works that describe generic methods.

Several table-based, multiplier-less methods for linear (or degree-1) approximation have evolved from the original paper by Sunderland et al [155]. See [50] or [117] for a review. These methods have very low latency but do not scale well beyond 20 bits: the table sizes scale exponentially, and so does the design-space exploration time.

The High-Order Table-Based Method (HOTBM) by Detrey and de Dinechin [51] extended the previous methods to higher-degree polynomial approximation. An open-source implementation is available in FloPoCo. However it is not suited to recent FPGAs with powerful DSP blocks and large embedded memories. In addition, it doesn't scale beyond 32 bits.

Lee et al [93] have published many variations on a generic datapath optimization tool called MiniBit to optimize polynomial approximation. They use ad-hoc mixes of analytical techniques such as interval analysis, and heuristics such as simulated annealing to explore the design space. However, the design space explored in these articles does not include the architectures we describe in the present paper: All the multipliers in these papers are larger than strictly needed, therefore they miss the optimal. In addition, this tool is closed-source and difficult to evaluate from the publications, in particular it is unclear if it scales beyond 32 bits.

Tisserand studied the optimization of low-precision (less than 10 bits) polynomial evaluators [157]. He finetunes a rounded minimax approximation using an exhaustive exploration of neighboring polynomials. He also uses other tricks on smaller (5-bit or less) coefficients to replace the multiplication by such a coefficient by very few additions. Such tricks do not scale to larger precisions.

Compared to these publications, the present work has the following distinctive features.
- This approach scales to precisions of 64 bits or more, while being equivalent or better than the previous approaches for smaller precisions.

*. www.ens-lyon.fr/LIP/Arenaire/Ware/FloPoCo/

Family	*Multipliers*
Virtex II to Virtex-4	18x18 signed or 17x17 unsigned
Virtex-5/Virtex-6	18x25 signed or 17x24 unsigned
Stratix II/III/IV	18x18 signed or unsigned

Table 6.1: Multiplier blocks in recent FPGAs

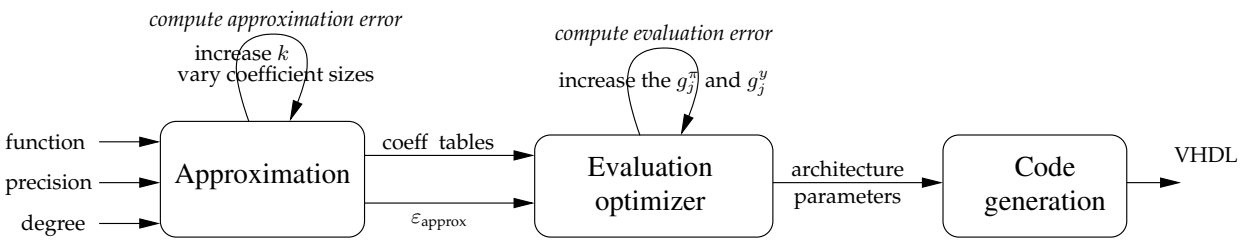


Figure 6.1: Automated implementation flow

- We use for polynomial approximation minimax polynomials provided by the Sollya tool *, which is the state-of-the-art for this application, as detailed in Section 6.2.2.
- We attempt to use the smallest possible multipliers. As others, we attempt to minimize the coefficient sizes. In addition, we also truncate, at each computation step, the input argument to the bare minimum of bits that are needed at this step. Besides, we use truncated multipliers [169, 5] in order to further reduce resource consumption.
- This approach is fully automated, from the parsing of an expression describing the function to VHDL generation. An open-source implementation is available as the FunctionEvaluator class in FloPoCo, starting with version 2.0.0. This implementation is fully operational, to the point that Table 6.2 was obtained in less one hour.
- The resulting architecture evaluates the function with last-bit accuracy. It may be automatically pipelined to a user-specified frequency thanks FloPoCo's pipelining framework [47].

6.1.2 Relevant features of recent FPGAs

Here are some of the features of recent FPGAs that can be used in polynomial evaluators.

- Embedded multipliers features are summed up in Table. 6.1 It is possible to build larger multipliers by assembling these embedded multipliers. The DSP blocks include specific adders and shifters designed for this purpose [5].
- Memories have a capacity of 9Kbit or 144Kbit (Altera) or 18Kbit (Xilinx) and can be configured in shape, for instance from $2^{16} \times 1$ to $2^9 \times 36$ for the Virtex-4.

A given FPGA typically contains a comparable number of memory blocks and multipliers. It therefore makes sense to try and balance the consumption of these two resources. However, the availability of these resources also depends on the wider context of the application, and it is even better to expose a range of trade-offs between them.

6.2 Function evaluation by polynomial approximation

We recall that for evaluating a function in hardware, it is usual to replace it by an approximation polynomial, in order to reduce this evaluation to additions and multiplications. For these

*. http://sollya.gforge.inria.fr/

operations, we can either build architectures (in FPGAs or ASICs), or use built-in operators (in processors or DSP-enabled FPGAs).

Building a polynomial-based evaluator for a function may be decomposed into two subproblems: 1/ *approximation*: finding a good approximation polynomial, and 2/ *evaluation*: evaluating it using adders and multipliers. The smaller the input argument, the better these two steps will behave, therefore a *range reduction* may be applied first if the input interval is large.

We now discuss each of these steps in more detail, to build the implementation flow depicted on Figure 6.1. In this paper we will consider, without loss of generality, a function f over the input interval $[0, 1)$.

In our implementation, the user inputs a function (assumed on $[0, 1)$, the input and output precisions, both expressed as least significant bit (LSB) weight, and the degree d of the polynomials used. This last parameter could be determined heuristically, but we leave it as a means for the user to trade-off multipliers and latency for memory size.

6.2.1 Range reduction

In this work, we use the simple range reduction that consists in splitting the input interval in 2^k sub-intervals, indexed by $i \in \{0, 1, ..., 2^k - 1\}$. The index i may be obtained as the leading bits of the binary representation of the input: $x = 2^{-k}i + y$ with $y \in [0, 2^{-k})$. This decomposition comes at no hardware cost. We now have $\forall i \in \{0, \ldots, 2^k - 1\} \quad f(x) = f_i(y)$, and we may approximate each f_i by a polynomial p_i. A table will hold the coefficients of all these polynomials, and the evaluation of each polynomial will share the same hardware (adders and multipliers), which therefore have to be built to accommodate the worst-case among these polynomials. Figure 6.3 describes the resulting architecture.

Compared to a single polynomial on the interval, this range reduction increases the storage space required, but decreases the cost of the evaluation hardware for two reasons. First, for a given target accuracy $\varepsilon_{\text{total}}$, the degree of each of the p_i decreases with k. There is a strong threshold effect here, and for a given degree there is a minimal k that allows to achieve the accuracy. Second, the reduced argument y has k bits less than the input argument x, which will reduce the input size of the corresponding multipliers. If we target an FPGA with DSP blocks, there will also be a threshold effect here on the number of DSP blocks used.

Many other range reductions are possible, most related to a given function or class of functions, like the logarithmic segmentation used in [34]. For an overview, see Muller [117]. Most of our contributions are independent of the range reduction used.

6.2.2 Polynomial approximation

We recall that in this case the coefficients of the approximation polynomial have to be *machine efficient*. We already saw that coefficients of well-known approximation polynomials like Taylor, Chebyshev or minimax can be obtained with very high precision in practice, using various numerical methods. So naively, the best polynomial to use would be the one obtained by rounding each coefficient of the minimax polynomial to smaller-precision numbers suitable for efficient evaluation to the machine representable format. On a processor, one will typically try to round to single- or double-precision numbers. On an FPGA, we may build adders and multipliers of arbitrary size, so we have one more question to answer: what is the optimal size of these coefficients? In [93], this question is answered by an error analysis that considers separately the error of rounding each coefficient of the minimax polynomial (considered as a real-coefficient one) and tries to minimize the bit-width of the rounded coefficients while remaining within acceptable error bounds.

However, there is no guarantee that the polynomial obtained by rounding the coefficients of the real minimax polynomial is the minimax among the polynomials with coefficients constrained

to these bit-width. Indeed, this assumption is generally wrong. One may obtain much more accurate polynomials for the same coefficient bit-width using a combination of an extension of Remez algorithm and lattice basis reduction algorithms, due to Brisebarre and Chevillard [26] and implemented as the `fpminimax` command of the Sollya tool. This command inputs a function, an interval and a list of constraints on the coefficient (e.g. constraints on bitwidths), and returns a polynomial that is very close to the best minimax approximation polynomial among those with such constrained coefficients.

Since the approximation polynomial now has constrained coefficients, we will not round these coefficients anymore. In other words, we have merged the approximation error and the coefficient truncation error of [93] into a single error, which we still denote $\varepsilon_{\text{approx}}$. We have seen in Chapter 3 how we can automatically validate an upper-bound for $\varepsilon_{\text{approx}}$. In fact, one can observe that we combined the the work of Brisebarre and Chevillard [26] for finding numerically *machine-efficient* polynomial approximations with the process of validated computation of supremum norms of approximation errors in order obtain a Rigorous Polynomial Approximation with machine-efficient coefficients. This provides an answer for the generic Problem 1 in the case of polynomials with constrained machine coefficients.

In this way, the only remaining rounding or truncation errors to consider are those that happen during the evaluation of the polynomial. Before dealing with that, let us first provide a good heuristic for determining the coefficient constraints.

Let $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$ be the polynomial on one of the sub-intervals (for clarity, we remove the indices corresponding to the sub-interval). The constraints taken by `fpminimax` are the minimal weights of the least significant bit (LSB) of each coefficient. To reach some target precision 2^{-p}, we need the LSB of a_0 to be of weight at most 2^{-p}. This provides the constraint on a_0. Now consider the developed form of the polynomial, as illustrated by Figure 6.2. As coefficient a_j is multiplied by y^j which is smaller than 2^{-kj}, the accuracy of the monomial $a_j y^j$ will be aligned on that of the monomial a_0 if its LSB is of weight 2^{-p+kj}. This provides a constraint on a_j.

The heuristic used is therefore the following. Remember that the degree d is provided by the user. The constraints on the $d+1$ coefficients are set as just explained. For increasing k, we try to find 2^k approximation polynomials p_i of degree d respecting the constraints, and fulfilling the target approximation error (which will be defined in Section 6.2.4). We stop at the first k that succeeds. Then, the 2^k polynomials are scanned, and the maximum magnitude of all the coefficients of degree j provides the most significant bit that must be tabulated, hence the memory consumed by this coefficient.

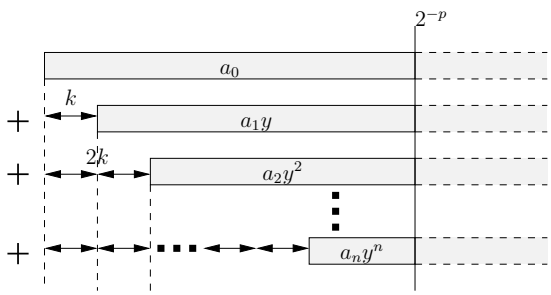


Figure 6.2: Alignment of the monomials

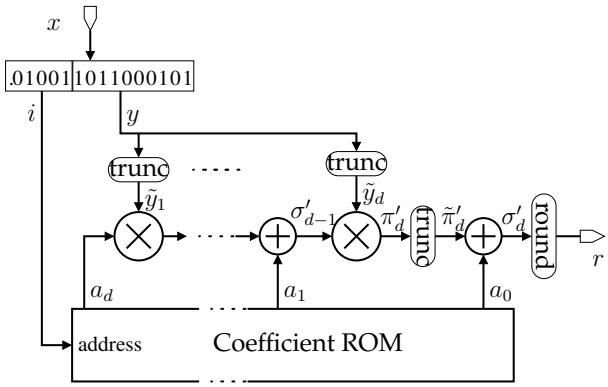


Figure 6.3: The function evaluation architecture

6.2.3 Polynomial evaluation

Given a polynomial, there are many possible ways to evaluate it. The HOTBM method [51] uses the developed form $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$ and attempts to tabulate as much of the computation as possible. This leads to short-latency architecture since each of the $a_i y^i$ may be evaluated in parallel and added thanks to an adder tree, but at a high hardware cost.

In this work, we chose a more classical Horner evaluation scheme, which minimizes the number of operations, at the expense of the latency: $p(y) = a_0 + y \times (a_1 + y \times (a_2 + ... + y \times a_d)...)$. Our contribution is essentially a fine error analysis that allows us to minimize the size of each of the operations. It is presented below in 6.2.4.

There are intermediate schemes that could be explored. For large degrees, the polynomial may be decomposed into an odd and an even part: $p(y) = p_e(y^2) + y \times p_o(y^2)$. The two sub-polynomial may be evaluated in parallel, so this scheme has a shorter latency than Horner, at the expense of the precomputation of x^2 and a slightly degraded accuracy. Many variations on this idea, e.g. the Estrin scheme, exist [117], and this should be the subject of future work. A polynomial may also be refactored to trade multiplications for more additions [87], but this idea is mostly incompatible with range reduction.

6.2.4 Accuracy and error analysis

The maximal error target $\varepsilon_{\text{total}}$ is an input to the algorithm. Typically, we aim at *faithful rounding*, which means that $\varepsilon_{\text{total}}$ must be smaller than the weight of the LSB of the result, noted u. In other words, all the bits returned hold useful information. This error is decomposed as follows: $\varepsilon_{\text{total}} = \varepsilon_{\text{approx}} + \varepsilon_{\text{eval}} + \varepsilon_{\text{finalround}}$ where

- $\varepsilon_{\text{approx}}$ is the approximation error, the maximum absolute difference between any of the p_i and the corresponding f_i over their respective intervals. This computation belongs to the approximation step and is also performed in Sollya, as explained in Chapter 3.
- $\varepsilon_{\text{eval}}$ is the total of all rounding errors during the evaluation;
- $\varepsilon_{\text{finalround}}$ is the error corresponding to the final rounding of the evaluated polynomial to the target format. It is bounded by $u/2$.

We therefore need to ensure $\varepsilon_{\text{approx}} + \varepsilon_{\text{eval}} < u/2$. The polynomial approximation algorithm iterates until $\varepsilon_{\text{approx}} < u/4$, then reports $\varepsilon_{\text{approx}}$. The error budget that remains for the evaluation is therefore $\varepsilon_{\text{eval}} < u/2 - \varepsilon_{\text{approx}}$ and is between $u/4$ and $u/2$.

In $p(y) = a_0 + a_1 y + a_2 y^2 + ... + a_d y^d$, the input y is considered exact, so $p(y)$ is the value of the polynomial if evaluated in infinite precision. What the architecture evaluates is $p'(y)$, and our

purpose here is to compute a bound on $\varepsilon_{\text{eval}}(y) = p'(y) - p(y)$.

Let us decompose the Horner evaluation of p as a recurrence:

$$\begin{cases} \sigma_0 = a_d \\ \pi_j = y \times \sigma_{j-1} & \forall j \in \{1...d\} \\ \sigma_j = a_{d-j} + \pi_j & \forall j \in \{1...d\} \\ p(y) = \sigma_d \end{cases}$$

This would compute the exact value of the polynomial, but at each evaluation step, we may perform two truncations, one on y, and one on π_j. As a rule of thumb, each step should balance the effect of these two truncations on the final error. For instance, in an addition, if one of the addends is much more accurate than the other one, it probably means that it was computed too accurately, wasting resources.

To understand what is going on, consider step j. In the addition $\sigma_j = a_{d-j} + \pi_j$, the π_j should be at least as accurate as a_{d-j}, but not much more accurate: let us keep g_j^π bits to the right of the LSB of a_{d-j}, where g_j^π is a small positive integer ($0 \leq g_j^\pi < 5$ in our experiments). The parameter g_j^π defines the truncation of π_j, and also the size of σ_j (which also depends on the weight of the MSB of a_{d-j}).

Now since we are going to truncate $\pi_j = y \times \sigma_{j-1}$, there is no need to input to this computation a fully accurate y. Instead, y should be truncated to the size of the truncated π_j, plus a small number g_j^y of guard bits.

The computation actually performed is therefore the following:

$$\begin{cases} \sigma'_0 = a_d, \\ \pi'_j = \tilde{y}_j \times \sigma'_{j-1}, & \forall j \in \{1...d\}, \\ \sigma'_j = a_{d-j} + \tilde{\pi}'_j, & \forall j \in \{1...d\}, \\ p'(y) = \sigma'_d \end{cases}$$

In both previous equations, the additions and multiplications should be viewed as exact: the truncations are explicited by the tilded variables, e.g. $\tilde{\pi}'_j$ is the truncation of π'_j to g_j^π bits beyond the LSB of a_{d-j}. There is no need to truncate the result of the addition, as the truncation of π'_j serves this purpose already.

We may now compute the rounding error:

$$\varepsilon_{\text{eval}} = p'(y) - p(y) = \sigma'_d - \sigma_d$$

where

$$\begin{aligned} \sigma'_j - \sigma_j &= \tilde{\pi}'_j - \pi_j \\ &= (\tilde{\pi}'_j - \pi'_j) + (\pi'_j - \pi_j) \end{aligned}$$

Here we have a sum of two errors. The first, $\tilde{\pi}'_j - \pi'_j$, is the truncation error on π' and is bounded by a power of two depending on the parameter g_j^π. The second is computed as

$$\begin{aligned} \pi'_j - \pi_j &= \tilde{y}_j \times \sigma'_{j-1} - y \times \sigma_{j-1} \\ &= (\tilde{y}_j \sigma'_{j-1} - y\sigma'_{j-1}) + (y\sigma'_{j-1} - y\sigma_{j-1}) \\ &= (\tilde{y}_j - y)\sigma'_{j-1} \quad + \quad y \times (\sigma'_{j-1} - \sigma_{j-1}) \end{aligned}$$

Again, we have two error terms which we may bound separately. The first bound is the truncation error on y, which depends on the parameter g_j^y, and is multiplied by a bound on σ'_{j-1} which has to be computed recursively itself. The second term recursively uses the computation of $\sigma'_j - \sigma_j$, and the bound $y < 2^{-k}$.

The previous error computation is implemented in C++. From the values of the parameters g_j^π and g_j^y, it decides if the architecture defined by these parameters is accurate enough.

Table 6.2: Examples of polynomial approximations obtained for several functions. S represents the scaling factor so that the function image is in [0,1]

$f(x)$	S	23 bits (single prec.)			36 bits			52 bits (double prec.)		
		d	k	Coeffs size	d	k	Coeffs size	d	k	Coeffs size
$\sqrt{1+x}$	$\frac{1}{2}$	2	64	$26, 20, 14$	3	128	$39, 32, 25, 18$	4	512	$55, 46, 37, 28, 19$
		1	2048	$26, 15$	2	2048	$39, 28, 17$	3	2048	$55, 44, 33, 22$
$\frac{\pi}{4} - \frac{\sin(\frac{\pi}{4}x)}{x}$	2^3	2	128	$26, 19, 12$	3	128	$39, 32, 25, 18$	4	256	$55, 47, 39, 31, 23$
		1	4096	$26, 14$	2	2048	$39, 28, 17$	3	2048	$55, 44, 33, 22$
$1 - \cos(\frac{\pi}{4}x)$	2	2	128	$26, 19, 12$	3	256	$39, 31, 23, 15$	4	256	$55, 47, 39, 31, 23$
		1	4096	$26, 14$	2	2048	$39, 28, 17$	3	4096	$55, 43, 31, 19$
$\log_2(1+x)$	1	2	128	$26, 19, 12$	3	256	$39, 31, 23, 15$	4	256	$55, 45, 35, 25, 15$
		1	4096	$26, 14$	2	4096	$39, 27, 15$	3	4096	$55, 43, 31, 19$
$\frac{\log(x+1/2)}{x-1/2}$	$\frac{1}{2}$	2	256	$26, 18, 10$	3	512	$39, 30, 21, 12$	4	1024	$55, 45, 35, 25, 15$
		1	4096	$26, 14$	2	4096	$39, 27, 15$	3	8192	$55, 42, 29, 16$

6.2.5 Parameter space exploration for the FPGA target

The last problem to solve is to find values of these parameters that minimize the cost of an implementation. This optimization problem is very dependent on the target technology, and we now present an exploration heuristic that is specific to DSP-enabled FPGAs: our objective will be to minimize the number of DSP blocks.

Let us first consider the g_j^y parameter. The size of this truncation directly influences the DSP count. Here, we observe that once a DSP block is used, it saves us almost nothing to under-use it. We therefore favor truncations which reduce the size of y to the smallest multiple of a multiplier input size that allows us to reach the target accuracy. For Virtex4 and StratixII, the size of y should target a multiple of 17 and 18 respectively. On Virtex5 and Virtex6, multiples of 17 or 24 should be investigated. Consequently, each g_j^y can take a maximum of three possible values: 0, corresponding to no truncation, and one or two soft spots corresponding to multiples of multiplier input size.

The determination of the possible values of g_j^π also depends on the DSP multiplier size, as the truncation of π'_j defines the size of the sum σ'_j, which is input to a multiplier. There are two considerations to be made: First, it makes no sense to keep guard bits to the right of the LSB of $\tilde{\pi}'_j$. This gives us an upper bound on g_j^π. Secondly, as we are trying to reduce DSP count, we should not allow a number of guard bits that increases the size of σ'_j over a multiple of the multiplier input size. This gives us a second upper bound on g_j^π. The real upper-bound in computed as a minimum of the two precomputed upper-bounds.

These upper bounds define the parameter space to explore. We also observe that the size of the multiplications increases with j in our Horner evaluation scheme. We therefore favor truncations in the last Horner steps, as these truncations can save more DSP blocks. This defines the order of exploration of the parameter space. The parameters g_j^π and g_j^y are explored using the above rules until the error $\varepsilon_{\text{eval}}$ satisfies the bound $\varepsilon_{\text{eval}} < u/2 - \varepsilon_{\text{approx}}$.

This is a fairly small parameter space exploration, and its execution time is negligible with respect to the few seconds it may take to compute all the constrained minimax approximations.

Table 6.2 presents the input and output parameters for obtaining the approximation polynomials for several representative functions mentioned in the introduction. The functions f are all considered over $[0, 1]$, with identical input and output precision. Three precisions are given in Table 6.2. Table 6.3 provides synthesis results for the same experiments.

Table 6.3: Synthesis Results using ISE 11.1 on VirtexIV xc4vfx100-12. l is the latency of the operator in cycles. All the operators operate at a frequency close to 320 MHz. The grayed rows represent results without coefficient table BRAM compaction and the use of truncated multipliers

$f(x)$	23 bits (single prec.)					36 bits					52 bits (double prec.)				
	d	l	slices	DSP	BRAM	d.	l	slices	DSP	BRAM	d	l	slices	DSP	BRAM
$\sqrt{1+x}$	2	8	92	2	1	3	17	672	3	2	4	31	1313	11	6
	1	4	37	1	3	2	11	373	3	5	3	23	819	9	18
$\frac{\pi}{4} - \frac{\sin(\frac{\pi}{4}x)}{x}$	2	9	120	2	1	3	19	1039	4	2	4	34	1172	14	3
	1	4	36	1	11	2	13	412	3	11	3	25	1029	10	19
$1 - \cos(\frac{\pi}{4}x)$	2	9	120	2	1	3	19	1039	4	2	4	34	1773	14	3
	1	4	36	1	11	2	13	412	3	11	3	22	790	9	40
$\log_2(1+x)$	2	9	120	2	1	3	21	1066	4	2	4	33	1569	14	6
	1	4	36	1	11	2	11	320	3	22	3	24	933	9	40
$\frac{\log(x+1/2)}{x-1/2}$	2	8	103	2	1	3	17	779	4	4	4	32	1584	12	11
	1	4	36	1	11	2	11	314	3	22	3	23	999	8	78

6.3 Examples and comparisons

Table 6.2 presents the input and output parameters for obtaining the approximation polynomials for several representative functions mentioned in the introduction. The functions f are all considered over $[0, 1]$, with identical input and output precision. Three precisions are given in Table 1. Table 2 provides synthesis results for the same experiments.

It is difficult to compare to previous works, especially as none of them scales to the large precisions we do. Our approach brings no savings in terms of DSP blocks for precisions below 17 bits.

We may compare to the logarithm unit [92] which computes $\log(1+x)$ on 27 bits using a degree-2 approximation. Our tool instantly finds the similar coefficient sizes 30, 22 and 12 (13 in [92]). However, our implementation uses 2 DSP blocks where [92] uses 6: one multiplier is saved thanks to the truncation of y and others thanks to truncated multipliers. For larger precisions, the savings would also be larger.

We should compare the polynomial approach to the CORDIC family of algorithms which can be used for many elementary functions [117]. Table 6.4 compares implementations for 32-bit sine and cosine, using for CORDIC the implementation from Xilinx LogiCore *. This table illustrates that these two approaches address different ends of the implementation spectrum. The polynomial approach provides smaller latency, higher frequency and low logic consumption (hence predictability in performance independently of routing pressure). The CORDIC approach consumes no DSP nor memory block. Variations on CORDIC using higher radices could improve frequency and reduce latency, but at the expense of an even higher logic cost. A deeper comparison remains to be done.

6.4 Conclusion

Application-specific systems sometimes need application-specific operators, and this includes operators for function evaluation. This work has presented a fully automatic design tool that allows one to quickly obtain architectures for the evaluation of a polynomial approximation with a

*. LogiCORE IP CORDIC v4.0, 2011, http://www.xilinx.com/support/documentation/ip_documentation/cordic_ds249.pdf

LogiCore CORDIC 4.0 sin+cos 32 cyles@296MHz, 3812 LUT, 3812 FF
This work, sin alone 16 cycles@353MHz, 2 BlockRam, 3 DSP48E, 575 FF, 770 LUT
This work, cos alone 16 cycles@390MHz, 2 BlockRam, 3 DSP48E, 609 FF, 832 LUT

Table 6.4: Comparison with CORDIC for 32-bit sine/cosine functions on Virtex5

uniform range reduction for large precisions, up to 64 bits. The resulting architectures are better optimized than what the literature offers, firstly thanks to state-of-the-art polynomial approximation tools, and secondly thanks to a finer error analysis that allows for truncating the reduced argument. They may be fully pipelined to a frequency close to the nominal frequency of current FPGAs.

This work will enable the design, in the near future, of elementary function libraries for reconfigurable computing that scale to double precision. However, we also wish to offer to the designer a tool that goes beyond a library: a generator that produces carefully optimized hardware for his very function. Such application-specific hardware may be more efficient than the composition of library components.

Towards this goal, this work can be extended in several directions.

- There is one simple way to further reduce the multiplier cost, by the careful use of truncated multipliers [169, 5]. Technically, this only changes the bound on the multiplier truncation error in the error analysis of 6.2.4. This improvement should be implemented soon.
- Another way, for large multiplications, is the use of the Karatsuba technique, which is also implemented in FloPoCo [49]. It is even compatible with the previous one.
- Non-uniform range reduction schemes should be explored. The power-of-two segmentation of the input interval used in [34] has a fairly simple hardware implementation using a leading zero or one counter. This will enable more efficient implementation of some functions.
- More parallel versions of the Horner scheme should be explored to reduce the latency.
- Parameter space exploration is tuned for minimizing DSP usage, it should also be tuned to make the best possible usage of available configurations of embedded memory blocks.
- Our tools could attempt to detect if the function is odd or even [91], and consider only odd or even polynomials for such case [117, 91]. Whether this works along with range reduction remains to be explored.
- We currently only consider a constant target error corresponding to faithful rounding, but a target error function could also be input.
- Designing a pleasant and universal interface for such a tool is a surprisingly difficult task. Currently, we require the user to input a function on $[0, 1)$, and the input and output LSB weight. Most functions can be trivially scaled to fit in this framework, but many other specific situations exist.

Bibliography

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions*. Dover, 1965.

[2] L. V. Ahlfors. *Complex analysis. An introduction to the theory of analytic functions of one complex variable.* McGraw-Hill New York, 3rd edition, 1979.

[3] American National Standards Institute and Institute of Electrical and Electronic Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754–1985, 1985.

[4] M.G. Arnold and S. Collange. A real/complex logarithmic number system ALU. *IEEE Transactions on Computers*, 60(2):202 –213, feb. 2011.

[5] S. Banescu, F. de Dinechin, B. Pasca, and R. Tudoran. Multipliers for floating-point double precision and beyond on FPGAs. *SIGARCH Comput. Archit. News*, 38:73–79, January 2011.

[6] C. Bendsten and O. Stauning. TADIFF, a Flexible C++ Package for Automatic Differentiation Using Taylor Series. Technical Report IMM-REP-1997-07, Technical University of Denmark, April 1997.

[7] A. Benoit, M. Joldes, and M. Mezzarobba. Rigorous uniform approximation of D-finite functions. 2011. In preparation.

[8] A. Benoit and B. Salvy. Chebyshev expansions for solutions of linear differential equations. In John May, editor, *ISSAC '09: Proceedings of the twenty-second international symposium on Symbolic and algebraic computation*, pages 23–30, 2009.

[9] S. Bernstein. *Leçons sur les propriétés extrémales et la meilleure approximation des fonctions analytiques d'une variable réelle professées à la Sorbonne*. Gauthier-Villars, Paris, 1926.

[10] J.-P. Berrut and L. N. Trefethen. Barycentric Lagrange interpolation. *SIAM Rev.*, 46(3):501–517, 2004.

[11] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

[12] M. Berz and K. Makino. COSY INFINITY Version 9.0. `http://cosyinfinity.org`.

[13] M. Berz and K. Makino. New methods for high-dimensional verified quadrature. *Reliable Computing*, 5(1):13–22, 1999.

[14] M. Berz and K. Makino. Rigorous global search using Taylor models. In *SNC '09: Proceedings of the 2009 conference on Symbolic numeric computation*, pages 11–20, New York, NY, USA, 2009. ACM.

[15] M. Berz, K. Makino, and Y-K. Kim. Long-term stability of the tevatron by verified global optimization. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 558(1):1 – 10, 2006. Proceedings of the 8th International Computational Accelerator Physics Conference - ICAP 2004.

[16] W. G. Bickley, L. J. Comrie, J. C. P. Miller, D. H. Sadler, and A. J. Thompson. *Bessel functions. Part II. Functions of positive integer order*. British Association for the Advancement of Science, Mathematical Tables, vol. X. University Press, Cambridge, 1952.

[17] M. Blum. Program result checking: A new approach to making programs more reliable. In Andrzej Lingas, Rolf Karlsson, and Svante Carlsson, editors, *Automata, Languages and Programming, 20th International Colloquium, ICALP93, Proceedings*, volume 700 of *Lecture Notes in Computer Science*, pages 1–14, Lund, Sweden, 1993. Springer-Verlag.

[18] S. Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, ENS Lyon, 2004. Available on the Web from http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2004/PhD2004-05.pdf.

[19] P. Borwein and T. Erdélyi. *Polynomials and Polynomial Inequalities*. Graduate Texts in Mathematics, Vol. 161. Springer-Verlag, New York, NY, 1995.

[20] A. Bostan, B. Salvy, and E. Schost. Power series composition and change of basis. In David Jeffrey, editor, *ISSAC'08*, pages 269–276. ACM, 2008.

[21] A. Bostan, B. Salvy, and E. Schost. Fast conversion algorithms for orthogonal polynomials. *Linear Algebra and its Applications*, 432(1):249–258, January 2010.

[22] R. J. Boulton. Efficiency in a fully-expansive theorem prover. Technical Report 337, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1993. Author's PhD thesis.

[23] J. P. Boyd. *Chebyshev and Fourier spectral methods*. Dover Publications Inc., Mineola, NY, second edition, 2001.

[24] R. P. Brent and H. T. Kung. $\mathcal{O}\left((n \log n)^{3/2}\right)$ algorithms for composition and reversion of power series. In J. F. Traub, editor, *Analytic Computational Complexity*, pages 217–225, New York, 1975. Academic Press.

[25] R. P. Brent and H. T. Kung. Fast Algorithms for Manipulating Formal Power Series. *Journal of the ACM*, 25(4):581–595, 1978.

[26] N. Brisebarre and S. Chevillard. Efficient polynomial L^∞ approximations. In *ARITH '07: Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 169–176, Washington, DC, 2007. IEEE Computer Society.

[27] N. Brisebarre and M. Joldes. Chebyshev interpolation polynomial-based tools for rigorous computing. In Wolfram Koepf, editor, *ISSAC*, pages 147–154. ACM, 2010.

[28] N. Brisebarre, J.-M. Muller, and A. Tisserand. Computing machine-efficient polynomial approximations. *ACM Transactions on Mathematical Software*, 32(2):236–256, June 2006.

[29] M. Bronstein. *Symbolic integration. I*, volume 1 of *Algorithms and Computation in Mathematics*. Springer-Verlag, Berlin, second edition, 2005. Transcendental functions, With a foreword by B. F. Caviness.

[30] M. Bronstein and B. Salvy. Full partial fraction decomposition of rational functions. In M. Bronstein, editor, *ISSAC'93*, pages 157–160. ACM, 1993.

[31] L. Brutman. Lebesgue functions for polynomial interpolation—a survey. *Ann. Numer. Math.*, 4(1-4):111–127, 1997. The heritage of P. L. Chebyshev: a Festschrift in honor of the 70th birthday of T. J. Rivlin.

[32] C.-Y. Chen. Computing interval enclosures for definite integrals by application of triple adaptive strategies. *Computing*, pages 81–99, 2006.

[33] E. W. Cheney. *Introduction to Approximation Theory*. McGraw-Hill, 1966.

[34] R.C.C. Cheung, Dong-U Lee, W. Luk, and J.D. Villasenor. Hardware generation of arbitrary random number distributions from uniform distributions via the inversion method. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 15(8):952 –962, aug. 2007.

[35] S. Chevillard. *Évaluation efficace de fonctions numériques. Outils et exemples*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2009.

[36] S. Chevillard, J. Harrison, M. Joldeş, and Ch. Lauter. Efficient and accurate computation of upper bounds of approximation errors. *Theoret. Comput. Sci.*, 412(16):1523–1543, 2011.

[37] S. Chevillard, M. Joldeş, and C. Lauter. Sollya: An environment for the development of numerical codes. In K. Fukuda, J. van der Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 28–31, Heidelberg, Germany, September 2010. Springer.

[38] S. Chevillard, M. Joldes, and C. Lauter. Certified and fast computation of supremum norms of approximation errors. In *19th IEEE SYMPOSIUM on Computer Arithmetic*, pages 169–176, 2009.

[39] S. Chevillard and Ch. Lauter. A certified infinite norm for the implementation of elementary functions. In *Proc. of the 7th International Conference on Quality Software*, pages 153–160, 2007.

[40] S. Chevillard, Ch. Lauter, and M. Joldes. Users' manual for the Sollya tool, Release 2.0. `https://gforge.inria.fr/frs/download.php/26860/sollya.pdf`, April 2010.

[41] F. Cháves and M. Daumas. A library of taylor models for pvs automatic proof checker. In *in Proceedings of the NSF workshop on reliable engineering computing*, pages 39–52, 2006.

[42] C. W. Clenshaw. The numerical solution of linear differential equations in Chebyshev series. *Proceedings of the Cambridge Philosophical Society*, 53(1):134–149, 1957.

[43] Coq development team. *The Coq Proof Assistant Reference Manual*, 2010. `http://coq.inria.fr`.

[44] G. F. Corliss. Survey of interval algorithms for ordinary differential equations. *Applied Mathematics and Computation*, 31:112 – 120, 1989. Special Issue Numerical Ordinary Diferrential Equations (Proceedings of the 1986 ODE Conference).

[45] G. F. Corliss. Guaranteed error bounds for ordinary differential equations. In *In Theory of Numerics in Ordinary and Partial Differential Equations*, pages 1–75. Oxford University Press, 1994.

[46] F. de Dinechin, M. Joldes, and B. Pasca. Automatic generation of polynomial-based hardware architectures for function evaluation. In *Application-specific Systems Architectures and Processors (ASAP), 2010 21st IEEE International Conference on*, pages 216 –222, july 2010.

[47] F. de Dinechin, C. Klein, and B. Pasca. Generating high-performance custom floating-point pipelines. In *International Conference on Field Programmable Logic and Applications*, Prague Czech Republic, 08 2009. IEEE. RR LIP 2009-16.

[48] F. de Dinechin, Ch. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions using Gappa. In P. Langlois and S. Rump, editors, *Proceedings of the 21st Annual ACM Symposium on Applied Computing - MCMS Track*, volume 2, pages 1318–1322, Dijon, France, April 2006. Association for Computing Machinery, Inc. (ACM).

[49] F. de Dinechin and B. Pasca. Large multipliers with fewer DSP blocks. In *International Conference on Field Programmable Logic and Applications*. IEEE, aug 2009.

[50] F. de Dinechin and A. Tisserand. Multipartite table methods. *IEEE Transactions on Computers*, 54(3):319–330, 2005.

[51] J. Detrey and F. de Dinechin. Table-based polynomials for fast hardware function evaluation. In *Application-Specific Systems, Architectures and Processors*, pages 328–333. IEEE, 2005.

[52] J. Detrey and F. de Dinechin. Floating-point trigonometric functions for FPGAs. In *Intl Conference on Field-Programmable Logic and Applications*, pages 29–34. IEEE, August 2007.

[53] J. Detrey and F. de Dinechin. Parameterized floating-point logarithm and exponential functions for FPGAs. *Microprocessors and Microsystems, Special Issue on FPGA-based Reconfigurable Computing*, 31(8):537–545, 2007.

[54] T.A. Driscoll, F. Bornemann, and L.N. Trefethen. The chebop system for automatic solution of differential equations. *BIT Numerical Mathematics*, 48(4):701–723, 2008.

[55] I. Eble and M. Neher. ACETAF: A Software Package for Computing Validated Bounds for Taylor Coefficients of Analytic Functions. *ACM Transactions on Mathematical Software*, 29(3):263–286, 2003.

[56] H. Ehlich and K. Zeller. Schwankung von Polynomen zwischen Gitterpunkten. *Mathematische Zeitschrift*, 86(1):41–44, February 1964.

[57] T. H. Einwohner and R. J. Fateman. A macsyma package for the generation and manipulation of chebyshev series. In *Proceedings of the ACM-SIGSAM 1989 international symposium on Symbolic and algebraic computation*, ISSAC '89, pages 180–185, New York, NY, USA, 1989. ACM.

[58] M. K. El-Daou, E. L. Ortiz, and H. Samara. A unified approach to the tau method and Chebyshev series expansion techniques. *Comput. Math. Appl.*, 25(3):73–82, 1993.

[59] D. Elliott, D. F. Paget, G. M. Phillips, and P. J. Taylor. Error of truncated Chebyshev series and other near minimax polynomial approximations. *J. Approx. Theory*, 50(1):49–57, 1987.

[60] C. Epstein, W.L. Miranker, and T.J. Rivlin. Ultra-arithmetic i: function data types. *Mathematics and Computers in Simulation*, 24(1):1–18, 1982.

[61] C. Epstein, W.L. Miranker, and T.J. Rivlin. Ultra-arithmetic ii: intervals of polynomials. *Mathematics and Computers in Simulation*, 24(1):19–29, 1982.

[62] L. Fousse. *Intégration numérique avec erreur bornée en précision arbitraire*. PhD in Computer Science, Université Henri Poincaré – Nancy 1, 2006.

[63] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33, June 2007.

[64] L. Fox. Chebyshev methods for ordinary differential equations. *The Computer Journal*, 4(4):318, 1962.

[65] L. Fox and I.B. Parker. *Chebyshev polynomials in numerical analysis*. Oxford University Press, 1968.

[66] W. Gautschi. Computational aspects of three-term recurrence relations. *SIAM Review*, 9(1):pp. 24–82, 1967.

[67] W. Gautschi. Questions of numerical condition related to polynomials. *Studies in Numerical Analysis, MAA Stud. Math., Math. Assoc. America*, (24):140–177, 1984.

[68] K.O. Geddes. Symbolic computation of recurrence equations for the Chebyshev series solution of linear ODE's. In *Proceedings of the 1977 MACSYMA User's Conference*, pages 405–423, jul 1977.

[69] A. O. Gel′fond. *Calculus of finite differences*. Hindustan Pub. Corp., Delhi, 1971. Translated from the Russian, International Monographs on Advanced Mathematics and Physics.

[70] P. Giorgi. On polynomial multiplication in Chebyshev basis. *To appear in IEEE Transactions on Computers*, 2010.

[71] X. Gourdon. *Combinatoire, Algorithmique et Géometrie des Polynômes*. PhD thesis, École Polytechnique, Paris, France, 1996.

[72] X. Gourdon and B. Salvy. Effective asymptotics of linear recurrences with rational coefficients. *Discrete Mathematics*, 153(1-3):145–163, 1996.

[73] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.

[74] M. Grimmer, K. Petras, and N. Revol. Multiple Precision Interval Packages: Comparing Different Approaches. In *Lecture Notes in Computer Science*, volume 2991, pages 64–90, 2004.

[75] Th. C. Hales. The flyspeck project. `http://code.google.com/p/flyspeck/`.

[76] E. Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker, 1992.

[77] J. Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[78] J. Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.

[79] 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*, pages 102–118. Springer-Verlag, 2007.

[80] IBM. *IBM High-Accuracy Arithmetic Subroutine Library (ACRITH)*, 1986.

[81] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008, August 2008. available at `http://ieeexplore.ieee.org/servlet/opac?punumber=4610933`.

[82] C. Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. PhD thesis, University of the Saarland, 2002. Available on the Web as `http://engr.smu.edu/~seidel/research/diss-jacobi.ps.gz`.

[83] R. Kaivola and M. D. Aagaard. Divider circuit verification with model checking and theorem proving. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 338–355. Springer-Verlag, 2000.

[84] E.W. Kaucher and W.L. Miranker. *Self-validating numerics for function space problems*. Academic Press, 1984.

[85] R. B. Kearfott. *Rigorous Global Search: Continuous Problems*. Kluwer, Dordrecht, Netherlands, 1996.

[86] P. Kirchberger. *Über Tchebychefsche Annäherungsmethoden*. Ph.D. thesis, Göttingen, Germany, 1902.

[87] D. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, 3rd edition, 1997.

[88] W. Krämer. Sichere und genaue Abschätzung des Approximationsfehlers bei rationalen Approximationen. Technical report, Institut für angewandte Mathematik, Universität Karlsruhe, 1996.

[89] C. Lanczos. Trigonometric interpolation of empirical and analytical functions. *J. Math. Phys*, 17:123–199, 1938.

[90] C. Lanczos. *Applied analysis*. Prentice-Hall, 1956.

[91] C. Lauter and F. de Dinechin. Optimising polynomials for floating-point implementation. In *Proceedings of the 8th Conference on Real Numbers and Computers*, pages 7–16, 2008.

[92] D.-U. Lee, J.D. Villasenor, W. Luk, and P.H.W. Leong. A hardware gaussian noise generator using the Box-Muller method and its error analysis. *IEEE Transactions on Computers*, 55(6), 2006.

[93] D.U. Lee, A.A. Gaffar, O. Mencer, and W. Luk. Optimizing hardware function evaluation. *IEEE Transactions on Computers*, 54(12):1520–1531, December 2005.

[94] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic (ARITH-16)*, Vail, CO, June 2001.

[95] S. Lewanowicz. Construction of a recurrence relation of the lowest order for coefficients of the Gegenbauer series. *Zastosowania Matematyki*, XV(3):345–395, 1976.

[96] S. Lewanowicz. A new approach to the problem of constructing recurrence relations for the jacobi coefficients. *Zastos. Mat*, 21:303–326, 1991.

[97] K. Makino. *Rigorous Analysis of Nonlinear Motion in Particle Accelerators*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 1998.

[98] K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4(4):379–456, 2003. http://bt.pa.msu.edu/pub/papers/TMIJPAM03/TMIJPAM03.pdf.

[99] K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4(4):379–456, 2003.

[100] K. Makino and M. Berz. Rigorous integration of flows and odes using taylor models. In *Proceedings of the 2009 conference on Symbolic numeric computation*, SNC '09, pages 79–84, New York, NY, USA, 2009. ACM.

[101] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[102] J. C. Mason and D. C. Handscomb. *Chebyshev polynomials*. Chapman & Hall/CRC, Boca Raton, FL, 2003.

[103] R. J. Mathar. Chebyshev series expansion of inverse polynomials. *J. Comput. Appl. Math.*, 196(2):596–607, 2006.

[104] K. Mehlhorn, S. Nher, M. Seel, R. Seidel, Th. Schilz, S. Schirra, and Ch. Uhrig. Checking geometric programs or verification of geometric structures. In *Proceedings of the 12th Annual Symposium on Computational Geometry (FCRC'96)*, pages 159–165, Philadelphia, 1996. Association for Computing Machinery.

[105] G. Melquiond. Floating-point arithmetic in the Coq system. In *Proc. of the 8th Conference on Real Numbers and Computers*, pages 93–102, 2008.

[106] F. Messine. *Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution de problèmes avec contraintes*. PhD thesis, INP de Toulouse, 1997.

[107] M. Mezzarobba. Numgfun: a package for numerical and analytic computation with d-finite functions. In Stephen M. Watt, editor, *ISSAC 2010: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation, 25-28 July 2010, Munich, Germany*, pages 139–146. ACM, 2010.

[108] M. Mezzarobba and B. Salvy. Effective bounds for p-recursive sequences. *Journal of Symbolic Computation*, 45(10):1075–1096, 2010.

[109] P. Molin. *Intégration numérique et calculs de fonctions L*. Ph.D. thesis, Université Bordeaux 1, 2010.

[110] M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, Waterloo ON, Canada, 2005.

[111] J. Strother Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division program. *IEEE Transactions on Computers*, 47:913–926, 1998.

[112] R. E. Moore. *Interval Arithmetic and Automatic Error Analysis in Digital Computing*. Ph.D. dissertation, Department of Mathematics, Stanford University, Stanford, CA, USA, November 1962. Also published as Applied Mathematics and Statistics Laboratories Technical Report No. 25.

[113] R. E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[114] R. E. Moore. *Methods and Applications of Interval Analysis*. Society for Industrial and Applied Mathematics, 1979.

[115] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to interval analysis*. SIAM, Philadelphia, PA, USA, 2009.

[116] J.-M. Muller. Projet ANR TaMaDi – dilemme du fabricant de tables – table maker's dilemma (ref. ANR 2010 BLAN 0203 01). `http://tamadiwiki.ens-lyon.fr/tamadiwiki/`.

[117] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhäuser Boston, MA, 2nd edition, 2006.

[118] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010. ACM G.1.0; G.1.2; G.4; B.2.0; B.2.4; F.2.1., ISBN 978-0-8176-4704-9.

[119] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torrès. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, November 2009.

[120] N. Th. Müller. The iRRAM: Exact arithmetic in c++.

[121] P. S. V. Nataraj and K. Kotecha. Global optimization with higher order inclusion function forms part 1: A combined Taylor-Bernstein form. *Reliable Computing*, 10(1):27–44, 2004.

[122] N. S. Nedialkov, K. R. Jackson, and G. F. Corliss. Validated solutions of initial value problems for ordinary differential equations. *Applied Mathematics and Computation*, 105(1):21 – 68, 1999.

[123] M. Neher, K. R. Jackson, and N. S. Nedialkov. On Taylor model based integration of ODEs. *SIAM J. Numer. Anal.*, 45:236–262, 2007.

[124] A. Neumaier. Taylor forms – use and limits. *Reliable Computing*, 9(1):43–79, 2003.

[125] J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1–14, 1999. Available on the Web as `http://download.intel.com/technology/itj/q11999/pdf/floating_point.pdf`.

[126] R. Pachón and L. N. Trefethen. Barycentric-Remez algorithms for best polynomial approximation in the chebfun system. *BIT Numerical Mathematics*, 49(4):721–741, 2009.

[127] V. Y. Pan. Optimal and nearly optimal algorithms for approximating polynomial zeros. *Comput. Math. Appl*, 31:97–138, 1996.

[128] V. Y. Pan. New fast algorithms for polynomial interpolation and evaluation on the Chebyshev node set. *Comput. Math. Appl.*, 35(3):125–129, 1998.

[129] P. A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Mathematical Programming*, 96:293–320, 2003.

[130] S. Paszkowski. Zastosowania numeryczne wielomianow i szeregow Czebyszewa. *Podstawowe Algorytmy Numeryczne*, 1975.

[131] R. B. Platte and L. N. Trefethen. Chebfun: A New Kind of Numerical Computing. In E. Fitt, A. D.; Norbury, J.; Ockendon, H.; Wilson, editor, *Progress in Industrial Mathematics at ECMI 2008*, pages 69–87. Springer, 2010.

[132] M. J. D. Powell. On the maximum errors of polynomial approximations defined by interpolation and by least squares criteria. *Comput. J.*, 9:404–407, 1967.

[133] M. J. D. Powell. *Approximation theory and methods*. Cambridge University Press, 1981.

[134] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C, The Art of Scientific Computing, 2nd edition*. Cambridge University Press, 1992.

[135] The Arénaire Project. CRlibm, Correctly Rounded mathematical library, July 2006. http://lipforge.ens-lyon.fr/www/crlibm/.

[136] L. B. Rall. The arithmetic of differentiation. *Mathematics Magazine*, 59(5):275–282, 1986.

[137] H. Ratschek and J. Rokne. *New computer methods for global optimization*. Ellis Horwood Ltd, 1988.

[138] L. Rebillard. *Etude théorique et algorithmique des series de Chebychev, solutions d'équations différentielles holonomes*. PhD thesis, Institut national polytechnique de Grenoble, 1998.

[139] N. Revol. Interval Newton Iteration in Multiple Precision for the Univariate Case. *Numerical Algorithms*, 34(2):417–426, 2003.

[140] N. Revol. Standardized Interval Arithmetic and Interval Arithmetic Used in Libraries. In *ICMS 2010 - Third International Congress on Mathematical Software Lecture Notes in Computer Science*, volume 6327, pages 337–341, Kobe Japan, 2010. Takayama, Nobuki and Fukuda, Komei and van der Hoeven, Joris and Joswig, Michael and Noro, Masayuki, Springer.

[141] N. Revol and F. Rouillier. The MPFI library. http://gforge.inria.fr/projects/mpfi/.

[142] T. J. Rivlin. *Chebyshev polynomials. From approximation theory to algebra*. Pure and Applied Mathematics. John Wiley & Sons, New York, 2nd edition, 1990.

[143] F. Rouillier and P. Zimmermann. Efficient isolation of polynomial's real roots. *Journal of Computational and Applied Mathematics*, 162(1):33–50, 2004.

[144] M.-F. Roy. *Basic algorithms in real algebraic geometry and their complexity: from Sturm's theorem to the existential theory of reals*, volume 23 of *Expositions in Mathematics*. de Gruyter, 1996. in F. Broglia (Ed.), Lectures in Real Geometry.

[145] S. Rump. *Developments in Reliable Computing, T. Csendes ed.*, chapter INTLAB - Interval Laboratory, pages 77–104. Kluwer, 1999.

[146] S. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):534–554, 1999.

[147] S. M. Rump. Algorithms for verified inclusion. In R. Moore, editor, *Reliability in Computing, Perspectives in Computing*, pages 109–126. Academic Press, New York, 1988.

[148] 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*, 1:148–200, 1998. Available on the Web at http://www.russinoff.com/papers/k7-div-sqrt.html.

[149] B. Salvy. D-finiteness: Algorithms and applications. In Manuel Kauers, editor, *ISSAC'05*, pages 2–3. ACM Press, 2005. Abstract for an invited talk.

[150] B. Salvy and P. Zimmermann. Gfun: a Maple package for the manipulation of generating and holonomic functions in one variable. *ACM Transactions on Mathematical Software*, 20(2):163–177, 1994.

[151] M. Schatzman. *Numerical Analysis, A Mathematical Introduction*. Oxford University Press, 2002.

[152] V. Stahl. *Interval Methods for Bounding the Range of Polynomials and Solving Systems of Nonlinear Equations*. PhD thesis, Johannes Kepler University Linz, Linz, Austria, 1995.

[153] R. P. Stanley. Differentiably finite power series. *European Journal of Combinatorics*, 1(2):175–188, 1980.

[154] T. Sunaga. Theory of interval algebra and its application to numerical analysis. *RAAG Memoirs, Ggujutsu Bunken Fukuy-kai, Tokyo*, 2:29–46 (547–564), 1958.

[155] D. A. Sunderland, R. A. Strauch, S. S. Wharfield, H. T. Peterson, and C. +R. Role. CMOS/-SOS frequency synthesizer LSI circuit for spread spectrum +communications. *IEEE Journal of Solid-State Circuits*, 19(4):497–506, August 1984.

[156] The PARI Group, Bordeaux. *PARI/GP, version* `2.3.4`, 2011. available from `http://pari.math.u-bordeaux.fr/`.

[157] A. Tisserand. High-performance hardware operators for polynomial evaluation. *International Journal of High Performance Syststem Architectures*, 1:14–23, April 2007.

[158] L. N. Trefethen. *Approximation Theory and Approximation Practice, Draft version, June 11, 2011*, `http://www2.maths.ox.ac.uk/chebfun/ATAP/`.

[159] L. N. Trefethen. Computing numerically with functions instead of numbers. *Mathematics in Computer Science*, 1(1):9–19, 2007.

[160] L. N. Trefethen et al. *Chebfun Version 4.0*. The Chebfun Development Team, 2011. `http://www.maths.ox.ac.uk/chebfun/`.

[161] W. Tucker. A Rigorous ODE Solver and Smale's 14th Problem. *Foundations of Computational Mathematics*, 2(1):53–117, 2002.

[162] W. Tucker. Auto-validating numerical methods, 2009.

[163] J. van der Hoeven. Making fast multiplication of polynomials numerically stable. Technical Report 2008-02, Université Paris-Sud, Orsay, France, 2008.

[164] J. van der Hoeven. Ball arithmetic. In Arnold Beckmann, Christine Gaßner, and Bededikt Löwe, editors, *Logical approaches to Barriers in Computing and Complexity*, number 6 in Preprint-Reihe Mathematik, pages 179–208. Ernst-Moritz-Arndt-Universität Greifswald, February 2010. International Workshop.

[165] L. Veidinger. On the numerical determination of the best approximations in the Chebyshev sense. *Numerische Mathematik*, 2:99–105, 1960.

[166] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, New York, 2nd edition, 2003.

[167] F. Wiedijk. *The seventeen provers of the world*, volume 3600 of *Lecture Notes in Computer Science*. Springer, 2006.

[168] J. Wimp. *Computation with Recurrence Relations*. Pitman, Boston, 1984.

[169] K. E. Wires, M. J. Schulte, and D. McCarley. FPGA resource reduction through truncated multiplication. In *International Conference on Field Programmable Logic and Applications*, pages 574–583. Springer-Verlag, 2001.

[170] S. Wolfram. *The Mathematica Book*. Wolfram Media, Incorporated, 5 edition, 2003.

[171] R. C. Young. The algebra of multi-valued quantities. *Mathematische Annalen*, 104(12):260–290, 1931.

[172] R. V. M. Zahar. A mathematical analysis of miller's algorithm. *Numerische Mathematik*, 27(4):427–447, 1976.

[173] D. Zeilberger. A holonomic systems approach to special functions identities. *Journal of Computational and Applied Mathematics*, 32(3):321–368, 1990.

[174] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.

[175] R. Zumkeller. Formal Global Optimization with Taylor Models. In *Proc. of the 4th International Joint Conference on Automated Reasoning*, pages 408–422, 2008.

[176] R. Zumkeller. *Global Optimization in Type Theory*. PhD thesis, École polytechnique, 2008.