Taylor Series Revisited

Xavier Thirioux, Alexis Maffart

RAPPORT TECHNIQUE

Réf. ISAE/DISC/RT2020/1

TAYLOR SERIES REVISITED

Réalisé par

Xavier Thirioux[(1)] et Alexis Maffart[(2)]
(1) ISAE-SUPAERO, Université de Toulouse, France
(2) IRIT, Toulouse INP, Université de toulouse, France

ISAE SUPAERO - Institut Supérieur de l'Aéronautique et de l'Espace
10, avenue Edouard-Belin - BP 54032 - 31055 Toulouse CEDEX 4 - FRANCE
Tél : 33 (0)5 61 33 80 80 - Télécopie : 33 (0)5 61 33 83 30 - Site internet : www.isae-supaero.fr

Taylor series revisited

Xavier Thirioux (ISAE - Supaéro, Toulouse, France)
Alexis Maffart (IRIT, Toulouse, France)

September 7, 2020

Abstract

We propose a renovated approach around the use of Taylor expansions to provide polynomial approximations. We introduce a coinductive type scheme and finely-tuned operations that altogether constitute an algebra, where our multivariate Taylor expansions are first-class objects. As for applications, beyond providing classical expansions of integro-differential and algebraic expressions mixed with elementary functions, we demonstrate that solving ODE and PDE in a direct way, without external solvers, is also possible. We also discuss the possibility of computing certified errors within our scheme.

Contents

1 Motivations

1.1 Taylor expansions

Our principal motivation is to provide an automatic way of approximating arbitrary multivariate numerical expressions, involving elementary functions, integrations, partial derivations and arithmetical operations. In terms of features, we propose an approach where Taylor expansions are first-class objects of our programming language, computed *lazily* on demand at any order. Finally, we also wish to obtain certified errors, which will by the end include errors of approximation and numerical errors, expressed in any suitable user-provided error domain, such as zero-centered intervals, intervals, zonotopes, etc. From a user's perspective, a typical workflow is first to compute a certified approximation at some order of some expression, second to evaluate the maximum error for the given domains of variables, and maybe third to compute a finer approximation at some higher order (without recomputing previous values) if the error is too coarse, and so on, until the approximation meets the user's expectations in terms of precision. We postulate that the expressions at hand are indeed analytical and possess a valid Taylor expansion around a given point and within variables' domains. If it is not the case, then the error computed at every increasing order won't show any sign of diminishing and could even diverge. Last but not least, our approach yields a direct means to express solutions to ODEs and PDEs and thus solve them, without complex numerical methods based on domains discretization.

Furthermore, we aim at bringing as much robustness and correction as possible to our library through a correct-by-construction approach. The type system is in charge of the correction as it ensures, at compile time, that dimensions of various tensors, functions, convolutions and power series conform to their specifications. This is of a particular importance in a complex and error-prone context involving a vast number of numerical computations such as ODEs and PDEs resolution. The type system which validates all dimension related issues greatly helps in reducing the focus on purely numerical concerns: correctness of approximation, precision, convergence. Moreover, correction could be proved more formally with a proof assistant such as COQ. This idea could be adressed in the future even if this work is likely to be laborious.

As a disclaimer, the current state of our contribution doesn't allow yet the computation of certified errors in the presence of differential equations, so we mainly focus here on infinite Taylor expansions without remainders. Still, as one of our prominent future goals, certified errors were taken into account in the design stage of our framework and we discuss them along this paper.

1.2 Applications

Among many possible applications, we more specifically aim at formally verifying systems dealing with complex numerical properties, such as controllers for embedded systems. Moreover, through certified integration of ODE, we may

also consider hybrid systems, such as a continuous plant coupled to a discrete controller.

1.3 Outline

We start by recalling some related works around formalization and mechanization of Taylor expansions in section 2. Then, we state a mathematical formulation of our on-demand multivariate Taylor expansions with errors, together with an implementation of our main data-structure, in section 3. We detail our implementation of operations that form an algebra for infinite Taylor series, i.e. without remainders, in section 4. We separately discuss the more complex case of composition in section 5. In section 6, we present some experiments done on solving differential equations in a direct way. In section 7, we discuss the specific issues raised by computing certified errors that allow to deal with finite Taylor expansions with (certified) remainders. Finally, we open up some perspectives then conclude, respectively in sections 9 and 10.

2 Related works

2.1 Taylor series

Although Taylor expansions are well known and form a very rich and interesting algebra, their realizations as software items are not widespread. From a mathematical perspective, some weaknesses may explain this lack of success: they only support analytical functions, a rather limited class of functions; they don't possess good convergence properties, uniform convergence is hardly guaranteed for instance; typical applications for polynomial approximations are usually not concerned with certified errors, mean error or integrated square error (through various norms) are more important and don't easily fit into Taylor expansion schemes. Finally, from a programming perspective, Taylor expansions are: hard to implement as they require many different operations to be implemented, from low-level pure numbers to high-level abstract Taylor expansions seen as first-class citizens; error-prone with lots of complex floating-point computations on non-trivial data structures; heavily resource demanding in our multi-dimensional setting because data structures rapidly grow as the precision order increases.

Here are a few works dealing with Taylor expansions. In [7], the author presents an early application of laziness to cleanly obtain Taylor polynomial approximations. Laziness allows to augment the degree of the resulting polynomial on demand. Yet, the setting is much simpler as it is strictly one-dimensional and certified errors are not in scope. With these restrictions, the author obtains nice formulations of automatic differentiation and polynomial approximations of classical phenomena in physics. Speaking about implementation, related works come in many flavors and date back to the now well established folklore of automatic differentiation (forward or backward modes). As for symmetric tensor

algebra, which forms a well-suited representation basis for partial derivatives, a huge menagerie of (mostly C++) libraries exists, for tensors of arbitrary orders and dimensions (but some libraries put a very low upper-bound on these values). These implementations are clearly not oriented towards reliability and proof of correctness, but towards mere efficiency. This also comes at the expense of some user-friendliness, as memory management and user interface are more complex and error-prone than in our own library. Still, we may consider interfacing our code base with a trusted and stable tensor library, for much better performance.

One of the most prominent implementation of Taylor expansions is the COSY tool, *cf.* [12, 9]. This tool has been used in industrial-scale engineering and scientific contexts, to modelize and predict the complex dynamics of particles in accelerators for instance. This tool supports $1D$ Taylor expansions with interval-based certified errors. Polynomial degree is not refinable on demand and Taylor expansions are not handled *per se* (*i.e.* not first-class citizens). The authors managed anyway to implement an error refinement scheme for solved form ordinary differential equations, that allows solving them with tight certified errors. Experiments show that this tool compares favorably to other traditional approximations and bounding techniques, such as branch-and-bound approaches and interval arithmetics, in terms of speed and precision. We also aim at implementing differential equation solving in our multi-dimensional setting.

At the other end of the spectrum, [11] proposes correct-by-construction univariate Taylor expansions with certified errors, which appears as a huge step. Integration of floating-point errors into this scheme is also a concern addressed in [10]. Still, apart from its limitation to the $1D$ case, this approach suffers from weaknesses: expansion degree is fixed and differential equations cannot be handled. The underlying algorithm won't be so easily turned into a co-inductive (lazy) equivalent version.

And in the middle of the spectrum comes [1], where the author defines a way to handle multivariate Taylor series and presents its implementation featuring on demand computation thanks to Scheme laziness. The few points he did not implement and that we will try to cope with in our library are: errors certification which is not handled and efficiency which is not optimal. For instance, the author's method to multiply multivariate power series is to define a generic composition between a bivariate function and a power series and to instantiate it with the multiplication. This method is simply built upon the chain rule but has some drawbacks. First, the generic equation given can usually be drastically simplified for instance in the case of multiplication and second, such a generic scheme implies that some parts of the resulting coefficients will be computed several times differently. Conversely, in our solution, the pervasive multiplication operation is implemented with a strong concern on optimality.

Our work and specifically our data-structure is based on the dissertation [13, Part 2], with the nuance that a single unbounded tree will be used instead of an infinite sequence of finite trees, each such tree representing a symmetric tensor of a given order. This choice notably enables the resolution of partial differential equations, which was impossible in the setting of [13].

2.2 Differential equations

Iterative methods are pervasive in integrating differential equations because they often provide an efficient way to find an approximation of an ODE solution. Some of them own validation aspects, such as [4] which relies on Runge-Kutta method to integrate ODE with a numerical validation. The main difference between these methods and our work as a direct method is that we don't need these next level iterations. We are able to yield a result in the equivalent of the first iteration.

3 Formalization

We recall the canonical presentation of a multivariate Taylor expansion at order R in dimension N. This expansion converges to $f(\mathbf{x})$ when $R \to +\infty$ for an **analytical** function f only in a chosen neighbourhood of point $\mathbf{0}$.

$$f(\mathbf{x}) = \sum_{|\alpha|<R} \mathbf{D}_f^\alpha(\mathbf{0}) \cdot \frac{\mathbf{x}^\alpha}{\alpha!} + \sum_{|\alpha|=R} \mathbf{D}_f^\alpha(\lambda * \mathbf{x}) \cdot \frac{\mathbf{x}^\alpha}{\alpha!} \tag{1}$$

In the above formulation, $\mathbf{x} = (\mathbf{x}_0, \ldots, \mathbf{x}_{N-1}) \in \mathbb{R}^N$, $\alpha = (\alpha_0, \ldots, \alpha_{N-1}) \in \mathbb{N}^N$ indexes the derivation order of f in the symmetric tensor of partial derivatives $\mathbf{D}_f^\alpha$ and $\lambda \in [0, 1]$ is an unknown coefficient that characterizes the exact Taylor remainder. We have to compute derivatives both at point $\mathbf{0}$ for the polynomial part and at point $\lambda * \mathbf{x}$ for the error part. We choose to use a single co-inductive data-structure that encodes all possible derivatives, indexed by some α. As for the elements of this structure, we handle $\langle value, error \rangle$ pairs. Our framework is error-agnostic as the value-error domain is user-defined and only requires arithmetical operations. Several solutions are available in the literature: zero-centered intervals, intervals, zonotopes, etc. In the remainder, we only assume that elements of our structures form an algebra (including addition, multiplication and some elementary functions), disregarding whether they are pure values or values with errors.

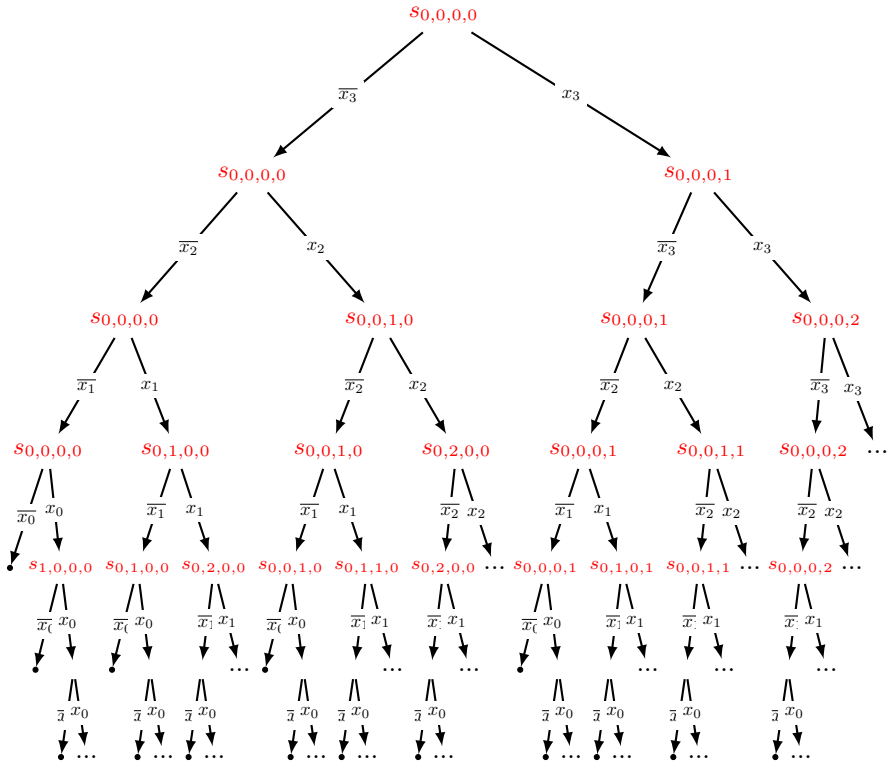
This co-inductive structure, that we coin a "cotensor", enables to compute finer approximations on demand and also to lazily represent expansions of solutions to ODEs and PDEs, when they are expressed in solved form, i.e. not implicit (as it would for instance be the case if the solution were specified as a zero of a polynomial form in a functional space).

3.1 Data structure

Coefficients are present in each node of a unique tree structure and are written as $s_{o_0,\ldots,o_{N-1}}$ where every o_i is the number of occurences of the variable x_i in the path that leads to the considered coefficient $s_{o_0,\ldots,o_{N-1}}$.

The principle is quite simple: at each node, we choose either to keep the same variable accounting for the final Taylor series, or we drop it and repeat the same process for lower dimension variables. This is pictured in tree branches of the following example as x_i for the first case and $\overline{x_i}$ for the second case. The

variable at the root of the tree is X_n if the dimension is $n+1$. This tree is developed below and represents a symmetric cotensor s of dimension 4 :



3.2 Structural decomposition

We will introduce for this co-inductive structure a few notations inspired from the computation of the quotient and the remainder with respect to variable X_n. We will call a left cotensor a cotensor which is the left branch of another cotensor and we will denote L_{n+1} the set of left symmetric cotensors and R_{n+1} the set of right symmetric cotensors in dimension $n+1$. If V is the set of labels at the root of the tree, we have the following definitions:

$$\begin{array}{lll} & L_{n+1} & \triangleq L_n + X_n.R_{n+1} \\ & R_{n+1} & \triangleq L_n + X_n.R_{n+1} + V \\ \text{Hence :} & R_{n+1} & = L_{n+1} + V \end{array}$$

We note that the only difference between left and right cotensors is the constant part $v \in V$ and from now, we are going to consider that right case is the general one and that left case is the specification of the right case with constant part equal to 0. This will prevent us from writing similar redundant equations for all algebraic operations we will describe later. A cotensor is, then, considered a

right cotensor by default, even if it has no parent because it contains a significant value $v \in V$ which is the constant part of the Taylor series. It comes then that a tree is interpreted as a Taylor series by adding together the term for the left tree, X_n times the term for the right tree and the label value of the root.

3.3 Implementation

Finally, in terms of OCaml implementation, this decomposition scheme naturally translates into the slightly relaxed following type definition, where L_n and R_n have been conflated in a single type:

```
type ('a, _) st =
  | Nil:   ('a, Nat.zero) st
  | Leaf:  ('a, 'n Nat.succ) st
  | Node:    ('a, 'n) st Lazy.t
           *  'a
           *  ('a, 'n Nat.succ) st Lazy.t
           -> ('a, 'n Nat.succ) st
  and
  ('a, 'n) tree = ('a, 'n) st Lazy.t
```

Here the type of symmetric cotensors `tree` has two type parameters: the type of elements `'a` and the dimension type `'n`. The last parameter not being constant through recursion, it appears as `_` in the type declaration. Then, the two cases for the dimension N: $N = 0$ and $N \neq 0$, are respectively handled with `Nil` and `Leaf`/`Node` constructors. `Leaf` is only a special case of `Node` where all the coefficients are zeros. Handling this particular case with a different constructor aims at saving some computations, for instance all polynomial forms will be represented by finite trees, not by unbounded ones with trailing zeros. And `Nil` constructor is used to mark the end of a branch when the dimension has decreased to 0, namely all the variables has been consumed. Type parameters of constructors' arguments behave accordingly to the decomposition of R_{n+1}.

The `Nat.zero` and `Nat.succ` type constructors encode the dimensions of manipulated cotensors, as we use GADT allowed by OCaml. We use a standard type-level encoding of Peano numbers and operations that we don't detail here. We hereby enforce a correct-by-construction use of our data-structures.

Finally, we assume throughout this presentation that cotensor elements (the parameter `'a` of type `('a, 'n) tree`) form a field, with arithmetical operations on it. It may be in practice a field of coefficients or/and errors. Most operations of our Taylor algebra are totally agnostic about the real nature of these coefficients.

4 An algebra of Taylor series

From this section until section 6 included, coefficients of infinite Taylor series we consider are only made of values (of derivatives), as opposed to remainders

[0] Generalized Algebraic Data Types

and errors.

4.1 Component-wise operations

These elements are denoted by V_A and V_B. Functions "$\lambda.\,\bullet$" and "$\bullet + \bullet$" straightforwardly witness the vector space structure of cotensors. The Hadamard product "$\bullet \odot \bullet$" is the component-wise product of two cotensors of same dimension. Hence with the notation $A_{n+1} \triangleq A_n^L + X_n.A_{n+1}^R + V_A$:

$$
\begin{aligned}
A_{n+1} + B_{n+1} &= (A_n^L + B_n^L) + X_n.(A_{n+1}^R + B_{n+1}^R) + (V_A + V_B) \\
\lambda.A_{n+1} &= \lambda.A_n^L + \lambda.X_n.A_{n+1}^R + \lambda.V_A \\
A_{n+1} \odot B_{n+1} &= (A_n^L \odot B_n^L) + X_n.(A_{n+1}^R \odot B_{n+1}^R) + (V_A * V_B)
\end{aligned}
$$

```
let rec somme : type n. (R.t, n) tree -> (R.t, n) tree -> (R.t, n) tree =
 fun st1 st2 ->
  lazy (
   match (Lazy.force st1), (Lazy.force st2) with
    | Nil                , Nil                -> Nil
    | st1                , Leaf               -> st1
    | Leaf               , st2                -> st2
    | Node(st1l, v1, st1r), Node(st2l, v2, st2r) -> Node(somme st1l st2l,
                                                         R.(v1 + v2),
                                                         somme st1r st2r)
  )

let lambda k st = linear_map (R.( * ) k) st

let rec hproduit : type n. (R.t, n) tree -> (R.t, n) tree -> (R.t, n) tree =
 fun st1 st2 ->
  lazy (
   match (Lazy.force st1), (Lazy.force st2) with
    | Nil                , Nil                -> Nil
    | _                  , Leaf               -> Leaf
    | Leaf               , _                  -> Leaf
    | Node(st1l, v1, st1r), Node(st2l, v2, st2r) -> Node(hproduit st1l st2l,
                                                         R.(v1 * v2),
                                                         hproduit st1r st2r)
  )
```

The structure of the implementation of these linear functions are the same. They are always recursively applying the operator along the trees while wrapping each step in a lazy call.

4.2 Multiplication

Let S be the following cotensor :

$$\begin{aligned} S(X_0, ..., X_N) &= (S_0 + S_1 \odot X + S_2 \odot X^2 + ... + S_m \odot X^m + ...) \quad \text{shortened in} \\ &= (S_0 + S_1 X + S_2 X^2 + ... + S_m X^m + ...) \\ \text{where} \quad X &= (X_0, ..., X_N) \end{aligned}$$

This notation is inspired by derivation order; even if we do not consider order of cotensors; because it will be of a great help when defining the multiplication and introducing the convolution product. Product of Taylor expansions is really pervasive and appears in many operations (derivation formulas, composition of Taylor series, etc). It is naturally defined with an explicit convolution. Concretely:

$$\begin{aligned} S(X_0, ..., X_N) \times T(X_0, ..., X_N) &= (S_0 + S_1 X + ... + S_p X^p + ...) \\ &\quad \times (T_0 + T_1 X + ... + T_q X^q + ...) \\ &= R_0 + R_1 X + R_2 X^2 + ... + R_k X^k + ... \\ \text{where} \quad \forall k \in \mathbb{N}, \quad R_k &= \sum_{i=0}^{k} S_i T_{k-i} \end{aligned}$$

To compute the coefficients at order k, we need to consider every product that will produce an order k, *i.e.* every coefficient of ordrer i by every coefficient of ordre $k - i$, i ranging from 0 to k.

In our setting, we maintain a typed convolution structure to express computation of the term $\sum_{i=0}^{k} S_i T_{k-i}$. This structure, while geared towards static guarantees and proof of correctness, still allows for some efficient implementation. Informally, we may specify our structure as an array containing couples of cotensors of a specific dimension and that will represent absolute paths. The same structure is used to represent relative paths. We introduce a path notation, illustrated by the following examples in dimension n:

- () is the considered tree
- (n) is the tree we get when we take the n-th variable (X_n) once in the considered tree
- $(n.n.n_{-1})$ when we take the X_n variable twice and then X_{n-1} once

The so called "considered tree" is the original tree given in parameter if considering absolute paths or a specific tree (descendant of the original one) if considering relative paths. Through the relative paths (left part of the semicolon), we will store the number of times we went down a right branch since the last left branch, namely relative paths are about the current variable and absolute paths are about all previous variables with respect to the order. Initially, the structure contains a couple of the two original trees given in parameter for both absolute paths (and the part for relative paths is empty) :

$$\left|\begin{array}{ccc} () & ; & () \\ () & & () \end{array}\right|$$

```
type _ conv_t =
| T : (((R.t, 'n) tree) * ((R.t, 'n) tree)) Seq.t -> 'n conv_t
```

`conv_t` is the type of the data structure containing convolution products. We said above that it could informally be specified as an array containing couple of cotensors but the definition is actually generic since `Seq.t` represents a sequence type which can be then specified as a list, an array or any iterator.

Then, at each step of the algorithm :

- If the current node is a right branch, we will update the relative paths by adding the current node ($k.k$ here) and shifting the lines as follows :

$$\left|\begin{array}{cccc} () & (k) & ; & \dots \\ (k) & () & & \dots \end{array}\right| \quad \text{becomes} \quad \left|\begin{array}{ccccc} () & (k) & (k.k) & ; & \dots \\ (k.k) & (k) & () & & \dots \end{array}\right|$$

- if the current node is a left branch, we will combine the relative paths with the absolute ones, store the result as the new absolute paths and empty the new relative paths :

$$\left|\begin{array}{ccccc} () & (n-1) & ; & () & (n) \\ (n-1) & () & & (n) & () \end{array}\right| \quad \text{becomes} \quad \left|\begin{array}{ccccccc} () & ; & () & (n) & (n-1) & (n-1.n) \\ () & & (n-1.n) & (n-1) & (n) & () \end{array}\right|$$

```
let distribute_aux : type n. int -> int ->
 (R.t, n Nat.succ) tree -> (R.t, n Nat.succ) tree ->
 ((R.t, n Nat.succ) tree * (R.t, n Nat.succ) tree) Seq.t ->
 ((R.t, n Nat.succ) tree * (R.t, n Nat.succ) tree) Seq.t =
 fun a i t1 t2 res_q ->
  (match (Lazy.force (go_through (a-i) t1), Lazy.force (go_through i t2)) with
   | Leaf, _ -> res_q
   | _, Leaf -> res_q
   | new_st1, new_st2 -> S.cons ((lazy new_st1), (lazy new_st2)) res_q)

let distribute_left : type n . int ->
 (R.t, n Nat.succ) tree -> (R.t, n Nat.succ) tree ->
 (((R.t, n Nat.succ) tree) * ((R.t, n Nat.succ) tree)) Seq.t =
 let rec aux =
  fun a i t1 t2 acc -> match i with
   | -1 -> acc
   | _  -> aux a (i-1) t1 t2 (distribute_aux a i t1 t2 acc)
 in fun a t1 t2 -> aux a a t1 t2 S.nil

let rec combine : type n. int -> n Nat.succ conv_t -> n Nat.succ conv_t =
 fun a (T conv_it) ->
  T S.(conv_it >>= fun (t1, t2) -> distribute_left a t1 t2)
```

Each of these functions deals with a particular level of combinatorics of the distribution. So basically, combining the relative paths with the absolute ones boils down to combining a integer with a convolution product. These objects are the parameters of the `combine` function which distributes the integer with the elements of the convolution in all different way and which binds the results at the end. An element of the convolution product is a couple of trees. The `distribute_left` function distributes the couple of trees with all the integers between 0 and the parameter a, once again binding the results together and finally, the `distribute_aux` function does the actual link between a relative path and an absolute one. Associating a relative path to an absolute one means concatenating them. Speaking in terms of trees, it means that the relative path begins where the absolute one ends in the tree.

Folding this structure to compute a term of a product simply consists in combining relative paths with absolute paths, multiplying cotensors roots column-wise and then summing these intermediate results altogether with the following functions :

```
let roots_prod : type n. (R.t, n Nat.succ) tree -> (R.t, n Nat.succ) tree -> R.t =
 fun st1 st2 ->
  match (Lazy.force st1, Lazy.force st2) with
   | Leaf, _                          -> zeroR
   | _, Leaf                          -> zeroR
   | Node(_, v1, _), Node(_, v2, _) -> R.(v1 * v2)

let prod_conv : type n. n Nat.succ conv_t -> R.t =
 fun (T it) -> Seq.fold it R.zero R.(fun acc (t1, t2) -> acc + (roots_prod t1 t2))
```

4.3 Differential operations

Cotensors of dimension N may not only be structurally decomposed on X_{N-1} but also on any other X_k, which we would call a non-structural decomposition. For that purpose, the "$\mathbf{.}[\mathbf{.}]$" function specializes a cotensor, *i.e.* drops some index by specializing it to a specific dimension k, and therefore represents the division by a monomial X_k. Conversely, the "$\mathbf{.} \uparrow \mathbf{.}$" function represents the multiplication by a monomial X_k. For a cotensor of dimension N, they are defined in terms of polynomials as:

$$\begin{aligned}(\mathbf{S}[k])(X_0,\ldots,X_{N-1}) &\triangleq \tfrac{\mathbf{S}(X_0,\ldots,X_{N-1})-\mathbf{S}(X_0,\ldots,X_{k-1},0,X_{k+1},\ldots,X_{N-1})}{X_k}\\ (\mathbf{S}\uparrow k)(X_0,\ldots,X_{N-1}) &\triangleq X_k.\mathbf{S}(X_0,\ldots,X_{N-1})\end{aligned}$$

Using the same notations as for component-wise operations, we show how

these operators simply fit the structural decomposition:

$$
\begin{aligned}
\mathbf{S}[k] &= (\mathbf{S}^L + X_{N-1}.\mathbf{S}^R + V_S)[k] \\
&= \begin{cases} \mathbf{S}^R, \text{ for } k = N-1 \\ \frac{\mathbf{S}^L + X_{N-1}.\mathbf{S}^R - \mathbf{S}^L{}_{|X_k \leftarrow 0} - X_{N-1}.\mathbf{S}^R{}_{|X_k \leftarrow 0} + V_S - V_S}{X_k} = \mathbf{S}^L[k] + X_{N-1}.\mathbf{S}^R[k], \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{for } k < N \end{cases} \\
\mathbf{S}{\uparrow}k &= \begin{cases} \mathbf{0} + X_{N-1}.\mathbf{S}, \text{ for } k = N-1 \\ (\mathbf{S}^L + X_{N-1}.\mathbf{S}^R + V_S).X_k = \mathbf{S}^L{\uparrow}k + X_{N-1}.(\mathbf{S}^R{\uparrow}k) + V_S.X_k, \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{for } k < N \end{cases}
\end{aligned}
$$

Differential operations introduce partial differentiation and integration in the cotensor algebra. These differentiation and integration operators respectively refer to $\mathbf{S}[\mathbf{.}]$ and $\mathbf{S} \uparrow \mathbf{.}$. They also use the cotensor of integration/derivation factors "Δ_k", where the o_i are the variable occurence number, such that:

$$
\begin{aligned}
&(\Delta_k)_{(o_0,\ldots,o_{N-1})} && \triangleq 1 + o_k, \text{ for } \textstyle\sum_i o_i = R \\
&\tfrac{d\mathbf{S}(X_0,\ldots,X_{N-1})}{dX_k} && \triangleq \mathbf{S}[k] \odot \Delta_k \\
&\textstyle\int_0^{X_k} \mathbf{S}(X_0,\ldots,x_k,\ldots,X_{N-1})dx_k && \triangleq (\mathbf{S} \odot \Delta_k^{-1}){\uparrow}k
\end{aligned}
$$

Figure 1 illustrates these operations on a cotensor of dimension $N = 4$, for $k = 2$. The `set` operation removes the red sub-trees and the blue edges and merges nodes at extremity of blue edges. The `lift` operation proceeds the other way by inserting blue edges and creating zero filled red sub-trees.

It yields the following implementation, where `set` and `lift` respectively denote "$\mathbf{.}[\mathbf{.}]$" and "$\mathbf{.} \uparrow \mathbf{.}$" and where `left` and `right` respectively return left and right sub-trees :

```
let rec set : type n d k. (d, k, n) Nat.add -> ('a, n Nat.succ) tree -> 'a ->
 ('a, n Nat.succ) tree =
  fun pr st zero ->
   lazy (
    match pr, (Lazy.force st) with
    | _           , Leaf              -> Leaf
    | Nat.Zadd    , Node (stl, v, str) -> Lazy.force str
    | Nat.Sadd pr', Node (stl, v, str) ->
     let set_stl = set pr' stl zero in
     let root = match Lazy.force set_stl with
      | Leaf           -> zero
      | Node(_, vl, _) -> vl in
     Node (set_stl, root, set pr str zero)
    )

let rec lift : type n d k. 'a -> n Nat.succ Nat.isnat -> k Nat.isnat ->
 (d, k, n) Nat.add -> ('a, n Nat.succ) tree -> ('a, n Nat.succ) tree =
  fun zero n k pr st ->
   lazy (
```

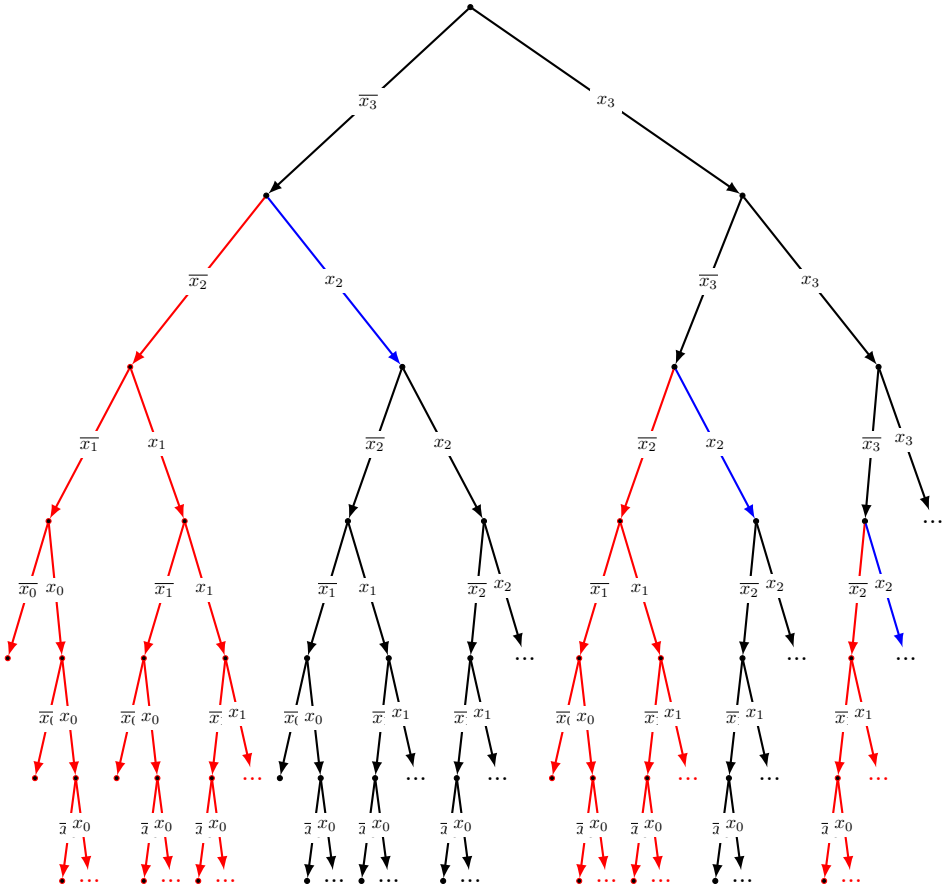


Figure 1: Illustration of set and lift operations.

```
  match pr with
  | Nat.Zadd      -> (match n with
   | (Nat.S Nat.Z)     -> Node(nil    , zero, st)
   | (Nat.S (Nat.S _)) -> Node(feuille, zero, st))
  | Nat.Sadd pr' -> let (Nat.S n') = n in
      Node (lift zero n' k pr' (left n st), zero, lift zero n k pr (right st))
 )
```

4.4 Differential equations

Within our framework, we are capable of defining the solution to a differential equation, univariate or multivariate, as a simple recursive value of the implementation language OCaml(or mutually recursive values, in case of a system of equations). We assume that the solution is analytical around point **0** and don't try to check whether this hypothesis holds.

Nevertheless, the differential equation has first to be homogeneous, i.e. only refers to the solution function at a single point of evaluation $u(x)$. Second, it also has to be put in solved form, i.e. under the form $u(x) = expr$ where $expr$ is any expression of our algebra involving (or not) the unknown $u(x)$, used through arithmetical and differential operators. Furthermore, it must also be *causal* for the coefficients of the infinite Taylor series to be well defined. By causal, we mean that the computation of any coefficient won't diverge, i.e. won't involve in turn the computation of some other coefficient lying deeper in the tree structure and so on.

In the univariate case, a simple syntactic criterion can be tested: an expression $expr$ occurring in the differential equation $u(x) = expr$ is causal whenever in each branch of the syntax tree of $expr$ that goes from the root down to the unknown u, the number of integration operators traversed is strictly greater than the number of derivation operators. The justification is the following: integration shifts the coefficients series right (inserting a 0 in head position) whereas the derivation shifts it left. So the dependency between coefficients is guaranteed to have finite depth, i.e. coefficients $\frac{d^i u}{dx^i}$ can only depend upon $\frac{d^j u}{dx^j}$ for $j < i$, when the criterion holds. The following definition computes the minimal shift δ performed by any branch of an expression $expr$:

$$\begin{array}{ll}
\delta(x) & = \delta(k) = -\infty \\
\delta(u(x)) & = 0 \\
\delta(f+g) & = \delta(f*g) = \max(\delta(f), \delta(g)) \\
\delta(\cos f) = \delta(\sin f) = \ldots & = \delta(f) \\
\delta(\frac{df}{dx}) & = \delta(f) + 1 \\
\delta(\int^X f) & = \delta(f) - 1
\end{array}$$

Causality is easily implied by the property $\delta(expr) < 0$, as a derivative at order α would then depend at most on derivatives at order $0, \ldots, \alpha + \delta$.

This criterion is very relaxed when compared to the usual syntactic restrictions imposed on differential equations, such as the *Initial Value Problem*

pattern: $u(x) = u_0(x) + \int^x expr(u(x), x)$ where $expr$ cannot differentiate u. This pattern is used in various tools promoting certified integration, such as DynIbex[4] or Flow*[3] and even appear in theorems that prove existence of solutions, such as the Picard-Lindelöf theorem.

In the multivariate case though, no such simple syntactic criterion seems to apply that would not rule out most interesting differential equations, such as the heat equation or the wave equation. More investigation is still needed in that respect. We discuss this topic further in section 6.

5 The composition operator

5.1 Differential method

5.1.1 Principle

The Taylor series algebra with the previous operations still remains basic, and that is why we are now interested in composing Taylor series with elementary functions. To do so, we only need to apply elementary functions to arbitrary arguments, *i.e.* to compose univariate Taylor series with multivariate ones. A general composition scheme of Taylor series is also possible in our setting but out of the scope of our current concerns. This method lies on a differential decomposition, namely a function is the sum of the integrals of its derivatives with respect to all its variables, plus a constant term :

$$H : \mathbb{R}^N \to \mathbb{R}, \qquad H = H(0) + \sum_{i<N} \int^{X_i} \frac{\partial H}{\partial X_i}\Big|_{\substack{X_k=0 \\ k>i}} dX_i$$

5.1.2 Example

We need to partially evaluate the derivatives at $\mathbf{0}$ to avoid counting several times the parts shared by different variables, as illustrates the following concrete example :
let $F : \mathbb{R}^3 \to \mathbb{R}, \qquad F(x, y, z) = x^3 + 2x^2y + xz + 5y^2 + 3yz^2$

$$\begin{cases} \frac{\partial f}{\partial x} = 3x^2 + 4xy + z & \int_0^x \frac{\partial f}{\partial x} dx = x^3 + 2x^2y + xz \\ \frac{\partial f}{\partial y} = 2x^2 + 10y + 3z^2 & \int_0^y \frac{\partial f}{\partial y} dy = 2x^2y + 5y^2 + 3yz^2 \\ \frac{\partial f}{\partial z} = x + 6yz & \int_0^z \frac{\partial f}{\partial z} dz = xz + 3yz^2 \end{cases}$$

The blue terms are redundant and that is why we have :

$$F(x, y, z) = F(0, 0, 0) + \int_0^x \frac{\partial f}{\partial x} dx + \int_0^y \frac{\partial f}{\partial y}\Big|_{x=0} dy + \int_0^z \frac{\partial f}{\partial z}\Big|_{\substack{x=0 \\ y=0}} dz$$

5.1.3 Composition

As we are in the specific case of composition, we will use the classic chain rule:

$$\frac{\partial(f \circ g)}{\partial X_i}_{i<N} = (\frac{\partial g}{\partial X_i})_{i<N} \times (f' \circ g)$$

Hence :

$$f \circ g = f \circ g(0) + \sum_{i<N} \int^{X_i} (\frac{\partial g}{\partial X_i} \times f' \circ g)\Big|_{\substack{X_k=0 \\ k>i}} dX_i$$

The computation of the partial derivatives $\frac{\partial(f \circ g)}{\partial X_i}_{i<N}$ is done case by case with respect to the elementary function f at use, each such function having a well-known derivative f'. The cases where $f = \exp, \sin, \cos, \log, \text{atan}, x^a, \ldots$ are easily handled. So, according to the above equation, we only need to partially evaluate these derivatives, to integrate them then and to finally sum the results.

This method will bring us satisfying results as detailed below, but one must bear in mind that despite the method is very short in terms of code and then easily implemented, it is not optimal in terms of computation. This differential method for the composition is not canonical in that it does not compute the minimum number of operations to produce the coefficients of the result. As a witness of non canonicity in the definition of composition, the Δ_k coefficients will be used for multiplication and division consecutively, which could be avoided. Besides, as long as we do not handle certified errors, the method does not need an additive decomposition of f, i.e. an expression of $f(x+y)$ in terms of $f(x)$ and $f(y)$ alone. But it will be the case as soon as we handle the errors and we will have to deal with this requirement. Fortunately, such decompositions are known for all elementary functions we intend to use.

5.2 Elementary functions

Elementary functions, limited to one argument functions, are specified as univariate Taylor series. Therefore, as only one branch of the cotensor will be meaningful, such series are treated separately. This is only a matter of efficiency and obviously not mandatory. To obtain a Taylor expansion of an elementary function, we need to be able to compute any n-th derivative. Taylor series for elementary functions are well known, so the first way to produce such a series is to compute the coefficients iteratively and lazily with respect to the known formulas, such as the following ones :

$$\begin{aligned}
\exp(x) &= \textstyle\sum_{i\in\mathbb{N}} \frac{x^i}{i!} \\
\log(1+x) &= \textstyle\sum_{i\in\mathbb{N}} \frac{-(-x)^i}{i} \\
(1+x)^p &= \textstyle\sum_{i\in\mathbb{N}} \binom{p}{i} x^i \\
\sin(x) &= \textstyle\sum_{i\in\mathbb{N}} \frac{(-1)^i}{(2i+1)!} x^{2i+1} \\
\cos(x) &= \textstyle\sum_{i\in\mathbb{N}} \frac{(-1)^i}{(2i)!} x^{2i}
\end{aligned}$$

Similar formulations are available for elementary functions not presented here.

6 Experimentations

Now that the main operations are available in our algebra, we can start using it. Differential equations are pervasive in dynamical systems and our point is to propose a direct (*i.e.* non-iterative) way to solve them. By direct method, we mean that coefficients are computed once and for all and therefore there is no need to iterate over their values until a specific precision is reached. Precision in our case is seen differently: coefficients are computed only once and if the user wants a finer precision, the user will increase the order of derivation which means that new and deeper coefficients will be computed.

6.1 Airy equation

To illustrate this direct approach for solving ODEs and PDEs, we will use the first dimension Airy equation which stands as follows :

$$f'' - xf = 0$$

As the equation contains a second derivative, we split it for convenience in two first order equations introducing f_dot as f derivative :

$$\begin{cases} f\_dot = f\_dot_0 + \int^x xf \\ f = f_0 + \int^x f\_dot \end{cases}$$

```
let (airy0, airy0') = (0.35, -0.26)

let rec f = (lazy (Lazy.force (somme (IST.constant airy0)
                                     (pinteg var_x fdot))))
and fdot  = (lazy (Lazy.force (somme (IST.constant airy0')
                                     (pinteg var_x (product x f)))))
```

Then, thanks to OCaml laziness, we express and solve this mutually recursive system directly, with the following principle :

- According to the second equation, computing the first coefficient of f, the constant part, means summing the constant part of f_0 with the constant part of $\int_x f\_dot$. We know that the constant part of an integral will be 0, whatever the integrand is.
- the first coefficient of f_dot, or equivalently the second coefficient of f, is computed the same way (no need to evaluate the argument of the integral).
- then the mutual recursion works and the third coefficient of f, or the second one of f_dot, is simply the result of integrating the constant part of xf, actually 0. The other coefficients are also computed in finite time.

So the trick is to stay a step ahead by computing a first coefficient of a recursive Taylor series without having to evaluate itself, thanks to the integral operator, and then to keep this advance all along the computation so that the recursion will always end. Indeed, if the computation scheme respects the causality, for example in one dimension : computing a coefficient requires only strictly lower order coefficients, then we can ensure the recursion will end.

Once we get the solution up to a specific order, we evaluate it as a polynomial function so that we can draw its graph:

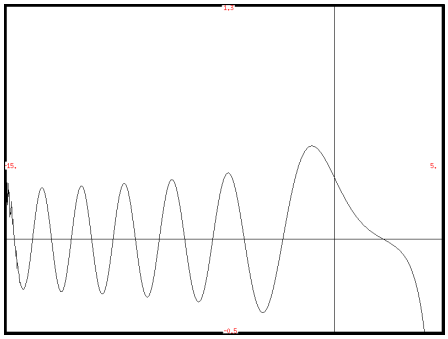


Figure 2: our function (at order 150)

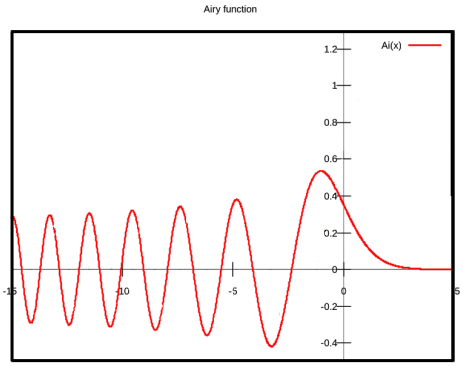


Figure 3: theoretical result

We can observe that the approximation is reliable on a specific interval and diverges outside of it. We can have this conclusion because we know the theoretical result in this case, but we won't know it in most cases. This is what will motivate the necessary handling of certified errors. Intervals of errors, which are only an example of error representation, will give the user information about how far the theoretical function could be from the returned approximation.

6.2 Heat equation

In order to explain the principle of causality more precisely and to show a more general case, we are going to present the 2-dimensional heat equation example:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

```
let u0 = Iter_Elemt.sinus

let rec u = Iter_IST.(lazy (Lazy.force (somme u0 (product alpha
                                                  (pinteg var_t u_aux)))))
and u_aux = Iter_IST.(lazy (Lazy.force (pdiff var_x (pdiff var_x u))))
```

There are 2 different ways of integrating this equation and we chose to integrate it with respect to variable t so that initial conditions are a function of

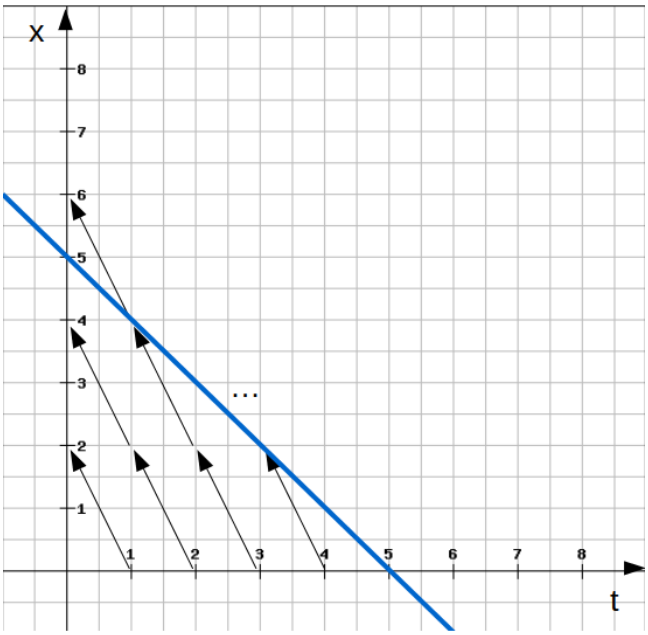
variable x at initial time $t = 0$. Here is the new form of the equation :

$$u(x,t) = u_0(x) + \alpha \times \int^t \frac{\partial^2 u(x,t)}{\partial x^2}$$

where $u_0(x)$ will be a data we have. The causality is respected if computing any derivative $\frac{\partial^{i+j} u}{\partial x^i \partial t^j}$ boils down to computing elements of initial condition $u_0(x)$. And in the case of the heat equation, we can ensure it will be possible thanks to *Schwarz*'s theorem about switching partial derivatives :

$$\frac{\partial^{i+j} u}{\partial x^i \partial t^j} = \frac{\partial^{i+j-1} u}{\partial x^i \partial t^{j-1}} \left(\frac{\partial u}{\partial t} \right) = \frac{\partial^{i+j-1} u}{\partial x^i \partial t^{j-1}} \left(\frac{\partial^2 u}{\partial x^2} \right) = \frac{\partial^{i+j+1} u}{\partial x^{i+2} \partial t^{j-1}} = ... = \frac{\partial^{i+2j} u}{\partial x^{i+2j}}$$

This graph illustrates the dependencies between the partial derivatives and



we see that all arrows will end up on the vertical axis which represents the derivatives with respect to x only, namely the different parts of $u_0(x)$. The causality being respected ensures that the recursion will end. This example in 2 dimensions shows how the principle of causality is more flexible than it was presented with the Airy equation. Indeed, we said that coefficients of specific order should require strictly lower order coefficients, which is graphically represented by arrows crossing the blue line from the top right-hand corner down to bottom left-hand corner. But we state now that it is not a necessary condition as we can see with the heat equation where higher order coefficients are required but with respect to other variables. So arrows are allowed to cross the blue line in the opposite direction as long as they end on the vertical axis.

Figure 4 shows our heat equation solution developed at order 25. The vertical axis is the temperature. We set the initial conditions to a sinus, which concretely means we impose the temperature on one axis to be an alternation of warm and cold at initial time. The graph converges to a uniform average value along the time which is consistent with the physical interpretation.

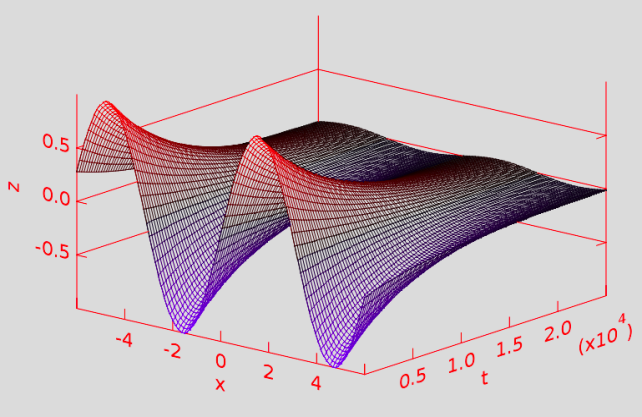


Figure 4: Heat equation solution

What we call order here and denote by R is only the unrolling depth of the infinite tree we build. The graph in figure 5 shows the computation times (in seconds, on a common laptop computer) of the heat equation solution according to order and the graph in figure 6 shows this computation time divided by the number of coefficients of the solution, which lies in $\theta(R^N)$ with N the dimension, according to [13]. By dividing the computation time by the number of

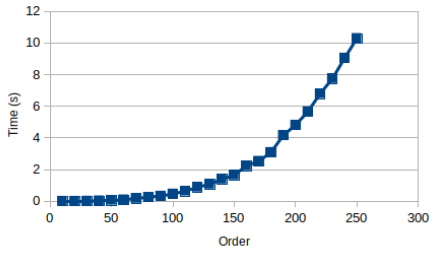


Figure 5: Computation time

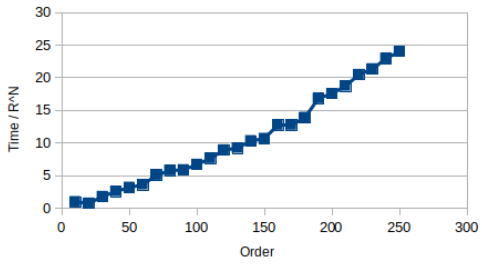


Figure 6: Computation time/R^2

computed coefficients (normalized to 1 for $R = 0$), we aimed at evaluating the amount of additional computation done per useful coefficient, *i.e.* the "administrative" overhead induced by the resolution of the equation, due to auxiliary data structures, memory allocations, etc. We observe only a linear overhead and despite the relative simplicity of the heat equation, it comforts us in the decisions taken so far for implementing our framework.

7 Certified errors

We will now discuss finite Taylor expansions, i.e. considering remainders and errors, not departing away from infinite Taylor series, as we use the same data-structure. Also, many operations from previous sections which were coefficients agnostic can be reused straightforwardly without any modification.

7.1 A simple error model

Differential equations put aside, we are already able to compute certified errors in our framework. It merely requires the introduction of an arithmetical domain for errors. We introduce below a very simple error domain based upon symmetric zero-centered monotonic error functions. According to equation 1, where the only points of evaluation are $\mathbf{0}$ and $\lambda * \mathbf{x}$, we wish to represent: first, any derivative at point $\mathbf{0}$ as the main value; second, (an over-approximation of) the difference between evaluating a derivative at point $\mathbf{0}$ and evaluating it at unknown point $\lambda * \mathbf{x}$, as the error part.

Let us assume $\mathbb{K}$ stands for the value domain. Error functions are then elements of the following domain $\mathbb{E}$, assuming we work in dimension N:

$$\mathbb{E} \triangleq \{f \in (\mathbb{K}^+)^N \to \mathbb{K}^+ \ \mid \ f(\mathbf{0}) = 0,\ f \text{ monotonic}\}$$

The error model is then the product $\mathbb{K} \times \mathbb{E}$. The semantics $[\![\,.\,]\!]$ of an element of this model represents a function from variable bounds to sets of possible values:

$$[\![\langle v, \epsilon\rangle]\!] \triangleq \mathbf{X} \in (\mathbb{K}^+)^N \mapsto \{k \in \mathbb{K} \mid |k - v| \leq \epsilon(\mathbf{X})\}$$

The error model has $N + 1$ constructors: $(k, \mathbf{0})$ for $k \in \mathbb{K}$, denoted "k" and the $i \in [0, N-1]$ indexed family $(0, \mathbf{X} \mapsto \mathbf{X}_i)$, denoted "$\mathbf{X}_i$". It is endowed with a $\mathbb{K}$-algebra structure and is further turned into an full-fledged domain using suitable definitions of elementary functions on $\mathbb{K} \times \mathbb{E}$, as illustrated below. Similar definitions may be devised for other elementary functions:

$$\begin{array}{ll}
\langle v_1, \epsilon_1\rangle + \langle v_2, \epsilon_2\rangle & \triangleq \langle v_1 + v_2, \epsilon_1 + \epsilon_2\rangle \\
\alpha \times \langle v, \epsilon\rangle & \triangleq \langle \alpha \times v, |\alpha| \times \epsilon\rangle \\
\langle v_1, \epsilon_1\rangle \times \langle v_2, \epsilon_2\rangle & \triangleq \langle v_1 \times v_2, |v_1| \times \epsilon_2 + |v_2| \times \epsilon_1 + \epsilon_1 \times \epsilon_2\rangle \\
e^{\langle v, \epsilon\rangle} & \triangleq \langle e^v, e^v \times (e^\epsilon - 1)\rangle \\
\log\langle v, \epsilon\rangle & \triangleq \langle \log v, \log\big(1 + \frac{\epsilon}{v}\big)\rangle \quad (v \neq 0) \\
\sin\langle v, \epsilon\rangle & \triangleq \langle \sin v, |\sin v| \times (1 - \cos\epsilon) + |\cos v| \times |\sin\epsilon|\rangle \\
\cos\langle v, \epsilon\rangle & \triangleq \langle \cos v, |\cos v| \times (1 - \cos\epsilon) + |\sin v| \times |\sin\epsilon|\rangle \\
\sinh(v, \epsilon) & \triangleq (\sinh v, |\sinh v| \times (\cosh\epsilon - 1) + |\cosh v| \times |\sinh\epsilon|) \\
\cosh(v, \epsilon) & \triangleq (\cosh v, |\cosh v| \times (\cosh\epsilon - 1) + |\sinh v| \times |\sinh\epsilon|)
\end{array}$$

Finally, all the useful information about the derivative at multi-index α of a multivariate function f will be represented as the following pair:

$$\langle v_\alpha^f, \epsilon_\alpha^f(X)\rangle \quad \text{where} \quad \left\{ \begin{array}{ll} v_\alpha^f & = \frac{1}{\alpha!} \cdot \frac{\partial^{|\alpha|} f}{\partial X^\alpha}(\mathbf{0}) \\ \epsilon_\alpha^f(X) & \geq \frac{1}{\alpha!} \cdot \max\limits_{\lambda \in [0,1]} \left| \frac{\partial^{|\alpha|} f}{\partial X^\alpha}(\lambda * X) - \frac{\partial^{|\alpha|} f}{\partial X^\alpha}(\mathbf{0}) \right| \end{array} \right.$$

This error model lets us compute approximations of polynomials with error bounds as illustrates the example in figure 7.

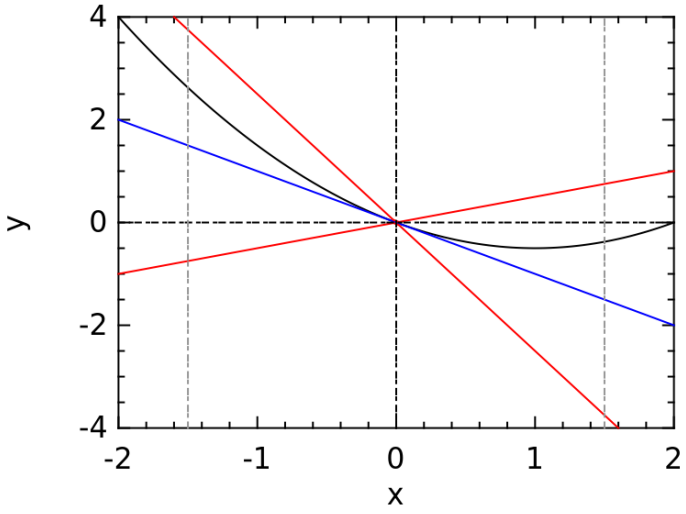


Figure 7: polynomial approximation

The **black** graph is the polynomial $P = \frac{1}{2}X^2 - X$.
The **blue** graph is the approximation of P at order 1.
The **red** graphs are the error bounds of the approximation on interval $[-1.5, 1.5]$.

7.2 Taylor models

Taylor models are then built from cotensors of $\langle value, error \rangle$ terms. We consider a function $f \in \mathbb{R}^N \to \mathbb{R}$, assumed analytical at point $\mathbf{0}$ and note respectively f_α and ϵ_α as the value and error at derivation multi-index α.

A Taylor model predicate $\mathcal{TM}(f, R)$ at order R in a neighbourhood of point $\mathbf{0}$ is defined as the following, obtained from equation 1:

$$\mathcal{TM}(f,R) \triangleq \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^N . \mathbf{x} \leq \mathbf{y} \implies |f(\mathbf{x}) - \sum_{\alpha=0}^{|\alpha| \leq R} f_\alpha \mathbf{x}^\alpha| \leq \sum_{|\alpha|=R} \epsilon_\alpha(\mathbf{y}) |\mathbf{x}|^\alpha$$

A Taylor model for parameters R is then the set of functions f such that $\mathcal{TM}(f, R)$ holds true. In this formulation, the interplay between the two variables $\mathbf{x}$ and $\mathbf{y}$ enables to evaluate only once any error function on the bounds $\mathbf{y}$

of the variation domain for the variable $\mathbf{x}$, since error functions are monotonic. Our goal is to allow avoiding computations of error functions at each point $\mathbf{x}$, while still allowing for some precision around $\mathbf{0}$, since the right-hand side of a Taylor model inequality is a $\mathbf{x}$-polynomial form that tends to $\mathbf{0}$ when $\mathbf{x} \to \mathbf{0}$.

7.3 Computing Taylor series with errors

The arithmetical operators of Taylor series, as seen in section 4, only involves arithmetical operators between coefficients. We demonstrated above that our error model is indeed endowed with such arithmetical operators. Therefore, the last ingredients to our error model needed to handle differential equations are the differential operators.

The case for partial derivatives is transparent and handled similarly as in section 4.3, as it only involves multiplying by Δ_k and shifting coefficients in the infinite tree structure.

Finally, to address the integration operator, we rely on the following generalized Leibniz's rule:

$$\frac{\partial}{\partial X^i}\int_0^{X_j} f = \int_0^{X_j} \frac{\partial f}{\partial X^i}\ (i \neq j) \qquad\qquad \frac{\partial}{\partial X^i}\int_0^{X_i} f = f$$

This rule entails that, as for values only, the infinite tree of $\int_0^{X_i} f$ is built by filling nodes with 0 (the value of the integral when $X_i = 0$), from the root down until the variable X_i occurs. Then, from that point downward, the remaining values are built from the ones taken from f at the same position but ignoring the first occurrence of X_i.

The situation is more complex as regards errors. We have the following, for α a derivation multi-index not containing X_i:

$$\begin{aligned}
& \tfrac{1}{\alpha!}\cdot \mid \tfrac{\partial^{|\alpha|}}{\partial X^\alpha}(\int_0^{X_i} f)(\lambda * X) - \tfrac{\partial^{|\alpha|}}{\partial X^\alpha}(\int_0^{X_i} f)(\mathbf{0}) \mid \\
= \ & \tfrac{1}{\alpha!}\cdot \mid \int_0^{X_i} (\tfrac{\partial^{|\alpha|} f}{\partial X^\alpha})(\lambda * X) - \int_0^{X_i} (\tfrac{\partial^{|\alpha|} f}{\partial X^\alpha})(\mathbf{0}) \mid \\
= \ & \tfrac{1}{\alpha!}\cdot \mid \int_0^{X_i} (\tfrac{\partial^{|\alpha|} f}{\partial X^\alpha})(\lambda * X) \mid \\
\leq \ & \tfrac{1}{\alpha!}\cdot \max_{\lambda\in[0,1]} \mid \tfrac{\partial^{|\alpha|} f}{\partial X^\alpha}(\lambda * X) \mid * \mid X_i \mid \\
\leq \ & (\mid v_\alpha^f \mid + \epsilon_\alpha^f(X)) * \mid X_i \mid
\end{aligned}$$

Then, denoting $\epsilon'_\alpha(X)$ the error function of the α derivative of $\int^{X_i} f$, we get:

$$\epsilon'_\alpha(X) = \frac{1}{\alpha!}\cdot \max_{\lambda\in[0,1]} \mid \frac{\partial^{|\alpha|}}{\partial X^\alpha}(\int_0^{X_i} f)(\lambda * X) - \frac{\partial^{|\alpha|}}{\partial X^\alpha}(\int_0^{X_i} f)(\mathbf{0}) \mid \leq (\mid v_\alpha^f \mid + \epsilon_\alpha^f(X)) * \mid X_i \mid$$

A difference finally appears in the treatement of values versus errors. Values of $\int^{X_i} f$ are either 0 or depend upon values of f strictly less deep in the tree structure. Error terms are not strictly causal as they may depend upon values and errors of f at the exact same position. This will raise issues when considering differential equations, as shown in section 7.4.

It yields the following implementation of the `lift` operator :

```
let rec lazy_absorb : type n. R.Err.t -> n Nat.isnat -> (R.t, n) tree ->
(R.t, n) tree =
 fun err n st ->
  lazy (match n with
   | Nat.Z    -> Nil
   | Nat.S n' -> Node (lazy_absorb err n' (left n st), R.cons (value_0 st) err,
                       lazy_absorb err n (right st)))

let lift : 'k Nat.isnat -> ('d, 'k, N.t') Nat.add ->
           ('a, N.t) tree -> ('a, N.t) tree =
 fun k pr st ->
  let err_k = R.Err.var k (Nat.Sadd pr) in
  let rec aux : type n d k. n Nat.isnat -> (d, k, n) Nat.add ->
                            ('a, n Nat.succ) tree -> ('a, n Nat.succ) tree =
  fun n pr st ->
   lazy (match n with
    | Nat.Z    -> Node (nil, R.cons (value_0 st) err_k, st)
    | Nat.S n' ->
      match pr with
      | Nat.Zadd     -> Node (lazy_absorb err_k n (left (Nat.S n) st),
                              R.cons (value_0 st) err_k,
                              st)
      | Nat.Sadd pr' -> Node (aux n' pr' (left (Nat.S n) st),
                              R.cons (value_0 st) err_k,
                              aux n pr (right st)))
  in aux (match N.isnat with Nat.S n' -> n') pr st
```

where `err_k` is the function $X \mapsto |X_i|$ and `R.cons (value_0 st) err_k` represents the delayed definition of the function $X \mapsto (\mid v_\alpha^f \mid +\epsilon_\alpha^f(X))* \mid X_i \mid$. This technical subtlety is necessary to ensure that recursive definitions of error functions, which naturally appear in differential equation solving, won't diverge at definition time (divergence at evaluation time is treated in section 7.5). Primitives `left`, `right` and `value_0` are all lazy accessors to respectively left sub-tree, right sub-tree and value of their argument. Apart from this extra laziness, the functions `lazy_absorb` and `lift` both amount to applying `R.cons` uniformly through the tree. They are very close in nature to the original `lift` function from section 4.

It is then possible to compute integration of Taylor series in dimension 1 with bounding errors as shows the example in figure 8.

We notice that the approximation looks like a polynomial of order 3 even if it has been computed until order 4 because there are no coefficients of order 4. We mention the fact that the errors computed with integration can be explained

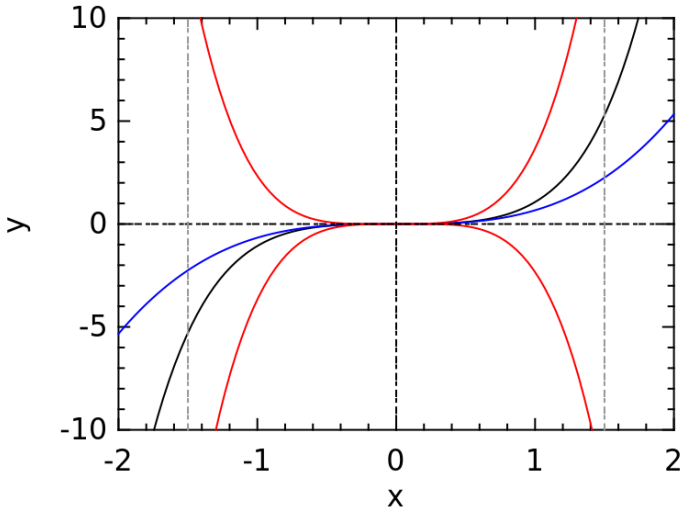


Figure 8: integration with errors

The **black** graph is the integral of $2(X^4 + X^2)$, which is $\frac{2}{5}X^5 + \frac{2}{3}X^3$.
The **blue** graph is the approximation of this integral at order 4.
The **red** graphs are the error bounds of the approximation on interval $[-1.5, 1.5]$.

by the over-approximation computed with the model presented above but can also be due to the zero-centered error model which has the inconvenient of not being very precise.

7.4 Issues with recursive definitions

As already stated, dependencies between errors at different derivation orders do not respect the causality relation fulfilled by pure values. We illustrate this discrepancy between values and errors, considering the following partial development of a Taylor series with errors for a bivariate function f:

$$f(X, Y) \triangleq \langle f_0, \epsilon_0 \rangle + X.\langle f_X, \epsilon_X \rangle + Y.\langle f_Y, \epsilon_Y \rangle + \dots$$

Then, integrating f along X, accounting for errors, yields the following series:

$$\int^X f = \langle 0, |X|.(|f_0| + \epsilon_0) \rangle + X.\langle f_0, \epsilon_0 \rangle + Y.\langle 0, |X|.(|f_Y| + \epsilon_Y) \rangle + \dots$$

Unfortunately, we remark that the error term $|X|.(|f_0|+\epsilon_0)$ at order 0, while still a zero-centered monotonic error function, directly depends on ϵ_0, the error

function of f at order 0. The same problem occurs at order Y. On the contrary, the value part of the integrand is always 0, so is independent of f. As we wish to define f recursively through such an integrand, setting for instance $f = \int^X f$, we face the necessity to find a different computation scheme for errors than for values.

7.5 Computing errors in the univariate case

As errors do not behave as values, we need first to reconsider the definition of causality given in 4.4. The change comes from the integration operator, where the error term at order $\alpha = 0$ of $\int^X f$ depends upon the error term of f at the same order. Therefore, for a (recursive) differential equation, whereas the value at order α depends only upon values at order strictly less than α, errors obey a looser scheme.

Still, any error (and value) depends upon a finite number of other terms. Indeed, let us assume an equation in solved form $u(x) = expr$ and consider any path in $expr$ leading to $u(x)$. Since $expr$ is causal, there is strictly more integration operators than derivation operators in that path, that is a strictly positive number of integrators I. Let us define M as the maximum of this quantity I on every such path in $expr$. Then, the interval $[0, M]$ is an upper bound on the set of derivation orders of the solution $u(x)$ that could be mutually dependent (as regards errors). Above that value, all remaining derivation orders of $u(x)$ strictly depend upon less deep (and previously computed) coefficients.

Even if the value of M may be approximated through a static analysis, this is left for future work as we choose to ask the user to provide this value to the fixpoint engine that will compute errors. The principle is quite simple: the first M errors will be treated as a vector of functions and we compute a fixpoint on that vector as a whole, while letting the values being computed as before. For other errors, we let the original scheme untouched since errors will then behave as causally as values. Therefore, the conception of the fixpoint engine only requires a slight adaptation to the original scheme which was trivial since it consisted in the straightforward definition of a recursive value in OCaml.

Concretely, we need a way to separate and handle the M first errors, M being user-provided, usually a small integer value. To that end, we use the primitive continuations defined in [8] as a clean solution. We briefly recall here the principles at work: a different return type `result` distinguishes computations of error functions that return normally with a value – `Done r` where `r` is the result – and interruptions that return a resumable computation instead – `Request (i, cont)` where `i` is the derivation order that was called and interrupted and `cont` is the continuation that waits for a value in order to resume.

```
type result = Done of R.t | Request of (int * (R.t -> result))
```

The first case is reserved for errors at order above M whereas the second one allows to adapt the scheme for the M first errors. Indeed, instead of calling an error function below order M and waiting for it to terminate, which it may

not do, we interrupt it and then we have to guess a proposed value for the fixpoint. If the guess is wrong, i.e. doesn't respect the required inequalities, we iterate until a correct guess is made that constitutes the return value of the error function for that call.

According to these principles, we first `untie` the first error functions, replacing them by interruptions. In formal terms, using the delimited continuations primitives `shift` and `reset`:

$$\texttt{untie}\ (k, \sum_{i\in\mathbb{N}} \langle v_i, \epsilon_i \rangle.X^i) = \sum_{i\in[0,k]} \langle v_i, \delta_i \rangle.X^i + \sum_{i>k} \langle v_i, \epsilon_i \rangle.X^i$$
$$\text{where}\quad \delta_i(x) = \texttt{shift (fun k -> Request (i, k))}$$

Then, we `tie` error functions back by guessing a correct fixpoint value. It amounts to finding a vector of values e of dimension M such that $e_i \geq expr[e_i/\epsilon_i(x)]$:

$$\texttt{tie}\ (k, \sum_{i\in\mathbb{N}} \langle v_i, \epsilon_i \rangle.X^i) = \sum_{i\in[0,k]} \langle v_i, \delta_i \rangle.X^i + \sum_{i>k} \langle v_i, \epsilon_i \rangle.X^i$$
$$\text{where}\quad \delta_i(x) \geq \texttt{match reset}\,\epsilon_i(x)\,\texttt{with Done r -> r | Request (j, k) -> k}\,\delta_j(x)$$

That is, the guess made for the solution $u(x)$ should be coarser than the one computed through $expr$, or equivalently said, $expr$ should be contractive with respect to errors. This simply transliterates the argument given in [2], which in turn stems from Schauder's fixed point theorem.

The complete OCaml code is not given here for the sake of simplicity as it also includes: heuristics for making a small enough correct guess; memoization of previously made guesses; and an additional loop around the `match ... with` construct, as evaluation of an error function ϵ_i may involve several interruptions made with `shift`. Globally, the `fixpoint` engine is built by composing `tie` and `untie` around `expr`, with `k`=M.

```
let fixpoint :
  'k Nat.isnat -> ((R.t, N.t) tree -> (R.t, N.t) tree) -> (R.t, N.t) tree =
  fun k expr ->
  let rec fix = lazy (Lazy.force (tie k (expr (untie k fix)))) in
  fix
```

8 Application to the Airy equation

8.1 Different orders

We experiment our computation scheme on the problem of finding a certified approximation to the Airy ODE, introduced in 6.1.

First, we claim that, by straightforwardly applying the various schemes dedicated to error computation and especially the integration operator, only the error function ϵ_0 (at order 0) depends upon itself in the Airy equation. Every other error function at higher orders can be defined recursively from ϵ_0.

Thus, it allows us to determine the minimal value $k = 1$ for the number of error functions that could depend upon each other. We now can make an

informed guess for the `k` parameter of the `fixpoint` function. As a matter of fact, providing a greater value for k can only produce coarser approximations, yet without compromising our method. On the contrary, providing a smaller value (here only $k = 0$ is possible) would result in a clear runtime error, showing that the error function ϵ_0 is ill defined.

Figure 9 shows our results about approximating the Airy ODE at different orders. Our approximations are quite cheap as they only require the computation of exactly $n + 1$ values and at most $n + 1$ error bounds at order n for each choice of interval of certified integration. This amounts, at order 15, the highest order we tested, to computing only 32 floating point values for the whole chosen interval. Note that our choice of Taylor model produces approximations not in the classical form of a tube surrounding the solution, but as a tube "pinched" at point $\mathbf{0}$, where the error is known to be zero.

As regards precision, since our error model is quite coarse, as shown in section 7.1, we are not yet capable of comparing favorably with mature tools such as Dynibex [4] which can integrate ODE for long time periods while keeping a very high precision. To mitigate this raw fact, the amount of floating point numbers computed by these tools is incomparably larger, as they tend to split the interval of integration in tiny time slices (typically 0.01 seconds) and then apply in each such slice certified sophisticated Runge-Kutta integration schemes. The maximum error spans, at successive orders 1, 5, 10 and 15, computed on the bounds of the integration interval, are approximately 0.322, 0.16, 0.1 and 0.016, which represents at most a deviation of 86.6%, 41.3%, 24.6% and 1.1% from the function we want to approximate on the whole interval.

9 Perspectives

9.1 Canonical method for composition

As defined in section 5, composition involves the resolution of a partial differential equation. This hinders the computation of error bounds. Indeed, as far as we know, there is no established general method to solve such equations with certified errors, beyond ad-hoc situations such as elliptic, parabolic, etc, equations with specific initial conditions.

In order to devise a direct more tractable and non recursive way to compose Taylor series, following schemes such as Faà di Bruno's formula, we first need to handle errors. As in formal power series, composition $(f \circ g)$ may be achieved only when g has no constant part. To factorize out the constant part of g (so that we fall back to evaluation at point $\mathbf{0}$), we depend on an additive decomposition of f, when available.

Again, we sum up some decompositions of standard elementary functions. For every $A_{n+1} \in R_{n+1}$, we have the following equations, where we remark that their right-hand sides are built from a constant part (V_A) and another term without a constant part:

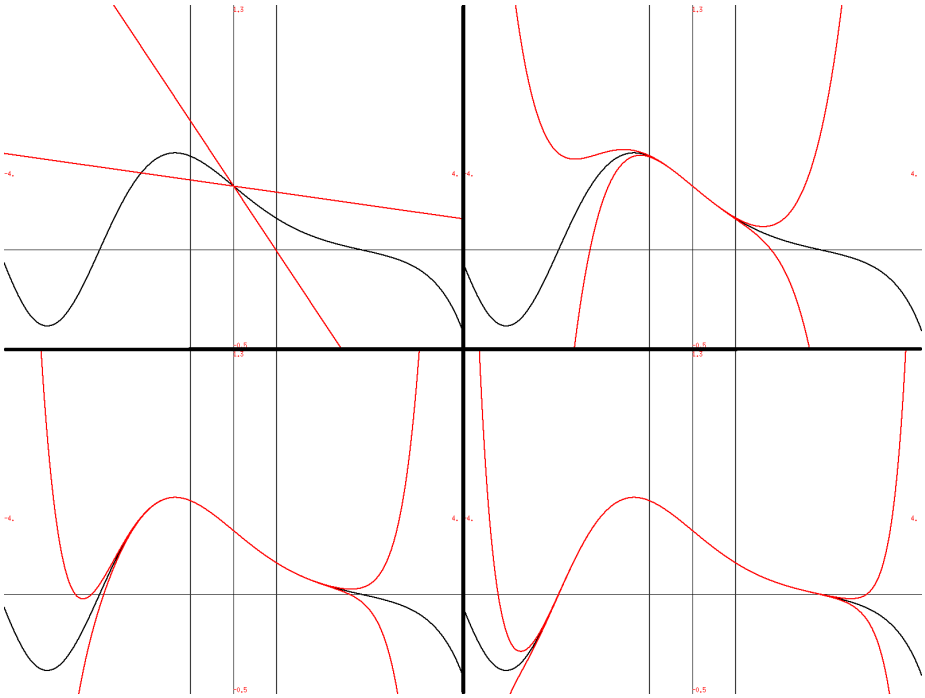


Figure 9: Airy ODE with errors

The **black** graph is the theoretical solution of Airy ODE.
The **grey** vertical bars shows the $[-.75, 0.75]$ interval on which errors are computed.
The **red** graphs are the error bounds of the approximation on interval $[-0.75, 0.75]$ at orders 1, 5, 10 and 15.

$$
\begin{aligned}
\exp\left(A_n^L + X_n.A_{n+1}^R + V_A\right) &= \exp(V_A)\exp\left(A_n^L + X_n.A_{n+1}^R\right) \\
\log\left(A_n^L + X_n.A_{n+1}^R + V_A\right) &= \log(V_A) + \log\left(1 + \tfrac{A_n^L + X_n.A_{n+1}^R}{V_A}\right) \\
\sin\left(A_n^L + X_n.A_{n+1}^R + V_A\right) &= \sin(V_A)\cos\left(A_n^L + X_n.A_{n+1}^R\right) \\
&\quad + \cos(V_A)\sin\left(A_n^L + X_n.A_{n+1}^R\right) \\
\arctan\left(A_n^L + X_n.A_{n+1}^R + V_A\right) &= \arctan(V_A) + \arctan\left(\tfrac{A_n^L + X_n.A_{n+1}^R}{1 + V_A.(A_n^L + X_n.A_{n+1}^R)}\right)
\end{aligned}
$$

We are currently developing a canonical composition operator $f \circ g$ following decomposition schemes that are all well known to strongly involve combinatorial reasoning. Our preliminary results already show that the administrative content of such heavy combinatorial computations, such as iterating over partitions, combinations, permutations and so on, have a great cost and are not yet on a par with the differential approach in terms of efficiency, at least for the tested instances. More investigation is required in that respect. We still expect to obtain an efficient canonical solution, with a simpler error propagation scheme

and furthermore less computations to reduce such propagation.

9.2 Going further

Many other sensible choices for computing errors are also possible such as arbitrary intervals, zonotopes, etc, but we haven't experimented with these solutions yet. We chose to stick to the lightweight zero-centered error domain, giving up some precision to save computation time, mostly because it is much simpler to implement and also because we rely on on-demand cotensor exploration to increase precision, by computing deeper coefficients of Taylor expansions. We nevertheless plan to address the problem of finding a well-suited error domain, in terms of precision with respect to computation time.

Accounting for numerical errors is also on our roadmap. As a first approach, we postulate that we would only have to represent every real number with an interval of lower and upper approximations given as two floating-point numbers, lifting every computation from an algebra of real numbers to an algebra of floating-point intervals. The main question will be to test whether accumulating numerical errors along a huge number of computations could significantly degrade precision, as the derivation order increases, jeopardizing the core feature of our framework.

Another method, closely related to our own functional language framework exploiting laziness, would be to consider using a setup for exact real number algebra, as illustrated for instance in [5]. Besides its lack of efficiency wrt. floating-point numbers, it would not suffer from a potential untamable accumulation of errors and would also open the way for a complete formal verification (including tensorial structure and numerical aspects).

As a last remark, veering off from certified errors, we emphasize the fact that our data-structures and operations are to a large extent unaware of a specific choice of basis. Thus we could express our analytical functions in another basis than the monomial one if it better fits our needs, without deeply impacting the development so far. One such sensible choice is the Poisson basis [6], which is used for geometric approximation and modelization. This would allow defining curves, surfaces, hypersurfaces, etc, as solutions of PDE, in a very compact way.

10 Conclusion

With a renovated view on Taylor series, we provide an implementation of a genuine full-fledged algebra of such series, in the multivariate case. Even if the work is far from being completed, it has been proven useful already as we are able to deal smoothly with partial differential equations in solved form, without any input from domain expert. To the best of our knowledge, implementing such an algebra of Taylor series with a concern on efficiency through carefully crafted algorithmics but also on correctness through strong typing has not been tried before. Indeed, although not presented here, our implementation puts an emphasis on strong typing, through extensive use of advanced OCaml GADT

features. This proved really helpful in designing correct-by-construction code, at least with respect to dimensions and derivation orders, while implementing complex and error-prone numerical computations.

Also, we are not aware of any implementation of such series that handles certified errors, even in the univariate case, while respecting our on-demand lazy computation scheme. This is a first promising step which paves the way for applying our library in the paradigm of guaranteed integration for instance. Careful investigations are needed to address the much more complex multivariate case.

The next big challenges to take up are: first, the introduction of a better composition scheme; second, various error domains and computation schemes compatible with every construction of our algebra.

References

[1] A. Pearlmutter, B., Siskind, J.: Lazy multivariate higher-order forward-mode AD. POPL 2007 (Jan 2007)

[2] Berz, M., Makino, K.: Verified integration of odes and flows using differential algebraic methods on high-order taylor models. Reliable Computing **4**(4), 361–369 (1998). https://doi.org/10.1023/A:1024467732637, `https://doi.org/10.1023/A:1024467732637`

[3] Chen, X., Sankaranarayanan, S., Ábrahám, E.: Flowstar (march 2017), https://flowstar.org/

[4] dit Sandretto, J.A., Chapoutot, A.: Validated explicit and implicit Runge–Kutta methods. Reliable Computing **22**(1), 79–103 (Jul 2016)

[5] Geuvers, H., Niqui, M., Spitters, B., Wiedijk, F.: Constructive analysis, types and exact real numbers. Mathematical Structures in Computer Science **17**(1), 3–36 (2007). https://doi.org/10.1017/S0960129506005834

[6] Goldman, R.N., Morin, G.: Poisson approximation. In: Geometric Modeling and Processing 2000, Hong Kong, China, April 10-12, 2000. pp. 141–149. IEEE Computer Society (2000). https://doi.org/10.1109/GMAP.2000.838246, `https://doi.org/10.1109/GMAP.2000.838246`

[7] Karczmarczuk, J.: Functional differentiation of computer programs. Higher-Order and Symbolic Computation **14**(1), 35–57 (2001). https://doi.org/10.1023/A:1011501232197

[8] Kiselyov, O.: Delimited control in ocaml, abstractly and concretely. Theor. Comput. Sci. **435**, 56–76 (2012). https://doi.org/10.1016/j.tcs.2012.02.025, `https://doi.org/10.1016/j.tcs.2012.02.025`

[9] Makino, K., Berz, M.: Rigorous integration of flows and ODEs using Tylor models. In: Symbolic Numeric Computation, SNC '09, Kyoto, Japan - August 03 - 05, 2009. pp. 79–84 (2009). https://doi.org/10.1145/1577190.1577206

[10] Martin-Dorel, É., Hanrot, G., Mayero, M., Théry, L.: Formally verified certificate checkers for hardest-to-round computation. J. Autom. Reasoning **54**(1), 1–29 (2015). https://doi.org/10.1007/s10817-014-9312-2

[11] Martin-Dorel, É., Rideau, L., Théry, L., Mayero, M., Pasca, I.: Certified, efficient and sharp univariate Taylor models in COQ. In: 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013. pp. 193–200 (2013). https://doi.org/10.1109/SYNASC.2013.33

[12] Revol, N., Makino, K., Berz, M.: Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. J. Log. Algebr. Program. **64**(1), 135–154 (2005). https://doi.org/10.1016/j.jlap.2004.07.008

[13] Thirioux, X.: Verifying Embedded Systems. Habilitation thesis, Institut National Polytechnique de Toulouse, France (sept 2016)