Polynomialization of ordinary differential equations given by straight-line programs[*][†]

Joris van der Hoeven[a], Grégoire Lecerf[b], Arnaud Minondo[c]

Laboratoire d'informatique de l'École polytechnique (LIX, UMR 7161)
CNRS, École polytechnique, Institut Polytechnique de Paris
1, rue Honoré d'Estienne d'Orves
91120 Palaiseau, France

a. *Email:* `vdhoeven@lix.polytechnique.fr`
b. *Email:* `lecerf@lix.polytechnique.fr`
c. *Email:* `minondo@lix.polytechnique.fr`

Preliminary version of February 6, 2026

Given a system of ordinary differential equations represented by a straight-line program, we show how to compute an equivalent ordinary differential system represented by a straight-line program which only uses ring operations. Under mild assumptions, our method essentially runs in linear time.

1. Introduction

In this paper, $\mathbb{K}$ will represent either $\mathbb{R}$ or $\mathbb{C}$. Let Ω be an open subset of $\mathbb{K}^n$, let Φ be a map $\Omega \to \mathbb{K}^n$, we will consider the Ordinary Differential Equation (ODE)

$$(x_1', \ldots, x_n') = \Phi(x_1, \ldots, x_n) \tag{1}$$

where $(x_1, \ldots, x_n): I \to \Omega$ stands for an unknown function on an open interval $I \subseteq \mathbb{R}$. A *polynomialization* of Φ is a polynomial map $\Psi = (\Psi_1, \ldots, \Psi_m) \in \mathbb{K}[X_1, \ldots, X_m]^m$ with $m \geqslant n$ such that any solution $(x_1, \ldots, x_n)$ of (1) on I extends into a solution $(x_1, \ldots, x_m)$ of

$$(x_1', \ldots, x_m') = \Psi(x_1, \ldots, x_m) \tag{2}$$

which is also defined on I. We say that a solution $(x_1, \ldots, x_n): I \to \Omega$ of (1) satisfies the initial condition

$$(\tau, \xi) \in \mathbb{R} \times \mathbb{K}^n$$

if $\tau \in I$ and $(x_1(\tau), \ldots, x_n(\tau)) = \xi$. Finding such solutions is called an *initial value problem*.

This paper is devoted to the fast computation of polynomializations of Φ, along with the corresponding initial conditions.

*. Grégoire Lecerf and Arnaud Minondo have been supported by the French ANR-22-CE48-0016 NODE project. Joris van der Hoeven has been supported by an ERC-2023-ADG grant for the ODELIX project (number 101142171).

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

†. This article has been written using GNU $\mathrm{T_EX_{MACS}}$ [11, 14].

1.1. Motivation

ODEs arising in many areas of physics and chemistry are often non-linear and include divisions, exponentiations, logarithms, square roots, etc. Although methods for ODE polynomialization are well known, we are not aware of complexity bounds for the case where Φ is given by a Straight-Line Program (SLP for short, defined in section 2). Although numerical integrators typically allow Φ to be an arbitrary blackbox function, it is most common that the implementation of Φ is actually an SLP.

Our motivation is three-fold. From a theoretical point of view, we are interested in a sharp worst case complexity bound.

Polynomialization is of practical interest, because the quotients and special functions occurring in Φ are typically more expensive to evaluate than the simpler ring operations occurring in Ψ. For example, when using a recent Intel™ CPU with SKYLAKE™ architecture, an AVX512 division in double precision has a latency of 23 CPU cycles and a throughput of 16, while one "Fused Multiply-Add" instruction (see section 3.3) has a latency of 4 and a throughput of 0.5. Similar kinds of overhead are observed for other coefficient types such as intervals and balls [13, 16].

Of course, in order to apply our work to speed up the numerical integration of (1), one should use an ODE solver that is able to exploit the lower evaluation cost of Ψ, while not suffering too much from the increased dimension $m > n$. This is typically the case for Runge–Kutta integrators [4], for which the complexity depends linearly on the evaluation cost of Φ. Lazy Taylor series integrators [10] implicitly use something close to polynomialization for the evaluation of Φ (e.g. exponentials $f = \mathrm{e}^g$ are evaluated by solving the equation $g' = f' g$). On the other hand, integrators that also compute the first variation (e.g. in order to improve the stability) may suffer from the increased dimension.

Finally, we plan to use polynomialization to simplify bound computations for certified Runge–Kutta ODE solvers as in [2]. Such bound computations are easier and tighter for SLPs that only use ring operations. Decreasing the degrees of the polynomials Ψ_i is also beneficial for this application.

1.2. Related work

Polynomialization is certainly part of the folklore of differential algebra. Recent studies focus on quadratization, i.e. a polynomialization $\Psi = (\Psi_1, \ldots, \Psi_m)$ with $\max_i \deg \Psi_i \leqslant 2$. Quadratization was established at least more than a century ago: historical and recent references, along with various applications, can be found in [5, 6, 8, 9, 15]. Polynomialization is achieved in quadratic time in [8, 9] for a dense representation of polynomials. In [8] the problem of finding the minimal number of extra variables necessary for quadratization is shown to be NP-hard. In [5] a quadratization with minimal order (namely $m - n$) is achieved for the dense representation of Φ. Polynomializations and quadratizations are not unique, and preserving stability is sometimes more important than the sole complexity issue; see various strategies in [6].

1.3. Our contributions

Our first contribution is a new method for polynomializing ODEs represented by SLPs at the price of increasing the number of instructions by a small constant factor. The precise overhead depends on the arithmetic instructions allowed by the SLP framework. Theorem 9 focuses on SLPs that only use additions, subtractions, and products, while Theorem 11 achieves a lower overhead if we also allow "Fused Multiply-Add" instructions.

Our second contribution concerns practical improvements over our worst case complexity bound. On the one hand, we restrict the additional costs to the subexpressions actually involved in the instructions of Φ which are not polynomials. On the other hand, we try to minimize the number of extra variables, namely $m-n$. Our optimizations and heuristics take linear time up to logarithmic factors. Even when polynomializing just a few instructions of a small SLP, our method turns out to of significant practical interest.

Our third contribution is an implementation of our polynomialization algorithm within the JIL library [1, 12], which is a free software C++ library for HPC (High-Performance Computing) computations with SLPs.

2. STRAIGHT-LINE PROGRAMS

Given $n \in \mathbb{N}$, we write $\mathbb{N}_n := \{0,\ldots,n-1\}$. Given two tuples $a=(a_1,\ldots,a_n)$ and $b=(b_1,\ldots,b_m)$, their *concatenation* $(a_1,\ldots,a_n,b_1,\ldots,b_m)$ is written $a \bowtie b$. We will encode names of variables by integers (typically of 32 bit or 64 bit machine size). For complexity analyses, we will assume that these integers are of size $O(1)$ and count unit time for operations on such integers. Similarly, we assume that elements of $\mathbb{K}$ take unit size. (Here we note that our main polynomialization algorithms from Theorems 9 and 11 require no arithmetic operations in $\mathbb{K}$.) Below, we follow the SLP framework of [12].

2.1. Definition

Let Σ be a set of operation symbols together with a function $|\cdot|:\Sigma \to \mathbb{N}$. We call Σ a *signature* and $|\sigma|$ the *arity* of σ, for any operation $\sigma \in \Sigma$. A *domain* with signature Σ is a set $\mathbb{A}$ together with a function $\sigma_{\mathbb{A}}: \mathbb{A}^{|\sigma|} \to \mathbb{A}$ for every $\sigma \in \Sigma$. We often write σ instead of $\sigma_{\mathbb{A}}$ if no confusion can arise.

Let $\mathbb{A}$ be a domain with signature Σ. An *instruction* is a tuple

$$\pi = (\sigma, a_0, \ldots, a_{|\pi|})$$

with *operation* $\sigma_\pi := \sigma \in \Sigma$ and *arguments* $a_0, \ldots, a_{|\pi|} \in \mathbb{N}$, where $|\pi| := |\sigma|$. We call a_0 the *destination argument* or *operand* and $a_1, \ldots, a_{|\pi|}$ the *source arguments* or *operands*. We denote by $\mathbb{I}_\Sigma$ the set of such instructions. Given $n \in \mathbb{N}$, we also let

$$\mathbb{I}_{\Sigma,n} := \{(\sigma, a_0, \ldots, a_{|\pi|}) \in \mathbb{I}_\Sigma : \sigma \in \Sigma, a_0, \ldots, a_{|\pi|} \in \mathbb{N}_n\}.$$

A *straight-line program* (*SLP*) over $\mathbb{A}$ is a quadruple $f=(D,I,O,P)$, where

- $D=(D_0,\ldots,D_{|D|-1}) \in \mathbb{A}^{|D|}$ is a tuple of data fields,
- $I=(I_0,\ldots,I_{|I|-1}) \in \mathbb{N}_{|D|}^{|I|}$ is a tuple of pairwise distinct input locations,
- $O=(O_0,\ldots,O_{|O|-1}) \in \mathbb{N}_{|D|}^{|O|}$ is a tuple of output locations,
- $P=(P_0,\ldots,P_{|P|-1}) \in \mathbb{I}_{\Sigma,|D|}^{|P|}$ is a tuple of instructions.

We regard $\mathbb{A}^{|D|}$ as the *working space* for our SLP. Each location in this working space can be represented using an integer in $\mathbb{N}_{|D|}$ and instructions directly operate on the working space as follows.

Given a *state* $\delta \in \mathbb{A}^{|D|}$ of our working space, the *execution* of an instruction $\pi=(\sigma,a_0,\ldots,a_{|\pi|}) \in \mathbb{I}_{\Sigma,|D|}$ gives rise to a new state $\delta' := \pi(\delta)$ where $\delta'_k := \delta_k$ if $k \neq a_0$ and $\delta'_k := \sigma_\pi(\delta_{a_1},\ldots,\delta_{a_{|\pi|}})$ if $k=a_0$. Given *input values* $x=(x_1,\ldots,x_{|I|}) \in \mathbb{A}^{|I|}$, the corresponding begin state $\delta_{0;}$ of our SLP is given by $\delta_{0;k} := D_k$ if $k \notin \{I_0,\ldots,I_{|I|-1}\}$ and $\delta_{0;I_k} := x_{k+1}$ for $k \in \mathbb{N}_{|I|}$. Execution of the SLP next gives rise to a sequence of states $\delta_{1;} := P_0(\delta_{0;}),\ldots,\delta_{|P|;} := P_{|P|-1}(\delta_{|P|-1;})$. The *output values* $y=(y_1,\ldots,y_{|O|}) \in \mathbb{A}^{|O|}$ of our SLP are now given by $y_{k+1} := \delta_{|P|;O_k}$ for $k \in \mathbb{N}_{|O|}$. In this way our SLP f gives rise to a function $\mathbb{A}^{|I|} \to \mathbb{A}^{|O|}; x \mapsto y$ that we will also denote by f. Two SLPs are said to be *equivalent* if they compute the same function.

It will be useful to call elements of $\mathbb{N}_{|D|}$ *variables* and denote them by more intelligible symbols like $v_0,\ldots,v_{|D|-1}$ whenever convenient. The entries of I and O are called *input* and *output variables*, respectively. Non-input variables that also do not occur as destination operands of instructions in P are called *constants*. An *auxiliary variable* is a variable that is neither an input variable, nor an output variable, nor a constant. Given a variable $v \in \mathbb{N}_{|D|}$, the corresponding *data field* is D_v. Input variables, output variables, and constants naturally correspond to *input fields*, *output fields*, and *constant fields*. For a subset $\Sigma' \subseteq \Sigma$, we denote $|P|_{\Sigma'} := |\{i \in \mathbb{N}_{|P|} : \sigma_{P_i} \in \Sigma'\}|$; given $\sigma \in \Sigma$, we also abbreviate $|P|_\sigma := |P|_{\{\sigma\}}$.

Example 1. Let $\mathbb{A} := \mathbb{N}$ and let $\Sigma = \{+, \times\}$ be the signatures of the binary sum and product. Let Φ be the SLP defined by $D = (0,0,2)$, $I := (1)$, $O := (0)$, and $P := ((+,0,1,2),(\times,0,0,0))$. With v_0, v_1, v_2 representing the values at positions $0,1,2$ in the working space, the instructions of f rewrite into $v_0 := v_1 + 2$, $v_0 := v_0^2$. Consequently, Φ computes the function $x_1 \mapsto (x_1 + 2)^2$.

When dealing with operations that are only partially defined, such as the unary inverse over fields, we say that an SLP is *executable* at a given input when all its instructions have their arguments in their definition domain during the execution.

Remark 2. The bit sizes of the integers in I and O and in the instructions of P are at most $O(\log |D|)$. For practical software implementations it is fair to consider that we always have $|D| \leqslant 2^{64}$ or even $|D| \leqslant 2^{32}$, so that these integers can be represented by hardware integers. This also "justifies" why we count a unit cost for integer operations in our complexity analyses.

2.2. Standard signatures

The *standard signature of a ring* $\mathbb{A}$, written Σ^{ring}, will be the set of the following operations:

- unary identity id, defined by $\mathsf{id}_{\mathbb{A}}: a \mapsto a$,
- unary negation $-$, defined by $-_{\mathbb{A}}: a \mapsto -a$,
- binary subtraction, also written $-$, defined by $-_{\mathbb{A}}: (a,b) \mapsto a - b$,
- binary addition $+$, defined by $+_{\mathbb{A}}: (a,b) \mapsto a + b$,
- binary multiplication $\times$, defined by $\times_{\mathbb{A}}: (a,b) \mapsto a \times b$.

The *standard signature of a field* $\mathbb{K}$ will be the union $\Sigma^{\text{field}} := \Sigma^{\text{ring}} \cup \Sigma^{\text{div}}$, where Σ^{div} is the set of the following operations:

- unary inversion inv, defined by $\mathsf{inv}_{\mathbb{K}}: a \mapsto a^{-1}$,
- binary division $/$, defined by $/_{\mathbb{K}}: (a,b) \mapsto a/b$.

Let Σ^{special} be a fixed finite set of operations σ of arity one such that each $\sigma \in \Sigma^{\text{special}}$ satisfies a differential equation

$$(\sigma(t)', x_2(t)', \ldots, x_{\mu_\sigma}(t)') = Q_\sigma(\sigma(t), x_2(t), \ldots, x_{\mu_\sigma}(t), t),$$

where $Q_\sigma: \mathbb{K}^{\mu_\sigma+1} \to \mathbb{K}$ is given by an SLP with signature Σ^{ring} over $\mathbb{K}$. We also assume that $q_\sigma: t \mapsto (\sigma(t), x_2(t), \ldots, x_{\mu_\sigma}(t))$ is given by an SLP with signature $\Sigma^{\text{field}} \cup \Sigma^{\text{special}}$ over $\mathbb{K}$. In this case, we call $\mathbb{K}$ a field with *special functions* Σ^{special}.

For convenience, Q_σ and q_σ will be rewritten in terms of macro instructions (following the usual programming language terminology). Precisely, we will assume given the following data:

- A segment of data D^{special} over $\mathbb{K}$ gathers all the constants and auxiliary variables involved in Q_σ and q_σ for all $\sigma \in \Sigma^{\text{special}}$.

- For all $\sigma \in \Sigma^{\text{special}}$, a macro instruction X_σ takes $2(\mu_\sigma + 1)$ integer arguments and returns a sequence of instructions of length written $|\mathrm{X}_\sigma|$. The last argument of X_σ stands for the position of D^{special} in the working space. For any $\delta \in \mathbb{K}^{|\delta|}$ and for all $a_1, \ldots, a_{\mu_\sigma}, b_1, \ldots, b_{\mu_\sigma+1}$ in $\mathbb{N}_{|\delta|}$, the execution of $\mathrm{X}_\sigma(a_1, \ldots, a_{\mu_\sigma}, b_1, \ldots, b_{\mu_\sigma+1}, |\delta|)$ at the state $\delta \bowtie D^{\text{special}}$ performs $(v_{a_1}, \ldots, v_{a_{\mu_\sigma}}) := Q_\sigma(v_{b_1}, \ldots, v_{b_{\mu_\sigma+1}})$ and only modifies the values at positions $a_1, \ldots, a_{\mu_\sigma}$ inside δ (the data fields in D^{special} may be modified freely).
- For all $\sigma \in \Sigma^{\text{special}}$, a macro instruction χ_σ takes $\mu_\sigma + 2$ integer arguments and returns a sequence of instructions of length written $|\chi_\sigma|$. Again, the last argument of χ_σ is the position of D^{special} in the working space. For any $\delta \in \mathbb{K}^{|\delta|}$ and all $a_1, \ldots, a_{\mu_\sigma}, b_1$ in $\mathbb{N}_{|\delta|}$, the execution of $\chi_\sigma(a_1, \ldots, a_{\mu_\sigma}, b_1, |\delta|)$ at the state $\delta \bowtie D^{\text{special}}$ performs $(v_{a_1}, \ldots, v_{a_{\mu_\sigma}}) := q_\sigma(v_{b_1})$ and only modifies the values at positions $a_1, \ldots, a_{\mu_\sigma}$ inside δ (the data fields in D^{special} may be modified freely).

Example 3. The exp function may be included in Σ^{special} with the defining functions $Q_{\exp}(x_1, t) := (x_1)$ and $q_{\exp}(t) := (\exp t)$. We may take $\mathrm{X}_{\exp}(a_1, b_1, b_2, |\delta|) := (v_{a_1} := v_{b_1})$ and $\chi_{\exp}(a_1, b_1, |\delta|) := (v_{a_1} := \exp v_{b_1})$.

Example 4. The log function may be included in Σ^{special} with the defining functions $Q_{\log}(x_1, x_2, t) := (x_2, -x_2^2)$ and $q_{\log}(t) := (\log t, 1/t)$. We may take $\mathrm{X}_{\log}(a_1, a_2, b_1, b_2, b_3, |\delta|) = (v_{a_1} := v_{b_2}, v_{a_2} := v_{b_2} v_{b_2}, v_{a_2} := -v_{a_2})$ and $\chi_{\log}(a_1, a_2, b_1, |\delta|) = (v_{|\delta|+c} := v_{b_1}, v_{a_1} := \log v_{|\delta|+c}, v_{a_2} := v_{|\delta|+c}^{-1})$, where c stands for the position of an auxiliary variable in D^{special}.

Example 5. For any fixed $r \in \mathbb{Q}$, the fractional power function $\sigma: t \mapsto t^r$ may be included in Σ^{special} with the defining functions $Q_\sigma(x_1, x_2, t) := (r\, x_1 x_2, -x_2^2)$ and $q_\sigma(t) := (t^r, 1/t)$. If the constant r is stored at position ρ in D^{special} and if c stands for the position of an auxiliary variable in D^{special}, then we may take $\mathrm{X}_\sigma(a_1, a_2, b_1, b_2, b_3, |\delta|) = (v_{|\delta|+c} := v_{b_2}, v_{a_1} := v_{b_1} v_{b_2}, v_{a_1} := v_{|\delta|+c} v_{a_1}, v_{a_2} := v_{|\delta|+c} v_{|\delta|+c}, v_{a_2} := -v_{a_2})$ and $q_\sigma(a_1, a_2, b_1, |\delta|) = (v_{|\delta|+c} := v_{b_1}, v_{a_1} := v_{|\delta|+c}^r, v_{a_2} := v_{|\delta|+c}^{-1})$.

2.3. Postponing divisions

An SLP $F = (D, I, O, P): \mathbb{K}^n \to \mathbb{K}^m$ with signature Σ^{field} may be transformed into an equivalent SLP with at most m divisions: one for each output. (In fact, one may even reduce to the case when we perform just one division, but this will not be needed in what follows.)

Let us show how to compute an SLP $\tilde{F} = (\tilde{D}, \tilde{I}, \tilde{O}, \tilde{P})$ for evaluating polynomials $A_1, B_1, \ldots, A_n, B_n$ such that $F = \left(\frac{A_1}{B_1}, \ldots, \frac{A_n}{B_n}\right)$ and none of the B_i vanishes at the set $\Omega \subseteq \mathbb{K}^n$ of points where F is executable. Roughly speaking, we obtain the A_i and B_i by rewriting elements of $\mathbb{K}$ as fractions, which are encoded as pairs of (not necessarily coprime) numerators and denominators. This rewriting process is well known and takes linear time. It is a special instantiation of the lifting algorithm from [12, section 5.5].

We now describe $\tilde{F}$ more precisely and bound its size. We define the following tuples:

- $\tilde{D} := (\tilde{D}_0, \ldots, \tilde{D}_{2|D|}) \in \mathbb{K}^{2|D|+1}$ is defined by $\tilde{D}_{2i} := D_i$ and $\tilde{D}_{2i+1} := 1$ for $i \in \mathbb{N}_{|D|}$. The last entry $\tilde{D}_{2|D|}$ is an auxiliary variable which may be filled arbitrarily. An entry at position $2i < 2|D|$ represents a numerator whose corresponding denominator is at position $2i+1$;
- $\tilde{I} := (2I_0, \ldots, 2I_{|I|-1})$;
- $\tilde{O} := (2O_0, 2O_0 + 1, \ldots, 2O_{|O|-1}, 2O_{|O|-1} + 1)$. The entries $2O_{i-1}$ and $2O_{i-1} + 1$ represent the values of A_i and B_i for $i = 1, \ldots, n$;

- $\tilde{P} := \bowtie_{p=0,\ldots,|P|-1} \operatorname{Lift} P_p$, where Lift maps instructions of P to sequences of instructions, as follows, where we write $V_i := v_{2i}$,$W_i := v_{2i+1}$, and $A := v_{|2D|}$ as shorthands:

$$\begin{aligned}
\operatorname{Lift}(v_i := v_j) &:= (V_i := V_j,\ W_i := W_j)\\
\operatorname{Lift}(v_i := -v_j) &:= (V_i := -V_j,\ W_i := W_j)\\
\operatorname{Lift}(v_i := v_j - v_k) &:= (A := V_j W_k,\ V_i := V_k W_j, V_i := A - V_i,\ W_i := W_j W_k)\\
\operatorname{Lift}(v_i := v_j + v_k) &:= (A := V_j W_k,\ V_i := V_k W_j, V_i := A + V_i,\ W_i := W_j W_k)\\
\operatorname{Lift}(v_i := v_j v_k) &:= (V_i := V_j V_k,\ W_i := W_j W_k)\\
\operatorname{Lift}(v_i := v_j^{-1}) &:= (A := V_j,\ V_i := W_j,\ W_i := A)\\
\operatorname{Lift}(v_i := v_j / v_k) &:= (A := V_k,\ V_i := V_j W_k,\ W_i := W_j A).
\end{aligned}$$

For example, with v_i representing the value at position i in the working space, the Lift of a sum $v_i := v_j + v_k$ consists in computing

$$\frac{V_j}{W_j} + \frac{V_k}{W_k} = \frac{V_j W_k + V_k W_j}{W_j W_k}.$$

We observe that $\tilde{\Phi}$ is executable over $\mathbb{K}^n$, that none of the B_i vanishes at Ω for $i = 1, \ldots, n$, and that

$$|\tilde{P}| \leqslant 4|P|. \qquad (3)$$

Example 6. Let $n := 1$ and let Φ be the SLP defined by $D := (0,0,2)$, $I := (1)$, $O := (0)$, and $P := ((+,0,1,2),(/,0,1,0))$. With v_0, v_1, v_2 representing the values at positions $0,1,2$ of the working space, the instructions of Φ rewrite into $v_0 := v_1 + 2$, $v_0 = v_1 / v_0$. Consequently, Φ computes the rational function $X_1/(X_1+2)$. Applying the above construction gives us $\tilde{D} := (0,1,0,1,2,1,0)$, $\tilde{I} := (2)$, $\tilde{O} := (0,1)$, and $\tilde{P}$ corresponds to the following sequence of instructions: $v_6 := v_2 v_5$, $v_0 := 2 v_3$, $v_0 := v_6 + v_0$, $v_1 := v_3 v_5$, $v_6 := v_0$, $v_0 := v_2 v_1$, $v_1 := v_6 v_3$. The expressions of the output values at positions 0 and 1 are $v_2 v_3 v_5$ and $v_3 (v_2 v_5 + 2 v_3)$. So we have $A_1 := X_1$ and $B_1 := X_1 + 2$.

Example 7. Let $n := 1$ and let Φ be the SLP defined by $D := (0,0)$, $I := (1)$, $O := (0)$, and $P := (v_0 := v_0 / v_1,\ v_0 := v_1)$. Although the first instruction is useless, Φ is not executable at 0. On the other hand, we have $A_1 = X_1$ and $B_1 = 1$. This shows that $B_1 \cdots B_n$ may be inverted over a domain strictly larger than Ω.

Remark 8. In absence of "aliasing", the lifting process can be further optimized. For instance, if $i \neq j$, then one may take $\operatorname{Lift}(v_i := v_j^{-1}) := (V_i := W_j,\ W_i := V_j)$. Our implementation in JIL systematically exploits such optimizations; see also [12, section 5.5].

2.4. Tangent numbers

An SLP $F = (D, I, O, P)$: $\mathbb{K}^n \to \mathbb{K}^m$ with signature Σ^{ring} may be transformed into an SLP $\tilde{F} = (\tilde{D}, \tilde{I}, \tilde{O}, \tilde{P})$ over $\mathbb{K}$ with signature Σ^{ring} which evaluates F over $(\mathbb{K}[\epsilon]/(\epsilon^2))^n$, where ϵ is a new variable. We define the following tuples:

- $\tilde{D} := (\tilde{D}_0, \ldots, \tilde{D}_{2|D|}) \in \mathbb{K}^{2|D|+1}$ is defined by $\tilde{D}_{2i} := D_i$ and $\tilde{D}_{2i+1} := 0$ for $i \in \mathbb{N}_{|D|}$. The last entry $\tilde{D}_{2|D|}$ is an auxiliary variable which may be filled arbitrarily. A variable v_i for F becomes $v_{2i} + v_{2i+1}\epsilon$ for $\tilde{F}$.
- $\tilde{I} := (2I_0, 2I_0 + 1, \ldots, 2I_{|I|-1}, 2I_{|I|-1} + 1)$;
- $\tilde{O} := (2O_0, 2O_0 + 1, \ldots, 2O_{|O|-1}, 2O_{|O|-1} + 1)$;

- $\tilde{P} := \bowtie_{p=0,\ldots,|P|-1} \mathsf{Lift}\, P_p$, where Lift maps instructions of P to sequences of instructions, as follows, where we use $V_i := v_{2i}$ $\dot{V}_i := v_{2i+1}$, and $A := v_{|2D|}$ as shorthands:

$$\begin{aligned}
\mathsf{Lift}(v_i := v_j) &:= (V_i := V_j,\ \dot{V}_i := \dot{V}_j)\\
\mathsf{Lift}(v_i := -v_j) &:= (V_i := -V_j,\ \dot{V}_i := -\dot{V}_j)\\
\mathsf{Lift}(v_i := v_j - v_k) &:= (V_i := V_j - V_k,\ \dot{V}_i := \dot{V}_j - \dot{V}_k)\\
\mathsf{Lift}(v_i := v_j + v_k) &:= (V_i := V_j + V_k,\ \dot{V}_i := \dot{V}_j + \dot{V}_k)\\
\mathsf{Lift}(v_i := v_j v_k) &:= (A := V_j \dot{V}_k,\ \dot{V}_i := \dot{V}_j V_k, \dot{V}_i := A + \dot{V}_i,\ V_i := V_j V_k).
\end{aligned}$$

Consequently,

$$|\tilde{P}| \leqslant 4\,|P|. \tag{4}$$

3. POLYNOMIALIZATION

We begin this section with a rather natural method for polynomializing SLPs over $\mathbb{K}$ with signature Σ^{field}. Then, we present our new faster and general approach.

3.1. SLPs with divisions

Let $\Phi = \left(\frac{A_1}{B_1}, \ldots, \frac{A_n}{B_n}\right)$ where $A_i \in \mathbb{K}[X_1, \ldots, X_n]$ and $B_i \in \mathbb{K}[X_1, \ldots, X_n]^*$ for $i = 1, \ldots, n$, and let $(x_1, \ldots, x_n)$ be a solution of (1) defined over I. We introduce the functions

$$\begin{aligned}
x_{n+i} &:= x_i'\\
x_{2n+i} &:= B_i(x_1, \ldots, x_n)^{-1}
\end{aligned}$$

over I, for $i = 1, \ldots, n$, and obtain

$$\begin{aligned}
x_{n+i}' &= \mathrm{D}A_i(x_1, \ldots, x_n)(x_{n+1}, \ldots, x_{2n})\, x_{2n+i} - A_i(x_1, \ldots, x_n)\, \mathrm{D}B_i(x_1, \ldots, x_n)(x_{n+1}, \ldots, x_{2n})\, x_{2n+i}^2\\
x_{2n+i}' &= -\mathrm{D}B_i(x_1, \ldots, x_n)(x_{n+1}, \ldots, x_{2n})\, x_{2n+i}^2.
\end{aligned}$$

Letting

$$\begin{aligned}
E_i &:= \mathrm{D}A_i(X_1, \ldots, X_n)(X_{n+1}, \ldots, X_{2n})\\
F_i &:= \mathrm{D}B_i(X_1, \ldots, X_n)(X_{n+1}, \ldots, X_{2n}),
\end{aligned}$$

the polynomial map

$$\Psi(X_1, \ldots, X_{3n}) := \begin{pmatrix} X_{n+1}\\ \vdots\\ X_{2n}\\ E_1 X_{2n+1} - A_1 F_1 X_{2n+1}^2\\ \vdots\\ E_n X_{3n} - A_n F_n X_{3n}^2\\ -F_1 X_{2n+1}^2\\ \vdots\\ -F_n X_{3n}^2 \end{pmatrix}$$

is a polynomialization of Φ, since

$$(x_1, \ldots, x_{3n})' = \Psi(x_1, \ldots, x_{3n}).$$

The computation of Ψ can be done efficiently as follows.

From the SLP for Φ with signature Σ^{field} we first determine an SLP $\tilde{\Phi}$ with signature Σ^{ring} which computes $A_1, \ldots, A_n, B_1, \ldots, B_n$. The number of instructions of $\tilde{\Phi}$ is $\leqslant 4|P|$ using (3).

By computing $\tilde{\Phi}(X_1 + \epsilon X_{n+1}, \ldots, X_n + \epsilon X_{2n})$ modulo ϵ^2 we next build another SLP for evaluating A_i, B_i, E_i, and F_i for $i = 1, \ldots, n$ using at most $4|\tilde{P}|$ instructions, by (4). This finally leads to an SLP of length at most

$$16|P| + 6n \tag{5}$$

for the polynomialization of Φ.

As to the initial conditions, we take:

$$\begin{aligned}(x_{n+1}(\tau), \ldots, x_{2n}(\tau)) &= \Phi(\xi)\\ (x_{2n+1}(\tau), \ldots, x_{3n}(\tau)) &= (B_1(\xi)^{-1}, \ldots, B_n(\xi)^{-1}).\end{aligned}$$

This requires $\leqslant 4|P| + n$ ring operations plus n inversions.

3.2. SLPs with special functions

In the previous subsection, we reduced the polynomialization of Φ to the case when it is explicitly given by rational functions. In this subsection, we present a new faster and more general method, which directly transforms all divisions and special functions of the SLP of Φ. Roughly speaking, a special instruction $r := \sigma(v)$ gives rise to μ_σ new unknown functions $u_1, \ldots, u_{\mu_\sigma}$ with $u_1 := \sigma(v)$ and which satisfy the differential equation

$$(u_1(v), \ldots, u_{\mu_\sigma}(v))' = v' \, Q_\sigma(u_1(v), \ldots, u_{\mu_\sigma}(v), v).$$

Our method is detailed in the proof of the following theorem, which improves upon the above polynomialization with cost (5).

THEOREM 9. *Let $\Phi: \mathbb{K}^n \to \mathbb{K}^n$ be given by an SLP (D, I, O, P) over $\mathbb{K}$ with signature $\Sigma^{\text{field}} \cup \Sigma^{\text{special}}$ and let $\mu_{\mathsf{inv}} := 1$, $\mu_/ := 1$, and*

$$\theta_p := \sum_{q < p, \sigma = \sigma_{P_q} \in \{\mathsf{inv}, /\} \cup \Sigma^{\text{special}}} \mu_\sigma.$$

We can compute the following SLPs over $\mathbb{K}$ in time $O(|D| + |P| + n)$:

- *$(\tilde{D}, \tilde{I}, \tilde{O}, \tilde{P})$ uses Σ^{ring} and represents a polynomialization Ψ of Φ such that $|\tilde{D}| = 2|D| + 2\theta_{|P|} + |D^{\text{special}}| + 1$, $|\tilde{I}| = |\tilde{O}| = 2n + \theta_{|P|}$, and*

$$|\tilde{P}| \leqslant 4|P|_{\Sigma^{\text{ring}}} + 7|P|_{\{\mathsf{inv}, /\}} + \sum_{\sigma = \sigma_{P_p} \in \Sigma^{\text{special}}} (|\mathrm{X}_\sigma| + \mu_\sigma + 2).$$

- *$(\bar{D}, \bar{I}, \bar{O}, \bar{P})$ uses $\Sigma^{\text{field}} \cup \Sigma^{\text{special}}$, computes the initial value $\psi(\xi, \tau)$ for Ψ from an initial value ξ for Φ at τ, and satisfies $|\bar{D}| = |D| + n + \theta_{|P|} + |D^{\text{special}}|$, $|\bar{I}| = n$, $|\bar{O}| = 2n + \theta_{|P|}$, and $|\bar{P}| = |P| + n + \theta_{|P|}$.*

Proof. The tuples $\tilde{D}, \tilde{I}, \tilde{O}, \tilde{P}$ are defined as follows:

- $\tilde{D} \in \mathbb{K}^{2|D| + 2\theta_{|P|} + |D^{\text{special}}| + 1}$ with $\tilde{D}_{2i} := D_i$, $\tilde{D}_{2i+1} := 0$ for $i \in \mathbb{N}_{|D|}$, $\tilde{D}_{2|D| + 2\theta_{|P|} + i} := D_i^{\text{special}}$ for $i \in \mathbb{N}_{|D^{\text{special}}|}$. Other entries of $\tilde{D}$ may be filled arbitrarily;
- $\tilde{I} := 2I \bowtie (2I + 1) \bowtie (2|D|, 2|D| + 2, \ldots, 2|D| + 2(\theta_{|P|} - 1))$;
- $\tilde{O} := 2O \bowtie (2O + 1) \bowtie (2|D| + 1, \ldots, 2|D| + 2(\theta_{|P|} - 1) + 1)$;

- $\tilde{P} := \bowtie_{p=0,\ldots,|P|-1} \mathsf{Lift}\, P_p$, where Lift maps instructions of P to tuples of instructions with signature Σ^{ring}. If P_p has type σ then we abbreviate $\mathscr{I}_{p;i} := v_{\tilde{I}_{2n+\theta_p+i}}$, $\mathscr{O}_{p,i} := v_{\tilde{O}_{2n+\theta_p+i}}$ and let $\mathscr{I}_p := (\mathscr{I}_{p;0}, \ldots, \mathscr{I}_{p;\mu_\sigma-1})$, $\mathscr{O}_p := (\mathscr{O}_{p;0}, \ldots, \mathscr{O}_{p;\mu_\sigma-1})$. We also abbreviate $V_i := v_{2i}$, $\dot{V}_i := v_{2i+1}$, $A := v_{|\tilde{D}|}$. The value of $\dot{V}_i$ will actually be the derivative of the value of V_i, and A will be an auxiliary variable. If $\sigma := \sigma_{P_p} \in \Sigma^{\mathrm{ring}}$, then we take

$$\begin{aligned}
\mathsf{Lift}(v_i := v_j) &:= (V_i := V_j, \dot{V}_i := \dot{V}_j)\\
\mathsf{Lift}(v_i := -v_j) &:= (V_i := -V_j, \dot{V}_i := -\dot{V}_j)\\
\mathsf{Lift}(v_i := v_j + v_k) &:= (V_i := V_j + V_k, \dot{V}_i := \dot{V}_j + \dot{V}_k)\\
\mathsf{Lift}(v_i := v_j - v_k) &:= (V_i := V_j - V_k, \dot{V}_i := \dot{V}_j - \dot{V}_k)\\
\mathsf{Lift}(v_i := v_j v_k) &:= (A := V_j \dot{V}_k, \dot{V}_i := \dot{V}_j V_k, \dot{V}_i := A + \dot{V}_i, V_i := V_j V_k).
\end{aligned}$$

  If $\sigma \in \Sigma^{\mathrm{div}}$, then we define

$$\begin{aligned}
\mathsf{Lift}(v_i := v_j^{-1}) &:= (V_i := \mathscr{I}_{p,0}, \mathscr{O}_{p;0} := -\dot{V}_j, \mathscr{O}_{p;0} := \mathscr{O}_{p;0} V_i, \mathscr{O}_{p;0} := \mathscr{O}_{p;0} V_i, \dot{V}_i := \mathscr{O}_{p;0})\\
\mathsf{Lift}(v_i := v_j / v_k) &:= (\mathscr{O}_{p;0} := -\dot{V}_k, \mathscr{O}_{p;0} := \mathscr{O}_{p;0} \mathscr{I}_{p;0}, \mathscr{O}_{p,0} := \mathscr{O}_{p,0} \mathscr{I}_{p,0},\\
&\qquad A := V_j \mathscr{O}_{p;0}, \dot{V}_i := \dot{V}_j \mathscr{I}_{p;0}, \dot{V}_i := A + \dot{V}_i, V_i := V_j \mathscr{I}_{p;0}).
\end{aligned}$$

  If $\sigma \in \Sigma^{\mathrm{special}}$, then we take

$$\begin{aligned}
\mathsf{Lift}(v_i := \sigma(v_j)) := {} & (V_i := \mathscr{I}_{p;0})\\
& \bowtie \mathrm{X}_\sigma(\tilde{O}_{2n+\theta_p}, \ldots, \tilde{O}_{2n+\theta_p+\mu_\sigma-1},\\
& \bowtie (\tilde{O}_{2n+\theta_p}, \ldots, \tilde{O}_{2n+\theta_p+\mu_\sigma-1}, \tilde{I}_{2n+\theta_p}, \ldots, \tilde{I}_{2n+\theta_p+\mu_\sigma-1}, 2j, 2|D| + 2\theta_{|P|})\\
& \bowtie (\mathscr{O}_{p;0} := \dot{V}_j \mathscr{O}_{p;0}, \ldots, \mathscr{O}_{p;\mu_\sigma-1} := \dot{V}_j \mathscr{O}_{p;\mu_\sigma-1})\\
& \bowtie (\dot{V}_i := \mathscr{O}_{p;0}).
\end{aligned}$$

Let us now prove that $(\tilde{D}, \tilde{I}, \tilde{O}, \tilde{P})$ represents a polynomialization of Φ. We first need to introduce appropriate notation. For this, let $(x_1, \ldots, x_n) \in \mathscr{F}^n$ be a solution of (1), where $\mathscr{F}$ is the space of differentiable functions of the interval $I \subseteq \mathbb{R}$ to $\mathbb{K}$. Now consider the evaluation of Φ at $(x_1, \ldots, x_n)$ over $\mathscr{F}$, as described in section 2.1, but with the modification that we introduce new function variables $x_{2n+\theta_p+1}, \ldots, x_{2n+\theta_p+\mu_\sigma} \in \mathscr{F}$ whenever we execute an instruction P_p with $\sigma := \sigma_{P_p} \notin \Sigma^{\mathrm{ring}}$. Let $\delta_{p;} \in \mathscr{F}^{|D|}$ be the current state of the working space just before the execution of P_p. If $P_p = (v_i := v_j^{-1})$, then we define $x_{2n+\theta_p+1} := \delta_{p;j}^{-1}$. If $P_p = (v_i := v_j / v_k)$, then we take $x_{2n+\theta_p+1} := \delta_{p;k}^{-1}$. If $P_p = (v_i := \sigma(v_j))$ with $\sigma \in \Sigma^{\mathrm{special}}$, then we define

$$(x_{2n+\theta_p+1}, \ldots, x_{2n+\theta_p+\mu_\sigma}) := q_\sigma(\delta_{p;j}).$$

For $i = 1, \ldots, n$, we also define $x_{n+i} := x_i'$ and we claim that

$$(x_1', \ldots, x_{2n+\theta_{|P|}}') = \Psi(x_1, \ldots, x_{2n+\theta_{|P|}}).$$

We now evaluate $\Psi(x_1, \ldots, x_{2n+\theta_{|P|}})$ over $\mathscr{F}$. Let $\delta_{p;} \in \mathscr{F}^{|\tilde{D}|}$ be the state just before the execution of $\mathsf{Lift}\, P_p$. It will be convenient to abuse notation and use v as an abbreviation of the current state $\delta_{p;}$ in addition to the usual semantics as a tuple of variables. We claim that we always have $v_{2i+1} = v_{2i}'$, i.e. $V_i' = \dot{V}_i$, for all $i \in \mathbb{N}_{|D|}$. Just after copying the input values into the working space (and assuming that all auxiliary variables are initialized with zeros), the claim clearly holds. Assuming that the claim holds before the execution of $\mathsf{Lift}\, P_p$, we need to show that it still holds afterwards.

If $P_p = (v_i := \sigma(v_j))$ or $P_p = (v_i := \sigma(v_j, v_k))$ with $\sigma \in \Sigma^{\text{ring}}$, then $\mathsf{Lift}\, P_p$ is the same operation σ with arguments $V_i + \dot{V}_i \epsilon$, $V_j + \dot{V}_j \epsilon$, and possibly $V_k + \dot{V}_k \epsilon$ in $\mathbb{K}[\epsilon] / (\epsilon^2)$. Since i, j, k belong to $\mathbb{N}_{|D|}$, the claim holds after the execution of $\mathsf{Lift}\, P_p$.

If $P_p = (v_i := v_j^{-1})$, then $\mathsf{Lift}\, P_p$ corresponds to computing

$$\begin{aligned} \mathcal{O}_{p;0} &:= -\dot{V}_j \mathcal{I}_{p;0}^2 \\ V_i + \dot{V}_i \epsilon &:= \mathcal{I}_{p;0} + \mathcal{O}_{p;0} \epsilon. \end{aligned}$$

By definition, we have $\mathcal{I}_{p;0} = V_j^{-1}$ hence $V_i' = (V_j^{-1})'$. Therefore $\mathcal{O}_{p;0} = \mathcal{I}_{p;0}'$ after the execution, as claimed.

If $P_p = (v_i := v_j / v_k)$, then $\mathsf{Lift}\, P_p$ corresponds to computing

$$\begin{aligned} \mathcal{O}_{p;0} &:= -\dot{V}_k \mathcal{I}_{p;0}^2 \\ V_i + \dot{V}_i \epsilon &:= V_j \mathcal{I}_{p;0} + (V_j \mathcal{O}_{p;0} + \dot{V}_j \mathcal{I}_{p;0}) \epsilon. \end{aligned}$$

From $\mathcal{I}_{p;0} = V_k^{-1}$ we obtain $V_i' = (V_j / V_k)'$. Hence $\mathcal{O}_{p;0} = \mathcal{I}_{p;0}'$ and $V_j \mathcal{O}_{p;0} + \dot{V}_j \mathcal{I}_{p;0} = (V_j \mathcal{I}_{p;0})'$ after the execution, as claimed.

If $P_p = (v_i := \sigma(v_j))$ with $\sigma \in \Sigma^{\text{special}}$, then $\mathsf{Lift}\, P_p$ computes

$$\begin{aligned} \mathcal{O}_p &:= \dot{V}_j Q_\sigma(\mathcal{I}_p, V_j) \\ V_i + \dot{V}_i \epsilon &:= \mathcal{I}_{p;0} + \mathcal{O}_{p;0} \epsilon. \end{aligned}$$

Consequently, $\mathcal{O}_{p;0} = \mathcal{I}_{p;0}'$ and our claim again holds after the execution of $\mathsf{Lift}\, P_p$. This completes the proof of the first assertion.

Let us now turn to the second assertion about the computation of the initial value. The tuples $\bar{D}, \bar{I}, \bar{O}, \bar{P}$ are defined as follows:

- $\bar{D} \in \mathbb{K}^{|D| + n + \theta_{|P|} + |D^{\text{special}}|}$ with $\bar{D}_i := D_i$ for $i \in \mathbb{N}_{|D|}$ and $\bar{D}_{|D| + n + \theta_{|P|} + i} := D_i^{\text{special}}$ for $i \in \mathbb{N}_{|D^{\text{special}}|}$. Other entries of $\bar{D}$ may be filled arbitrarily;
- $\bar{I} := I$;
- $\bar{O} := (|D|, \ldots, |D| + n - 1) \bowtie O \bowtie (|D| + n, \ldots, |D| + n + \theta_{|P|} - 1)$;
- $\bar{P} := (v_{|D|} := v_{I_0}, \ldots, v_{|D| + n - 1} := v_{I_{n-1}}) \bowtie \bigotimes_{p \in \mathbb{N}_{|P|}} \mathsf{lift}\, P_p$, where lift is a function which maps instructions of P to tuples of instructions as follows. If $\sigma_{P_p} \in \Sigma^{\text{ring}}$, then we take $\mathsf{lift}\, P_p := (P_p)$. If $P_p = (v_i := v_j^{-1})$, then

$$\mathsf{lift}\, P_p := \big(v_{\bar{O}_{2n + \theta_p}} := v_j^{-1},\ v_i := v_{\bar{O}_{2n + \theta_p}}\big).$$

If $P_p = (v_i := v_j / v_k)$, then

$$\mathsf{lift}\, P_p := \big(v_{\bar{O}_{2n + \theta_p}} := v_k^{-1},\ v_i := v_j v_{\bar{O}_{2n + \theta_p}}\big).$$

If $P_p = (v_i := \sigma(v_j))$ with $\sigma \in \Sigma^{\text{special}}$, then

$$\mathsf{lift}\, P_p := \chi_\sigma(\bar{O}_{2n + \theta_p}, \ldots, \bar{O}_{2n + \theta_p + \mu_\sigma - 1}, j, |D| + n + \theta_{|P|}) \bowtie (v_i := v_{\bar{O}_{2n + \theta_p}}).$$

Let $(x_1, \ldots, x_n) \in \mathcal{F}^n$ be a solution of (1) for the initial condition $(x_1, \ldots, x_n)(\tau) = \xi$ and let $x_{n+1}, \ldots, x_{2n + \theta_{|P|}}$ be defined as above. Let $\psi \colon \mathbb{K}^{|I|} \longrightarrow \mathbb{K}^{|\tilde{I}|}$ stand for the map defined by the SLP $(\bar{D}, \bar{I}, \bar{O}, \bar{P})$. By construction, $\psi(\xi) = \xi \bowtie \Phi(\xi) \bowtie (x_{2n+1}', \ldots, x_{2n + \theta_{|P|}}')(\tau)$.

As for the complexity, we note that $\theta_{|P|} = O(|P|)$ and that $|D^{\text{special}}|$ is a constant. The constructions of $\tilde{D}, \tilde{I}, \tilde{O}, \tilde{P}$ and $\bar{D}, \bar{I}, \bar{O}, \bar{P}$ take linear time. □

Remark 10. In more precise complexity models such as computation trees, RAM machines [3, 7], or many-tape Turing machines [17], the complexity bound of Theorem 9 becomes $O((|D|+|P|+n)\log(|D|+|P|+n))$, due to the logarithmic space overhead for storing the indices of variables. On the other hand, the algorithm behind Theorem 9 uses no operations in $\mathbb{K}$, no hash tables, and no other complex operations on integers.

3.3. Exploiting FMA instructions

Modern computers often support an efficient so-called "Fused Multiply-Add" (FMA) instruction set over floating point numbers in single and double precisions. For Intel™ and ARM™ CPUs and also NVidia™ GPUs, these instructions compute expressions of the form $\pm a b \pm c$ as fast as a single product. This motivates us to define the *FMA signature* Σ^{fma} for a ring $\mathbb{A}$ as Σ^{ring}, together with the following operations:

- the fused multiply-add, defined by $\mathsf{fma}_{\mathbb{A}}: (a, b, c) \mapsto a b + c$,
- the fused multiply-sub, defined by $\mathsf{fms}_{\mathbb{A}}: (a, b, c) \mapsto a b - c$,
- the fused negative multiply-add, defined by $\mathsf{fnma}_{\mathbb{A}}: (a, b, c) \mapsto -a b + c$,
- the fused negative multiply-sub, defined by $\mathsf{fnms}_{\mathbb{A}}: (a, b, c) \mapsto -a b - c$,
- the negative multiply, defined by $\mathsf{nmul}_{\mathbb{A}}: (a, b, c) \mapsto -a b$.

We revisit Theorem 9 with FMA signatures.

THEOREM 11. *By allowing the SLPs for Φ, Ψ, ψ to be of signature Σ^{fma}, Theorem 9 holds with the sharper bound*

$$|\tilde{P}| \leqslant 3|P|_{\Sigma^{\text{fma}}} + 5|P|_{\{\mathsf{inv}, /\}} + \sum_{\sigma = \sigma_{P_p} \in \Sigma^{\text{special}}} (|\mathsf{X}_\sigma| + \mu_\sigma + 2). \qquad (6)$$

Proof. The construction of the SLPs extends the one presented in the proof of Theorem 9, by taking FMA operations into account. On the one hand, we define

$$\begin{aligned}
\mathsf{Lift}(v_i := v_j v_k) &:= (A := V_j \dot{V}_k,\ \dot{V}_i := \dot{V}_j V_k + A,\ V_i := V_j V_k)\\
\mathsf{Lift}(v_i := -v_j v_k) &:= (A := V_j \dot{V}_k,\ \dot{V}_i := -\dot{V}_j V_k - A,\ V_i := -V_j V_k)\\
\mathsf{Lift}(v_i := v_j v_k + v_l) &:= (A := V_j \dot{V}_k + \dot{V}_l,\ \dot{V}_i := \dot{V}_j V_k + A, V_i := V_j V_k + V_l)
\end{aligned}$$

and similarly for fms, fnma, and fnms, hence the contribution $3|P|_{\Sigma^{\text{fma}}}$ in the complexity bound. One the other hand, we redefine

$$\mathsf{Lift}(v_j := v_j^{-1}) := (V_i := \mathscr{I}_{p;0},\ \mathscr{O}_{p;0} := -\dot{V}_j V_i, \mathscr{O}_{p;0} := \mathscr{O}_{p;0} V_i,\ \dot{V}_i := \mathscr{O}_{p;0})$$

and

$$\begin{aligned}
\mathsf{Lift}(v_i := v_j / v_k) := (&\mathscr{O}_{p;0} := -\dot{V}_k \mathscr{I}_{p;0},\ \mathscr{O}_{p;0} := \mathscr{O}_{p;0} \mathscr{I}_{p;0},\\
&A := V_j \mathscr{O}_{p;0},\ \dot{V}_i := \dot{V}_j \mathscr{I}_{p;0} + A, V_i := V_j \mathscr{I}_{p;0}),
\end{aligned}$$

hence the contribution $5|P|_{\{\mathsf{inv}, /\}}$ in the complexity bound. □

Remark 12. In fact, the factor 3 in $3|P|_{\Sigma^{\text{fma}}}$ can often be further reduced. More precisely, let $|P|_{\text{cheap}}$ be the number of ring nodes of P whose ancestors are also all ring nodes. Then it can be verified that (after simplification of the SLP) the term $3|P|_{\Sigma^{\text{fma}}}$ can be replaced by $|P|_{\Sigma^{\text{fma}}} + 3(|P|_{\Sigma^{\text{fma}}} - |P|_{\text{cheap}})$.

4. IMPLEMENTATION

We implemented Theorem 11 within the JIL library [1, 12] (the present paper corresponds to GIT version `58a6ba506f8216260f6b8e8d28cc4311ddcfd8e7`). Currently, we support FMA instructions and Σ^{special} consists of all elementary functions exp, log, $\sqrt{}$, cos, sin, tan, acos, asin, atan, cosh, sinh, tanh, acosh, asinh, atanh. JIL is a free software C++ library for HPC computations with SLPs. It provides a convenient interface for automatic differentiation, common sub-expression elimination, lifting, transposition, and much more. It also comes with a JIT (Just In Time) compiler dedicated to SLPs. This compiler can generate executable machine code directly in memory, which is useful when the same SLP needs to be evaluated $N \gg 1$ times in an efficient manner.

The JIT compiler of JIL has the advantage of being one or two orders of magnitude faster than general purpose compilers such as GCC or CLANG. Consequently, JIT compilation becomes beneficial for relatively small values of N like $N \approx 1000$. In addition, if we have several competing SLPs for the same task, then we may efficiently generate machine code for each of them and empirically select the best SLP.

The source code of our polynomialization can be found in the file `src/ode/slp_ode_polynomialize.cpp` of JIL. Examples are in the `check/ode/` sub-directory. Usual simplifications of SLPs in JIL are done using the `simplify` function (implemented in the file `src/transforms/slp_simplify.cpp`): it folds constants, removes unneeded computations, and shares common subexpressions. The expected running time is linear, under the standard assumption that underlying hash tables are well balanced. Operations in $\mathbb{K}$ are only performed during constant folding [12, section 4]. Therefore, the number of operations in $\mathbb{K}$ never exceeds the number of instructions of the SLP under simplification.

Roughly speaking, the overhead 3 in the term $3|P|_{\Sigma^{\text{fma}}}$ of (6) is due to evaluating Φ over $\mathbb{K}[\epsilon]/(\epsilon^2)$. In order to decrease this overhead, we first present an optimization which aims at reducing the number of first order derivatives x_i' used by Ψ: we will separate ring instructions needed by divisions and special functions and other operations.

Example 13. Consider the system

$$x_1' = x_2, \qquad x_2' = x_2^2 - \frac{1}{1 + x_1^2}.$$

We represent it using the following SLP with signature Σ^{fma}: $D \in \mathbb{K}^6$ with $D_5 := 1$, $I := (0,1)$, $O := (1,4)$, and $P := (v_2 := v_0 v_0 + 1, v_3 := v_2^{-1}, v_4 := v_1 v_1 - v_3)$. The direct application of Theorem 11 leads to an SLP for a polynomialization Ψ with 5 inputs and outputs and 10 instructions, which drops to 7 after simplification. It turns out that this SLP is suboptimal since we may simply define $x_3 := (1 + x_1^2)^{-1}$ and obtain the following shorter polynomialization:

$$x_1 = x_2, \quad x_2' = x_2^2 - x_3, \quad x_3' = -2 x_3^2 x_2 x_1, \tag{7}$$

which can be represented by an SLP with 5 instructions.

In the above example, the second derivatives of x_1 and x_2 are not needed, because x_3 only depends on x_1, whose first derivative is available in the SLP of Φ before computing x_3'. We may compute the shorter polynomialization in a more systematic manner using the following optimization.

OPTIMIZATION 14. *Consider the SLP of Φ. If $x_i' := \Phi_i(x_1, \ldots, x_n)$ is computed before an instruction P_k with $\sigma_{P_k} \notin \Sigma^{\mathrm{fma}}$ depends on x_i, then we may modify the polynomialization Ψ constructed in Theorems 9 and 11 as follows:*

- *Just after $\Phi_i(x_1, \ldots, x_n)$ is computed, we copy its value into the position of x_{n+i} in the working space;*
- *The output corresponding to x_{n+i}' is discarded in the SLP of Ψ.*

In order to implement Optimization 14 efficiently, we need to solve the following problem: for a given subset $\mathscr{P}$ of instructions of the SLP of Φ (namely the P_p with $\sigma_{P_p} \notin \Sigma^{\mathrm{fma}}$) and for each variable x_i, we wish to determine the first instruction in $\mathscr{P}$ that depends on x_i. This is a common task for compilers, which can be achieved in linear time using one forward and one backward pass through the SLP.

More precisely, during the forward pass, one maintains an array that, for each variable v_i, indicates the (index of the) last instruction that modifies v_i. This information is used during the same pass to build an array that, for each argument v_{j_l} of an instruction $P_k = (v_i := \sigma(v_{j_1}, \ldots, v_{j_{|\sigma|}}))$, indicates the latest instruction that modified v_{j_l}. During the backward pass, for each instruction $P_k = (v_i := \sigma(v_{j_1}, \ldots, v_{j_{|\sigma|}}))$ and each input variable v_i, we determine the first instruction in $\mathscr{P}$ that depends on v_i.

Remark 15. Permuting independent instructions of the SLP might allow to reduce the dependencies and apply Optimization 14 more aggressively. However, finding the best permutation might be rather expensive, so we left this issue for future investigation.

Example 16. If $\Phi(X_1) = X_1^{-1}$, then Optimization 14 yields the simplified polynomialization $\Psi(X_1, X_2) = (X_2, -X_2^3)$ in terms of the SLP defined by $\tilde{D} \in \mathbb{K}^2$, $\tilde{I} := (0, 1)$, $\tilde{O} := (1, 2)$, and $\tilde{P} := (v_2 := v_1 v_1, v_2 := -v_2 v_1)$.

Example 17. If $\Phi(X_1, X_2) = (X_2, X_2 / X_1)$, then Optimization 14 yields the simplified polynomialization $\Psi(X_1, X_2, X_3) = (X_2, X_2 X_3, -X_2 X_3^2)$ in terms of a SLP with 2 instructions.

Example 18. If we swap the variables and equations of Example 17, then $\Phi(X_1, X_2) = (X_1 / X_2, X_1)$ and Optimization 14 yields the simplified polynomialization $\Psi(X_1, X_2, X_3, X_4) = (X_1 X_4, X_1, X_1 X_4, -X_3 X_4^2)$ which takes 3 instructions, but x_2' remains an input of Ψ. This example motivates our next optimization.

OPTIMIZATION 19. *Let the notation be as in Theorem 9. If $i \in \{1, \ldots, n\}$ and $j \in \{n+1, \ldots, 2n + \theta_{|P|}\}$ are such that $\Psi_i = \Psi_j$ and $\psi_i = \psi_j$ then we may simplify the polynomialization into $\tilde{\Psi}(X_1, \ldots, X_{j-1}, X_{j+1}, \ldots, X_{2n+\theta_{|P|}}) := (\Psi_1, \ldots, \Psi_{j-1}, \Psi_{j+1}, \ldots, \Psi_{2n+\theta_{|P|}})\,(X_1, \ldots, X_{j-1}, X_i, X_{j+1}, \ldots, X_{2n+\theta_{|P|}})$ and $\tilde{\psi}(X_1, \ldots, X_n) := (\psi_1, \ldots, \psi_{j-1}, \psi_{j+1}, \ldots, \psi_{2n+\theta_{|P|}})$.*

Example 20. In Example 18 the values of x_2'' and x_1' coincide. As for the initial conditions, we have $\psi(X_1, X_2, T) := (X_1 / X_2, X_1, X_1)$. Following Optimization 19, we may simplify the polynomialization into $\tilde{\Psi}(X_1, X_2, X_3) = (X_1 X_3, X_1, -X_1 X_3^2)$.

In order to minimize the number of extra variables in our polynomializations, we may compute the values of $x_1', \ldots, x_n'$ using Φ and $x_{2n+1}, \ldots, x_{2n+\theta_{|P|}}$ before evaluating Ψ. For sure, we obtain a polynomialization with only $n + \theta_{|P|}$ variables, still in linear time, which is in general higher than the one obtained in Theorems 9 and 11. Nevertheless, sometimes and thanks to the simplification routine of JIL, this approach turns out to be useful. The construction summarizes as follows.

OPTIMIZATION 21. *Let the notation be as in Theorem 9 and its proof. A polynomialization of Φ with $n+\theta_{|P|}$ inputs can be obtained as follows:*

1. *Compute an SLP $\tilde{\Phi}$ over $\mathbb{K}$ with signature in Σ^{ring} which takes $x_1,\ldots,x_n,x_{2n+1},\ldots,x_{2n+\theta_{|P|}}$ as input and returns $\Phi(x_1,\ldots,x_n)$.*
2. *Construct the SLP of the polynomialization $\tilde{\Psi}$ which takes $x_1,\ldots,x_n,x_{2n+1},\ldots,x_{2n+\theta_{|P|}}$ as input and returns $(\Psi_1,\ldots,\Psi_n,\Psi_{2n+1},\ldots,\Psi_{2n+\theta_{|P|}})(x_1,\ldots,x_n,\tilde{\Phi}_1(x_1,\ldots,x_n,x_{2n+1},\ldots,x_{2n+\theta_{|P|}}),\ldots,$ $\tilde{\Phi}_n(x_1,\ldots,x_n,x_{2n+1},\ldots,x_{2n+\theta_{|P|}}),x_{2n+1},\ldots,x_{2n+\theta_{|P|}})$.*
3. *As for the initial conditions for $\tilde{\Psi}$, construct the SLP for $\tilde{\psi}(x_1,\ldots,x_n):=(\tilde{\psi}_1,\ldots,\tilde{\psi}_n,\tilde{\psi}_{2n+1},\ldots,$ $\tilde{\psi}_{2n+\theta_{|P|}})(x_1,\ldots,x_n)$.*

In our implementation, we compare the SLP resulting of Optimizations 14 and 19 with the one of Optimizations 21 and 19: we return the one which has the smallest number of instructions. In case of equality, we return the one with the smallest number of variables.

Example 22. Optimization 21 applied to Example 18 yields a simplified polynomialization with 3 inputs and 2 instructions.

Example 23. Let $\varphi(X_1)=1+1/(1+X_1^2)$ and $\Phi:=\varphi\circ\varphi\circ\varphi$. Evaluating Φ takes 12 instructions. Optimizations 21 and 19 yield a polynomialization with 15 instructions, while Optimizations 14 and 19 yield 16 instructions.

5. CONCLUSION

We have shown how the polynomialization of an ODE given by an SLP can be achieved in linear time. Combined with suitable optimizations, our implementation returns efficient ODEs, especially for numerical integration.

Quadratization seems to be a harder problem for SLPs. A first straightforward approach is to expand the SLP into a sparse polynomial and apply the algorithms from [5]. However, this approach destroys the SLP structure. In particular, occasional products of linear forms are all expanded. We expect that more natural quadratizations exist for SLPs. Finding such SLPs with a better complexity is an interesting problem for future investigation.

BIBLIOGRAPHY

[1] A. Ahlbäck, J. van der Hoeven, and G. Lecerf. JIL: a high performance library for straight-line programs. `https://sourcesup.renater.fr/projects/jil`, 2025.

[2] J. Alexandre Dit Sandretto and A. Chapoutot. Validated explicit and implicit Runge–Kutta methods. *Reliable Computing (electronic edition)*, 22:79–113, 2016.

[3] P. Bürgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*, volume 315 of *Grundlehren der Mathematischen Wissenschaften*. Springer-Verlag, 1997.

[4] J. C. Butcher. *Numerical methods for ordinary differential equations*. Wiley, Chichester, third edition, 2016.

[5] A. Bychkov, O. Issan, G. Pogudin, and B. Kramer. Exact and optimal quadratization of nonlinear finite-dimensional nonautonomous dynamical systems. *SIAM J. Appl. Dyn. Syst.*, 23(1):982–1016, 2024.

[6] Y. Cai and G. Pogudin. Dissipative quadratizations of polynomial ODE systems. In B. Finkbeiner and L. Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2024*, volume 14571 of *Lect. Notes Comput. Sci.*, pages 323–342. Springer, Cham, 2024.

[7] J. von zur Gathen and J. Gerhard. *Modern computer algebra*. Cambridge University Press, New York, 3rd edition, 2013.

[8] M. Hemery, F. Fages, and S. Soliman. On the complexity of quadratization for polynomial differential equations. In A. Abate, T. Petrov, and V. Wolf, editors, *Computational Methods in Systems Biology. CMSB 2020*, volume 12314 of *Lect. Notes Comput. Sci.*, pages 120–140. Springer, Cham, 2020.

[9] M. Hemery, F. Fages, and S. Soliman. Compiling elementary mathematical functions into finite chemical reaction networks via a polynomialization algorithm for ODEs. In E. Cinquemani and L. Paulevé, editors, *Computational Methods in Systems Biology. CMBS 2021*, volume 12881 of *Lect. Notes Comput. Sci.*, pages 74–90. Springer, Cham, 2021.

[10] J. van der Hoeven. Relax, but don't be too lazy. *J. Symbolic Comput.*, 34:479–542, 2002.

[11] J. van der Hoeven. *The Jolly Writer. Your Guide to GNU TeXmacs*. Scypress, 2020.

[12] J. van der Hoeven and G. Lecerf. Towards a library for straight-line programs. Technical Report, HAL, 2025. https://hal.science/hal-05075591. Accepted for publication in *Appl. Algebra Eng. Commun. Comput.*

[13] J. van der Hoeven, G. Lecerf, and A. Minondo. Static bounds for straight-line programs. Technical Report, HAL, 2025. https://hal.science/hal-05105518.

[14] J. van der Hoeven et al. GNU TeXmacs. https://www.texmacs.org, 1998.

[15] B. Kramer and G. Pogudin. Discovering Polynomial and Quadratic Structure in Nonlinear Ordinary Differential Equations. Technical Report 2502.10005, arXiv, 2025. https://arxiv.org/abs/2502.10005.

[16] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. SIAM Press, 2009.

[17] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.