Computation of higher order Lie derivatives on the Infinity Computer

F. Iavernaro[a], F. Mazzia[b,*], M.S. Mukhametzhanov[c,d], Ya.D. Sergeyev[c,d]

[a]*Dipartimento di Matematica, Università degli Studi di Bari Aldo Moro, Italy*
[b]*Dipartimento di Informatica, Università degli Studi di Bari Aldo Moro, Italy*
[c]*DIMES, Università della Calabria, Italy*
[d]*IITMM, Lobachevsky State University of Nizhni Novgorod, Russia*

Abstract

In this paper, we deal with the computation of Lie derivatives, which are required, for example, in some numerical methods for the solution of differential equations. One common way for computing them is to use symbolic computation. Computer algebra software, however, might fail if the function is complicated, and cannot be even performed if an explicit formulation of the function is not available, but we have only an algorithm for its computation. An alternative way to address the problem is to use automatic differentiation. In this case, we only need the implementation of the algorithm that evaluates the function in terms of its analytic expression in a programming language, but we cannot use this if we have only a compiled version of the function. In this paper, we present a novel approach for calculating the Lie derivative of a function, even in the case where its analytical expression is not available, that is based on the Infinity Computer arithmetic. A comparison with symbolic and automatic differentiation shows the potentiality of the proposed technique.

Keywords: Lie derivatives, Ordinary differential equations, Multi-derivative methods, Derivatives computation, Numerical infinitesimals, Infinity Computer.
2010 MSC: 65L06, 65D25

1. Introduction

Lie derivatives play an important role in many mathematical and physical problems [1]. In particular, many methods in nonlinear control and system theory require the computation of Lie derivatives [2]. In this paper, we restrict our attention to the use of Lie derivatives in numerical methods for the solution of differential equations

$$y(t)' = f(y(t)), \qquad y(t_0) = y_0, \tag{1}$$

where $f : \mathbb{R}^m \to \mathbb{R}^m, m \geq 1$, is a $C^l, l \geq 1$, function on its domain and $y_0 \in \mathbb{R}^m$ is assigned. To get an intuitive understanding, Lie derivatives can be considered as total derivatives of certain fields along the solution of a differential equation. One of the main motivations for such an interest is the recent introduction of a class of multiderivative methods in the context of geometric integration [3, 4, 5]. When applied to canonical Hamiltonian problems, these methods have been

---

*Corresponding author
Email addresses:* `felice.iavernaro@uniba.it` (F. Iavernaro), `francesca.mazzia@uniba.it` (F. Mazzia), `m.mukhametzhanov@dimes.unical.it` (M.S. Mukhametzhanov), `yaro@dimes.unical.it` (Ya.D. Sergeyev)

shown to be conjugate-symplectic up to order $p + 2$, where p denotes the order of convergence. This property makes them eligible for the numerical simulation of Hamiltonian problems over long times, provided that a reliable and efficient technique to compute the derivatives of the function $f(y(t))$ is available, where $y(t)$ is implicitly defined by the differential problem itself (see (1)).

Two standard approaches to attack the problem of determining Lie derivatives in an automatic procedure make use of *Symbolic Differentiation* and *Automatic Differentiation*. The former exploits expression manipulation in computer algebra systems and is available in many problem solving environment such as Mathematica, Maxima, Maple and Matlab. Computer algebra software, however, could fail if the function is complicated and could not even be performed if an explicit analytic formulation of the function is not available at all.

In contrast, Automatic Differentiation (AD) can successfully operate also in the case where the function is defined through an algorithm implemented in a suitable programming language but would be useless if only a compiled version of the source code is available. AD refers to a family of techniques that compute derivatives through accumulation of values during code execution to generate derivative evaluations rather than derivative expressions. This, in general, allows estimations of derivatives whose accuracy is close to the underlying machine precision.

The website `http://www.autodiff.org` reports updated informations about papers and software available for AD. We refer to [6] for a survey on Automatic Differentiation in Machine Learning, and to [7] for a benchmark of selected algorithmic differentiation tools in the same field of research. Here the authors compare hand-derivative computation, finite differences, two symbolic differentiation tools, and 11 automatic differentiation tools written in different languages. The analyzed problems are simple objective functions from computer vision and machine learning. Simple means that there are no iterative loops and conditional statements are encapsulated in functions such as `abs`, `log`, `sum` or `exp`. Many problems in machine learning are of this form because the objective function should be handled efficiently by the differentiation tool.

We stress that many of the existing computational platforms are not equipped with functions for computing Lie derivatives. One toolbox that allows the computation of Lie derivatives is ADOL-C, a c++ package for automatic differentiation of algorithms written in C/C++ [8, 9, 10].

Additional techniques are those based on dual numbers [11, 12] and hyperdual numbers theory [13], that allow the computation of the first derivative and of the second derivative respectively, and the ones based on complex and multicomplex numbers [14]. This latter can be used for computing derivatives of any order [14].

A framework that allows an automatic and easy computation of derivatives is provided by the Infinity Computer, a computational platform able to handle infinite and infinitesimal numbers and to execute operations on them. Interestingly, the differentiation tool devised inside this environment can successfully be applied even to black-box functions, namely functions whose analytical expression may be not accessed. In other words, the function f may be given by a code or formula which are unknown to the user. He/she provides an argument y and obtains a result $f(y)$ without any knowledge about how this result has been obtained. As was emphasized above, in this case the user cannot calculate exact derivatives either analytically or symbolically (see [3, 15, 16, 17, 18]).

This tool has been theoretically analysed in [18] and extended to Lie derivatives in [15, 16, 17, 19] where Taylor methods for solving (1) have been proposed and in [3] where the class of multiderivative Euler-MacLaurin methods has been analyzed.

The Infinity Computer is based upon a positional numeral system with the infinite radix ① (called *grossone*) representing the number of elements of the set of natural numbers $\mathbb{N}$. For a complete description of the related theory we refer the reader to [20, 21, 22, 23, 24, 25].

A number C in this system, called *grossnumber*, is a linear combination of powers of ① of the

2

form

$$C = c_{p_m} ① ^{p_m} + \ldots + c_{p_1} ① ^{p_1} + c_{p_0} ① ^{p_0} + c_{p_{-1}} ① ^{p_{-1}} + \ldots + c_{p_{-l}} ① ^{p_{-l}}, \tag{2}$$

where all numerals $c_i \neq 0$ belong to a traditional numeral system and are called *grossdigits*, while numerals p_i, that may be finite, infinitesimal or infinite, are sorted in the decreasing order

$$p_m > p_{m-1} > \ldots > p_1 > p_0 > p_{-1} > \ldots p_{-(l-1)} > p_{-l},$$

with $p_0 = 0$ and called *grosspowers*. A grossnumber has a finite part if the grossdigit associated with the grosspower p_0 is different from zero (see [25, 26] and the patents [27] for details).

Terms having finite positive grosspowers represent the simplest infinite parts of C. Analogously, terms having negative finite grosspowers represent the simplest infinitesimal parts of C. For example, the number $C_1 = 2①^{-1} + 3.54①^{-2}$ represents only infinitesimal parts, the number $C_2 = -5.34 \cdot 10^{-3}①^2 + 2.77 \cdot 10^5①^1$ represent only infinite parts, while their sum $C_3 = C_2 + C_1 = -5.34 \cdot 10^{-3}①^2 + 2.77 \cdot 10^5①^1 + 2①^{-1} + 3.54①^{-2}$ brings both infinite and infinitesimal quantities. A number is called purely finite if it does not contain infinitesimal parts. Hereinafter, for the sake of simplicity, we will call these numbers just finite. In positional notation, a grossnumber takes the form

$$C = c_{p_m} ① ^{p_m} \ldots c_{p_1} ① ^{p_1} c_{p_0} ① ^{p_0} c_{p_{-1}} ① ^{p_{-1}} \ldots c_{p_{-l}} ① ^{p_{-l}}.$$

So, for example, the number C_3 is represented as $C_3 = (-5.34 \cdot 10^{-3})①^2 2.77 \cdot 10^5①^1 2①^{-1} 3.54①^{-2}$.

As we will see, for the computation of the derivative, the simplest grossnumber used is $①^{-1}$, which corresponds to evaluate the Lie derivative using forward finite difference with an infinitesimal step.

Besides the computation of derivatives, the ①-based methodology has been successfully applied in several areas of Mathematics and Computer Science: e.g., in optimization (see [28, 29, 30, 31, 32, 33, 34]) and going through infinite series (see, e.g., [25, 35]), in modeling and numerical simulation ([3, 15, 36, 37]), fractals and cellular automata (see [38, 39]), the first Hilbert problem and Turing machines (see [25, 40]), infinite decision making processes, game theory, and probability (see [41, 42, 43, 44]), etc.

The paper is organized as follows. In Section 2, we give a brief review about the technique used for the computation of standard derivatives of a function $y(t)$ using the Infinity Computer. Section 3 focuses on the computation of Lie derivatives, and a new technique is introduced and analyzed. Finally, in Section 4, the techniques presented are compared with algorithms based on symbolic differentiation and automatic differentiation using Matlab.

2. Differentiation techniques using the Infinity Computer

The computation of the derivatives using the Infinity Computer has been first derived in [18]. It is worth noting that, since the Infinity Computer works numerically (not symbolically), the algorithm that performs the evaluation of the derivative only needs an implemented version of the function $y = g(x)$ as input. In fact, we will see that a suitable numerical evaluation of this function on the Infinity Computer will allow the users to get the requested evaluation of $g'(x)$ with an accuracy close to the unit roundoff. For completeness we report the related theorem whose proof may be found in [18]:

Theorem 1. *Assume that:*

(i) for a function $g(x)$ evaluated by a procedure implemented on the Infinity Computer there exists an unknown Taylor expansion in a finite neighborhood $\delta(u)$ of a finite point u;

(ii) $g(x), g'(x), g''(x), \cdots, g^{(l)}(x)$ *assume finite values or are equal to zero for finite* $x \in \delta(u)$;

(iii) $g(x)$ *has been evaluated at a point* $u + ①^{-1} \in \delta(u)$.

Then the Infinity Computer returns the result of this evaluation in the positional numeral system with the infinite radix $①$ *in the following form*

$$g(u + ①^{-1}) = c_0 ①^0 c_1 ①^{-1} c_2 ①^{-2} \ldots c_{(l-1)} ①^{-(l-1)} c_l ①^{-l} \ldots,$$

where

$$g(u) = c_0, \quad g'(u) = c_1, \quad g''(u) = 2!c_2, \quad \ldots, \quad g^{(l)}(u) = l!c_l.$$

To better understand this result, we report an example where the function $g(x)$ is evaluated, using an infinitesimal step, in $g(x + ①^{-1})$. All the arithmetic operations are performed using the arithmetic that works on grossnumbers. For simplicity, we choose the simple polynomial function

$$g(u) = u^4 + 2u.$$

To compute the derivatives of g at a finite point x, we perform on the Infinity Computer arithmetic the evaluation of $g(u)$ at $x + ①^{-1}$:

$$g(x + ①^{-1}) = (x + ①^{-1})(x + ①^{-1})(x + ①^{-1})(x + ①^{-1}) + 2(x + ①^{-1}),$$

that is

$$x^4 + 4x^3①^{-1} + 6x^2①^{-2} + 4x①^{-3} + ①^{-4} + 2x + 2①^{-1} = (x^4 + 2x)①^0(4x^3 + 2)①^{-1}(6x^2)①^{-2}(4x)①^{-3}①^{-4}.$$

It is clear that the derivatives of the function are contained in the coefficients of the Taylor expansion of g. Observe that the computation is performed numerically and not symbolically, so the obtained results are exact up to machine precision.

Moreover, it should be stressed that this procedure is completely different from the one related to the dual and hyper-dual numbers and to multi-complex number. In facts, using the Infinity Computer it is possible to compute derivatives of any order. Other examples related to the computation of the derivatives could be found in [18].

3. Computation of Lie derivatives

Some classes of methods for the solution of the differential equation (1) use Lie derivatives at each step. A relevant example is the class of Hermite–Obreshkov (HO) linear multistep methods [45], which take the form

$$\sum_{i=0}^{k} \alpha_i y_{n+i} = \sum_{j=1}^{l} h^j \sum_{i=0}^{k} \beta_{ji} y_{n+i}^{(j)}, \tag{3}$$

where $y_{n+i}^{(j)}$ denotes an approximation to the j-th derivative of the solution $y(t)$ at t_{n+i}, with $t_{n+i} = t_n + ih$, and is defined below at formula (5). Recently, we have analyzed four different one step ($k = 1$) HO methods: the Euler–Maclaurin methods, which are higher derivative collocation methods deriving their name from the well-known Euler–Maclaurin integration formula; the BS Hermite Obreckoff methods based on B-spline collocation [5]; the Multi-Derivative Midpoint and Trapezoidal methods, generalizations of the midpoint and the trapezoidal rule respectively based on the implicit and explicit Taylor expansions up to a given order [4].

Let us now remind that if the Initial Value Problem (IVP) is scalar and autonomous the first four derivatives of $y(t)$ in terms of the function f are:

$$
\begin{aligned}
y'(t) &= f(y(t)), \\
y''(t) &= f'(y(t))f(y(t)), \\
y'''(t) &= f''(y(t))(f(y(t))^2 + (f'(y(t)))^2 f(y(t)), \\
y^{(iv)}(t) &= f'''(y(t))(f(y(t)))^3 + 4f''(y(t))f'(y(t))(f(y(t)))^2 + (f'(y(t)))^3 f(y(t)).
\end{aligned}
\tag{4}
$$

If we apply a numerical method such as (3), the involved Lie derivatives are defined as

$$
y_n^{(j)} := D_{j-1}f(y_n), \quad j = 1, 2, \ldots, l,
\tag{5}
$$

where

- $D_0 = I$ is the identity operator;

- the operator $D_l f(u)$ is defined as the l-th total derivative of $f(y(t))$ computed at $y(t) = u$, assuming that $y(t)$ satisfies the differential equation in (1).

We have used the subscript to define this operator to avoid confusion with the same order classical derivative operator denoted by D^l. Of course, the two operators yield the same result when applied to the projection of the true solution $y(t)$ on the mesh points but, in general, they will differ.

The techniques for computing the derivatives based on symbolic or automatic differentiation can be used to obtain the desired result [9]. Recall that (see [46]) working with a system of differential equations, that is with $m > 1$ the analytical computation of the j-th derivative $y^{(j)}$ involves a tensor of order j and the computation becames, in general, more involved. For example,

$$
y''(t) = \frac{d}{dt}f(y(t)) = f'(y(t))\,f(y(t)),
$$

where $f'(y) := \frac{\partial f}{\partial y}$ is a $m \times m$ matrix (tensor of rank 2) denoting the Jacobian of f with respect to y. Analogously, for the third derivative we get

$$
y'''(t) = \frac{d}{dt}\left(f'(y(t))\,f(y(t))\right) = f''(y)(f(y), f(y)) + (f'(y(t)))^2 f(y(t)),
$$

where $f''(y) := \frac{\partial^2 f}{\partial y^2}(y(t))$ is a tensor of rank 3 and dimension $m \times m \times m$, that is a bilinear map. One further differentiation would yield

$$
y^{(iv)}(t) = f'''(y(t))(f(y(t)), f(y(t), f(y(t))) + 3f''(y(t))(y''(t), f(y(t))) + f'(y(t))y'''(t),
$$

where the third derivative $f'''(y) := \frac{\partial^3 f}{\partial y^3}(y(t))$ is a tensor of rank 4 and dimension $m \times m \times m \times m$, that is a trilinear map. Notice that the computation of $y^{(j)}$, $j \geq 1$ according to the formulae above, would require $O(m^j)$ function evaluations and as many multiplications. It turns out that for large m and j, methods based on such formule would be too time consuming to be practical.

The only technique that does not produce a huge increase in its computational complexity when evaluating Lie derivatives of increasing order is the one based on finite differences: this is the approach adopted on the Infinity Computer. In fact, this strategy only requires the evaluation of the function f in some specific values y_n thus avoiding the use of tensor matrices.

Two techniques for computing the l-th Lie derivative $y_n^{(l)}$ have been developed in [3, 16]. The first one is based on finite differences computed on approximations of $y(t)$ using infinitesimal steps: this strategy will be referred to as FDY. The second one is based on finite differences computed on approximations of $f(y(t))$ (strategy FDF). Here we propose a new technique that outperforms the previous ones, based on the explicit Taylor Method (ETM). Hereafter we illustrate the three possible approaches.

3.1. Strategies FDY and FDF

Strategy FDY was first proposed in [16] and consists in performing l infinitesimal steps starting at time t_n using the explicit Euler formula with stepsize $h = ①^{-1}$ as follows:

$$y_{n,1} = y_n + ①^{-1} f(y_n), \quad y_{n,2} = y_{n,1} + ①^{-1} f(y_{n,1}), \ldots, \quad y_{n,l} = y_{n,l-1} + ①^{-1} f(y_{n,l-1}).$$

Then, the values of the needed derivatives are obtained by means of the forward differences

$$F_{①-1}^l [y_{n,0}, y_{n,1}, \ldots, y_{n,l}] = \sum_{j=0}^{l} (-1)^j \binom{l}{j} y_{n,l-j}, \quad y_{n,0} = y_n, \tag{6}$$

as follows

$$y_i^{(l)} = D_{l-1} f(y_n) = \frac{F_{①-1}^l [y_{n,0}, y_{n,1}, \ldots, y_{n,l}]}{①^{-l}} + O(①^{-1}). \tag{7}$$

As was proven in [16], since the error of the approximation is $O(①^{-1})$, the finite part of the value

$$\frac{F_{①-1}^l [y_{n,0}, y_{n,1}, \ldots, y_{n,l}]}{①^{-l}}$$

gives the *exact* derivative $y_n^{(l)}$.

In strategy FDF finite differences are employed directly on the value of f as follows:

$$y_n^{(l)} = D_{l-1} f(y_n) = \frac{F_{①-1}^{l-1} [f(y_{n,0}), f(y_{n,1}), \ldots, f(y_{n,l-1})]}{①^{-(l-1)}} + O(①^{-1}). \tag{8}$$

In [3], it has been proved that formulae (7) and (8) are equivalent.

3.2. Strategy ETM

The new proposed strategy uses the explicit Taylor expansion for approximating the differential equation (1) and is based on the following result.

Theorem 2. *Let $f : \Omega \subset \mathbb{R}^m \to \mathbb{R}^m$, $m \geq 1$, be a C^l function on an open domain Ω and $u \in \Omega$ be assigned. Assume that:*

(a) *for the function $f(z)$ calculated by a procedure implemented on the Infinity Computer there exists an unknown Taylor expansion in a finite neighborhood $\delta(u)$ of the point u;*

(b) *all partial derivatives up to order k of f assume finite values or are equal to zero for finite $z \in \delta(u)$;*

(c) *all the Lie derivatives $D_{j-1} f(u)$, $j = 1, \ldots, l$, for some $l \geq 1$ are known.*

Then, given the truncated B-series expansion

$$B_l(u) = u + ①^{-1}f(u) + \frac{①^{-2}}{2}D_1f(u) + \cdots + \frac{①^{-(l)}}{l!}D_{l-1}f(u),$$

the coefficient of $①^{-l}$ in the Taylor expansion of $f(B_l(u))$ multiplied by $l!$ is equal to the Lie derivative $D_lf(u)$.

<u>Proof.</u> We first prove the assertion for $l = 1$. To simplify the notation, but without loss of generality, we consider $m = 1$. We have that $B_1(u) = u + ①^{-1}f(u)$. The Taylor expansion of $f(B_1(u))$ is

$$f(B_1(u)) = f(u + ①^{-1}f(u)) = f(u) + ①^{-1}\frac{\partial f}{\partial y}(u)f(u) + O(①^{-2}) = f(u) + ①^{-1}D_1f(u) + O(①^{-2}),$$

so the coefficient of $①^{-1}$ is $D_1f(u)$ and the theorem is stated. For $l = 2$ we obtain

$$f(B_2(u)) = f(u + ①^{-1}f(u) + \frac{①^{-2}}{2}D_1f(u)) =$$

$$f(u) + ①^{-1}\frac{\partial f}{\partial y}(u)f(u) + \frac{①^{-2}}{2}(\frac{\partial f}{\partial y}(u)D_1f(u) + \frac{\partial^2 f}{\partial y^2}(u)f^2(u))) + O(①^{-(3)}),$$

and again the coefficients associated with $①^{-1}$ and $①^{-2}$ are $D_1f(u)$ and $D_2f(u)/2$ respectively. To prove the results for a generic index l we need to apply the theory of B-series, since the Taylor expansion may be viewed as a special B-series method. The result is proved with similar arguments as in [46, Theorem 1.9, page 58]. □

Let us go back to the question of evaluating the Lie derivatives (5) needed by formula (3). The ETM computes the Lie derivative iteratively, at each step of the computational procedure, starting from $D_1f(y_n)$.

The first step of the ETM is the same as the one resulting from the FDF approach: we perform one step of the explicit Euler method with stepsize $①^{-1}$, that corresponds to the first order explicit Taylor method; we truncate all the terms of order higher than one in $①^{-1}$; we get $D_1f(y_n)$ as the coefficient associated with the grosspower -1 of $f(B_1(y_n))$.

After this step $D_1f(y_n)$ is available, and hence it is possible to execute one step of the explicit Taylor method of order two using as integration step $①^{-1}$ and grosspowers up to -2. We obtain:

$$B_2(y_n) = y_n + ①^{-1}f(y_n) + \frac{①^{-2}}{2}D_1f(y_n).$$

Now given $f(B_2(y_n))$ by Theorem 2 we can compute $D_2f(y_n)$ as the coefficient of the grosspower -2 multiplied by 2.

More in general, the algorithm to obtain the derivative up to order l is the following. Starting from $f(y_n)$, for $j = 1,\ldots,l$, form the truncated B-series $B_j(y_n)$ of order j,

$$B_j(y_n) = y_n + \sum_{s=1}^{j}\frac{①^{-s}}{s!}D_{s-1}f(y_n).$$

Then compute the Taylor approximation of $f(B_j(y_n))$ using grosspowers up to $-j$. The coefficient of the grosspower $-j$ of this expansion multiplied by $j!$ is equal to the Lie derivative $D_jf(y_n)$.

The main advantage of ETM is that its computational cost is far lower than those resulting from the FDY and FDF strategies, for the following reasons:

7

1. We do not need to compute approximations of the solution at several mesh-points $t_n + j①^{-1}$, but we compute approximations of increasing order at $t_n + ①^{-1}$.

2. All the computations in (7) (strategy FDY) should be performed using the grosspowers up to $-l$. A slight improvement is yielded by strategy FDF, since formula (8) allows us to work with the numbers using only the grosspowers up to $-(l-1)$. In contrast, ETM allows us to use a variable number of grosspowers, starting from -1 and increasing the value for each new derivative computed, up to $-(l-1)$.

To better elucidate this aspect, let us consider the following example taken from [3, 17].

Example 1. *Let us consider the following initial value problem:*

$$\frac{dy}{dt} = f(t, y) := \frac{y - 2ty^2}{1 + t}, \quad y(t_0) = 0.4, \tag{9}$$

whose exact solution is

$$y(t) = \frac{1 + t}{2.5 + t^2}. \tag{10}$$

The problem has been rewritten as an autonomous system, but to simplify the notation we leave the dependence on t in the description. We would like to find the first 3 derivatives $D_1 f(t_0, y_0)$, $D_2 f(t_0, y_0)$ and $D_3 f(t_0, y_0)$ of the solution $y(t)$ at the point $t_0 = 0$. Differentiating (10) we get the exact values of these Lie derivatives:

$$D_1 f(t_0, y_0) = -0.32, \quad D_2 f(t_0, y_0) = -0.96, \quad D_3 f(t_0, y_0) = 1.536.$$

Now, let us find these derivatives using the FDY approach. First, we perform 3 iterations of the Euler method with integration step $①^{-1}$, truncating all values after the grosspower -4:

$$
\begin{aligned}
y_1 &= y_0 + ①^{-1} f(t_0, y_0) = 0.4 + 0.4①^{-1}, \\
y_2 &= y_1 + ①^{-1} f(t_0 + ①^{-1}, y_1) = 0.4 + 0.8①^{-1} - 0.32①^{-2} - 0.32①^{-3}, \\
y_3 &= y_2 + ①^{-1} f(t_0 + 2①^{-1}, y_2) = 0.4 + 1.2①^{-1} - 0.96①^{-2} - 1.92①^{-3} + 1.344①^{-4}, \\
y_4 &= y_3 + ①^{-1} f(t_0 + 3①^{-1}, y_3) = 0.4 + 1.6①^{-1} - 1.92①^{-2} - 5.76①^{-3} + 6.912①^{-4}.
\end{aligned}
$$

Applying formulae (7) and (6), we obtain

$$
\begin{aligned}
D_1 f(t_0, y_0) &\simeq ①^2 \cdot F^2_{①^{-1}}[y_0, y_1, y_2] = ①^2 \cdot (y_2 - 2y_1 + y_0) \\
&= ①^2 \cdot (0.4 + 0.8①^{-1} - 0.32①^{-2} - 0.32①^{-3} - 2(0.4 + 0.4①^{-1}) + 0.4) \\
&= -0.32 - 0.32①^{-1} = -0.32 + O(①^{-1}),
\end{aligned}
$$

$$
\begin{aligned}
D_2 f(t_0, y_0) &\simeq ①^3 \cdot F^3_{①^{-1}}[y_0, y_1, y_2, y_3] = ①^3 \cdot (y_3 - 3y_2 + 3y_1 - y_0) \\
&= ①^3 \cdot (0.4 + 1.2①^{-1} - 0.96①^{-2} - 1.92①^{-3} + 1.344①^{-4} \\
&\quad -3(0.4 + 0.8①^{-1} - 0.32①^{-2} - 0.32①^{-3}) + 3(0.4 + 0.4①^{-1}) - 0.4) \\
&= -0.96 + 1.344①^{-1} = -0.96 + O(①^{-1}),
\end{aligned}
$$

$$
\begin{aligned}
D_3 f(t_0, y_0) &\simeq ①^4 \cdot F^4_{①^{-1}}[y_0, y_1, y_2, y_3, y_4] = ①^4 \cdot (y_4 - 4y_3 + 6y_2 - 4y_1 + y_0) \\
&= ①^4 \cdot (0.4 + 1.6①^{-1} - 1.92①^{-2} - 5.76①^{-3} + 6.912①^{-4} \\
&\quad -4(0.4 + 1.2①^{-1} - 0.96①^{-2} - 1.92①^{-3} + 1.344①^{-4}) + \\
&\quad 6(0.4 + 0.8①^{-1} - 0.32①^{-2} - 0.32①^{-3}) - 4(0.4 + 0.4①^{-1}) + 0.4) \\
&= 1.536,
\end{aligned}
$$

8

from where we can extract the exact values of $D_1f(t_0, y_0)$, $D_2f(t_0, y_0)$ and $D_3f(t_0, y_0)$ as finite parts of $-0.32 - 0.32①^{-1}$, $-0.96 + 1.344①^{-1}$ and 1.536, respectively.

Let us now apply the FDF strategy. Here, we need to perform 3 steps of the Euler method, obtaining the values y_1, y_2 and y_3, truncating them after the grosspower -3, and then we evaluate the function f:

$$
\begin{aligned}
f(t_0, y_0) &= 0.4, \\
f(t_0 + ①^{-1}, y_1) &= 0.4 - 0.32①^{-1} - 0.32①^{-2}, \\
f(t_0 + 2①^{-1}, y_2) &= 0.4 - 0.64①^{-1} - 1.6①^{-2} + 1.344①^{-3}. \\
f(t_0 + 3①^{-1}, y_3) &= 0.4 - 0.96①^{-1} - 3.84①^{-2} + 5.568①^{-3}.
\end{aligned}
$$

Applying formulae (8), we obtain

$$
\begin{aligned}
D_1f(t_0, y_0) &\simeq ①^1 \cdot F^1_{①^{-1}}[f(t_0, y_0), f(t_0 + ①^{-1}, y_1)] \\
&= ① \cdot (f(t_0 + ①^{-1}, y_1) - f(t_0, y_0)) \\
&= ① \cdot (0.4 - 0.32①^{-1} - 0.32①^{-2} - 0.4) = -0.32 - 0.32①^{-1} \\
&= -0.32 + O(①^{-1}),
\end{aligned}
$$

$$
\begin{aligned}
D_2f(t_0, y_0) &\simeq ①^2 \cdot F^2_{①^{-1}}[f(t_0, y_0), f(t_0 + ①^{-1}, y_1), f(t_0 + 2①^{-1}, y_2)] \\
&= ①^2 \cdot (f(t_0 + 2①^{-1}, y_2) - 2f(t_0 + ①^{-1}, y_1) + f(t_0, y_0)) \\
&= ①^2 \cdot (0.4 - 0.64①^{-1} - 1.6①^{-2} + 1.344①^{-3} - 2(0.4 - 0.32①^{-1} - 0.32①^{-2}) + 0.4) \\
&= -0.96 + 1.344①^{-1} = -0.96 + O(①^{-1}),
\end{aligned}
$$

$$
\begin{aligned}
D_3f(t_0, y_0) &\simeq ①^3 \cdot F^3_{①^{-1}}[f(t_0, y_0), f(t_0 + ①^{-1}, y_1), f(t_0 + 2①^{-1}, y_2), f(t_0 + 3①^{-1}, y_3)] \\
&= ①^3 \cdot ((t_0 + 3①^{-1}, y_3) - 3f(t_0 + 2①^{-1}, y_2) + 3f(t_0 + ①^{-1}, y_1) - f(t_0, y_0)) \\
&= ①^3 \cdot (0.4 + 1.2①^{-1} - 0.96①^{-2} - 1.92①^{-3} \\
&\quad -3(0.4 + 0.8①^{-1} - 0.32①^{-2} - 0.32①^{-3}) + 3(0.4 + 0.4①^{-1}) - 0.4) \\
&= 1.536,
\end{aligned}
$$

from where we can again extract the exact values of $D_1f(t_0, y_0)$, $D_2f(t_0, y_0)$, and $D_3f(t_0, y_0)$ as finite parts of $-0.32 - 0.32①^{-1}$, $-0.96 + 1.344①^{-1}$, 1.536 respectively.

It should be noticed that the value y_3 cannot be truncated after the grosspower -3 using the first strategy, because the coefficient of $①^{-3}$ at the value y_3 is used also for computing $D_3f(t_0)$. On the contrary, the FDF method allows us to use the grosspowers up to -3, which decreases the computational cost of the procedures computing the $2-nd$ and the $3-rd$ Lie derivatives.

Let us now apply the ETM. Here, we compute three approximations of the solution at step $t_0 + ①^{-1}$ by exploiting the explicit Taylor method with increasing order. We get:

$$
\begin{aligned}
B_1(t_0, y_0) &= y_0 + f(t_0, y_0)①^{-1} = 0.4 + 0.4①^{-1}, \\
f(t_0 + ①^{-1}, B_1(y_0)) &= 0.4 - 0.32①^{-1},
\end{aligned}
$$

the first Lie derivative is computed as: $D_1f(t_0, y_0) = -0.32$;

$$
\begin{aligned}
B_2(t_0, y_0) &= y_0 + f(t_0, y_0)①^{-1} + \frac{1}{2}D_1f(t_0, y_0)①^{-2} \\
&= 0.4 + 0.4①^{-1} - 0.16①^{-2}, \\
f(t_0 + ①^{-1}, B_2(y_0)) &= 0.4 - 0.32①^{-1} - \frac{0.96}{2}①^{-2},
\end{aligned}
$$

the second Lie derivative is then computed as: $D_2 f(t_0, y_0) = -0.96$;

$$B_3(t_0, y_0) = y_0 + f(t_0, y_0)①^{-1} + \frac{1}{2} D_1 f(t_0, y_0)①^{-2} + \frac{1}{6} D_2 f(t_0, y_0)①^{-3}$$
$$= 0.4 + 0.4①^{-1} - 0.16①^{-2} - 0.16①^{-3},$$
$$f(t_0 + ①^{-1}, B_3(y_0)) = 0.4 - 0.32①^{-1} - \frac{0.96}{2}①^{-2} + \frac{1.536}{6}①^{-3},$$

the third Lie derivative is finally obtained: $D_3 f(t_0, y_0) = 1.536$. We stress that now we have computed $D_j f(t_0, y_0)$ using grosspowers up to $-j$, for $j = 1, 2, 3$.

Each technique, used in combination with a numerical method for ODEs, defines a new scheme based on finite and infinitesimal steps. In particular, numerical methods for the solution of the ODEs that use the infinitesimal steps, explicit Euler methods for the EMY and EMF strategies and explicit Tayor method for the ETM are used to compute the unknown values of the Lie derivatives.

4. Numerical illustrations

To compare the proposed techniques, we evaluate the Lie derivatives for the following differential problems of increasing dimension m: the nonlinear pendulum ($m = 2$), the Kepler problem ($m = 4$), the Argon problem ($m = 28$), the Brusselator problem ($m = 38$) and the Burgers' equation ($m = 38$). The first three problems have been analyzed in [47], and for the first two we have applied the multiderivative HO methods in [3, 4, 5]. The Brussellator and Burgers' problems are available in the Matlab ODE suite [48].

The strategies discussed in Section 3 have been implemented in a new Matlab class, using only grossnumbers defined by a finite expansion of integer grosspowers such as, for example,

$$X = ①^P \sum_{j=0}^{T} x_j ①^{-j}, \qquad \text{with grossdigits} \quad x_j \in \mathbb{R}, \tag{11}$$

where P and T are given positive integers. This restriction is sufficient for the purposes considered in the present work since the theory related to both derivatives and Lie derivatives is based on the Taylor expansion in finite (integer) powers of ①. The advantage is that this working assumption makes each floating-point operation on grossnumbers analogous to the corresponding one in the ring of polynomials, with a relevant reduction of the associated computational cost. As an example, for the two grossnumbers

$$X = ①^P(x_0 ①^0 + x_1 ①^{-1}), \quad Y = ①^P(y_0 ①^0 + y_1 ①^{-1} + y_2 ①^{-2}), \tag{12}$$

we get

$$X + Y = ①^P((x_0 + y_0)①^0 + (x_1 + y_1)①^{-1} + y_2 ①^{-2}),$$
$$X \cdot Y = ①^{2P}(x_0 y_0 ①^0 + (x_0 y_1 + x_1 y_0)①^{-1}$$
$$+ (x_0 y_2 + x_1 y_1)①^{-2} + x_1 y_2 ①^{-3}),$$

and analogously for the division X/Y. The same subclass of grossnumbers has been used in [49, 36] to implement a dynamic precision floating point arithmetic. For the moment, the Matlab class has been implemented without using the vectorization facility, thus requiring a loop for vectorial functions.[1]

---

[1]This question will be addressed in a future work.

The first test compares the three strategies presented in the previous section for the Infinity Computer, just to show that the new proposed method is more efficient independently on the size of the problem. The experiments have been performed on a computer with the Windows 10 operating system, an i7-8550U processor, 8 GB of RAM, and the Matlab version 2016b.
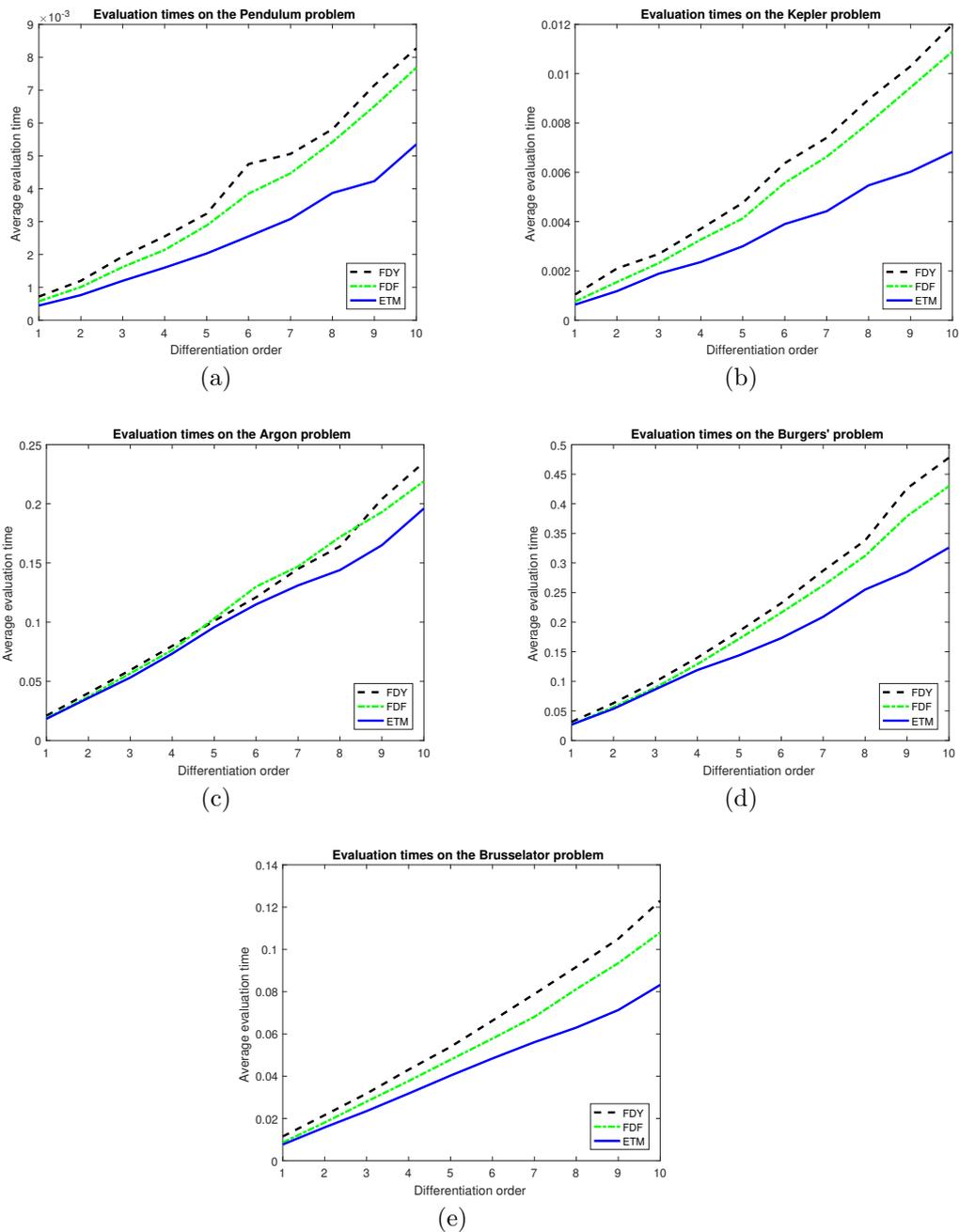

Figure 1: Average evaluation times for the computation of the Lie derivative of order $k, k = 1, \cdots, 10$ for the 5 test functions using the three strategies FDY, FDF and ETM.

In Figure 1, the graph of the average values of the execution times obtained after 50 iteration are presented for the three computational strategies on the Infinity Computer. It can be seen that the strategy FDF has a smaller execution time with respect to the strategy FDY in almost all the cases, while the strategy ETM has the smallest execution time in all the cases with respect to both FDY and FDF.

In the second series of experiments, the strategy ETM is compared with a couple of available tools for computing Lie derivatives. In more detail, we have considered using the symbolic Matlab toolbox and ADiGator, which is a Matlab toolbox for Automatic Differentiation (AD). This latter has shown very interesting potentialities with respect to other existing AD toolboxes (see [50, 51]). We have carried out the computation of the Lie derivatives according to (4) using these two toolboxes as follows. First, we have implemented formulae (4) for the computation of the derivatives of order up to 6. In the following we report the formulae for $m = 1$:

$$D_1 f(y_0) \;=\; f_y^{(1)}(y_0)f(y_0), \qquad D_2 f(y_0) = f_y^{(2)}(y_0)(f(y_0))^2 + f_y^{(1)}(y_0)D_1 f(y_0),$$

$$D_3 f(y_0) \;=\; f_y^{(3)}(y_0)(f(y_0))^3 + 3f_y^{(2)}(y_0)f(y_0)D_1 f(y_0) + f_y^{(1)}(y_0)D_2 f(y_0),$$

$$D_4 f(y_0) \;=\; f_y^{(4)}(y_0)(f(y_0))^4 + 6f_y^{(3)}(y_0)(f(y_0))^2 D_1 f(y_0) + 3f_y^{(2)}(y_0)(D_1 f(y_0))^2 +$$
$$+4f_y^{(2)}(y_0)f(y_0)D_2 f(y_0) + f_y^{(1)}(y_0)D_3 f(y_0),$$

$$D_5 f(y_0) \;=\; f_y^{(5)}(y_0)(f(y_0))^5 + 10f_y^{(4)}(y_0)(f(y_0))^3 D_1 f(y_0) + f_y^{(3)}(y_0)f(y_0)[10f(y_0)D_2 f(y_0)+$$
$$+15(D_1 f(y_0))^2] + 10f_y^{(2)}(y_0)D_1 f(y_0)D2 f(y_0) + 5f_y^{(2)}(y_0)f(y_0)D_3 f(y_0)+$$
$$+f_y^{(1)}(y_0)(D_4 f(y_0)),$$

$$D_6 f(y_0) \;=\; f_y^{(6)}(y_0)(f(y_0))^6 + 15f_y^{(5)}(y_0)(f(y_0))^4 D_1 f(y_0) + f_y^{(4)}(y_0)(f(y_0))^2[45(D_1 f(y_0))^2+$$
$$+20f(y_0)D_2 f(y_0)] + f_y^{(3)}(y_0)f(y_0)[60D_1 f(y_0)D_2 f(y_0) + 15f(y_0)D_3 f(y_0)]+$$
$$+15f_y^{(3)}(y_0)(D_1 f(y_0))^3 + 6f_y^{(2)}(y_0)f(y_0)D_4 f(y_0) + 15f_y^{(2)}(y_0)D_1 f(y_0)D_3 f(y_0)+$$
$$+10f_y^{(2)}(y_0)(D_2 f(y_0))^2 + f_y^{(1)}(y_0)D_5 f(y_0),$$

$$(13)$$

where $f_y^{(k)}(y_0) = \frac{\partial^k}{\partial y^k}f(y_0)$ denotes the k-th order partial derivatives of the function $f(y)$ evaluated at $y = y_0$. These formulae have been generalized for vectorial problems, and thus using tensors for representing the higher order partial derivatives of f with respect to y. With this implementation we experienced two drawbacks:

1) Due to the "*out of memory*" error, the symbolic computation is not able to generate the derivative of order higher than three for Burgers' problem and Brussellator and four for Argon problem, while ADiGator is not able to generate the codes that computes partial derivatives of order higher than five for Argon problem and three for Burgers' problem.

2) The evaluation of the Lie derivative generates an "*out of memory*" when we use symbolic computation for Lie derivatives higher than four for Argon problem and three for Brusselator and Burgers' problems. A similar problem is experienced when using ADiGator, which is not able to compute the solution for Lie derivatives of order higher than four for Argon and Brusselator problems and three for Burgers' problem. This issue is caused by the need of using an intermediate full multidimensional array in the computation (note that in Matlab it is not possible to define sparse multidimensional arrays).

We tried to solve the drawback 2) by exploiting the tensor toolbox available in [52], which allows one to work with sparse tensors. However, we experienced much higher computational costs and execution times, so we decided not to report here the obtained results.

The second type of implementation is based upon the following recurrence relation for the computation of Lie derivatives:

$$D_j f(y(t)) = \frac{\partial f}{\partial y} D_{j-1} f(y(t)) f(y(t)), \quad j \geq 1, \tag{14}$$

that requires, at each step, only the computation of the Jacobian of the Lie derivative obtained at the previous computed step. This relation is simpler from an implementation point of view, but the associated evaluation cost is higher, because we cannot reuse the information already obtained for evaluating the lower order derivatives. On the other hand, it requires less memory and so, for large-dimensional problems, this approach is preferable.

In the following, we denote by "Tensor (T)" the implementation described in (13) and by "Jacobian (J)" the one using the recurrence relation (14).

In Table 1, the obtained average execution times over 50 trials are presented for the five test problems: the first 6 Lie derivatives have been successfully computed by means of the Tensor method, while the Jacobian method was able to compute the first 9 Lie derivatives. We also report the times needed to create the higher order partial derivatives (SGT, AGT) and the Jacobian of the iterative formula (SGJ, AGJ). Concerning the symbolic computations, the expression of the Jacobian has been simplified by means of the Matlab command "`simplify`" for the Pendulum, Kepler, and Brusselator problems. For the Argon and Burgers' problems, the Jacobian was not simplified due to the "*out of memory*" errors during the execution of the command "`simplify`". We observe that, for the Argon, Burgers' and Brussellator problems, both methods fail due to a "*out of memory*" issue when the order of the Lie derivative increases.

In Figure 2, graphs of average evaluation times are presented for the 5 test problems and the first 10 derivatives computed by the symbolic toolbox, ADiGator and ETM on the Infinity Computer. One can see that the behaviour of the Tensor implementation and the Jacobian implementation is similar, but the Jacobian approach allows us to calculate the derivatives of a higher order. Moreover, the ADiGator has calculated the 3-rd and 4-th derivatives of the Brusselator problem using the Jacobian faster than using Tensors, since for this problem the sparsity pattern of the tensors was not duly exploited. An opposite behavior has been observed for the Pendulum problem, since this problem has a small dimension and formulae (13) allowed us to optimize the computations by using the results obtained in the previous iterations.

For the first two Lie derivatives, ADiGator performs better than the Infinity Computer, but as soon as the order of the Lie derivative is increased, the curve of the computational times grows faster for the algorithm based on symbolic computation and automatic differentiation so that the results become always in favour of the ETM. Keeping into account that the Infinity Computer simulator is not optimized, we think that this procedure for computing the Lie derivatives is promising, also considering that no computational time is required to preprocessing the data.

Observe that the Infinity Computer is able to construct the first 10 Lie derivatives of all 5 test problems showing a good computational efficiency even when the dimension of the problem is relatively high. For example, the Infinity Computer is able to calculate the 6-th Lie derivative for the Burgers' problem faster than what ADiGator does for the 3-rd derivative.

As we can see from the numerical tests, our implementation using ADiGator and symbolic computation is not able to produce results for problems of a relatively high dimension due to the "*out of memory*" errors, that is the code cannot be executed on the Matlab environment because requires more memory than the available one. A better implementation that exploits the sparsity pattern of the tensor or the Jacobian matrices could solve this issue, but it is important here to

13

Table 1: Times needed to generate the higher order partial derivatives and average evaluation times for the computation of the first 9 Lie derivatives of 5 test functions by means of the symbolic toolbox (rows SGT and SET using (13), SGJ and SEJ using (14)), by means of ADiGator (rows AGT and AET using (13), AGJ and AEJ using (14)) and by means of the strategy ETM on the Infinity Computer (rows IC). A star in the table means that the method is not able to end the computation.

Method	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9
Pendulum									
SGT	$9.57e-2$	$1.21e-1$	$1.21e-1$	$2.56e-1$	$3.79e-1$	$7.20e-1$			
SET	$6.94e-3$	$7.11e-3$	$7.71e-3$	$8.20e-3$	$9.91e-3$	$1.17e-2$			
AGT	$1.78e+0$	$3.50e+0$	$5.40e+0$	$7.23e+0$	$9.95e+0$	$1.09e+1$			
AET	$7.06e-5$	$1.20e-4$	$2.05e-4$	$4.09e-4$	$7.96e-4$	$1.59e-3$			
SGJ	$1.98e-1$	$1.15e-1$	$1.50e-1$	$1.73e-1$	$2.42e-1$	$3.06e-1$	$3.72e-1$	$4.52e-1$	$5.13e-1$
SEJ	$7.02e-3$	$7.17e-3$	$7.80e-3$	$8.14e-3$	$9.11e-3$	$9.89e-3$	$1.11e-2$	$1.18e-2$	$1.27e-2$
AGJ	$9.24e+0$	$3.77e+0$	$2.74e+0$	$2.45e+0$	$3.01e+0$	$3.92e+0$	$4.86e+0$	$7.06e+0$	$1.16e+1$
AEJ	$6.43e-5$	$1.87e-4$	$4.77e-4$	$1.02e-3$	$2.29e-3$	$4.22e-3$	$8.67e-3$	$1.65e-2$	$3.13e-2$
IC	$4.46e-4$	$7.65e-4$	$1.20e-3$	$1.60e-3$	$2.03e-3$	$2.55e-3$	$3.08e-3$	$3.87e-3$	$4.23e-3$
Kepler									
SGT	$3.37e-2$	$1.13e-1$	$2.28e-1$	$6.94e-1$	$2.44e+0$	$9.32e+0$			
SET	$1.14e-2$	$1.49e-2$	$2.56e-2$	$5.59e-2$	$1.33e-1$	$4.73e-1$			
AGT	$1.85e+0$	$3.79e+0$	$6.01e+0$	$9.37e+0$	$1.26e+1$	$1.95e+1$			
AET	$1.03e-4$	$2.77e-4$	$8.35e-4$	$2.32e-3$	$7.19e-3$	$2.30e-2$			
SGJ	$1.26e-1$	$2.39e-1$	$5.88e-1$	$1.49e+0$	$3.16e+0$	$6.11e+0$	$1.03e+1$	$1.68e+1$	$2.54e+1$
SEJ	$1.22e-2$	$1.32e-2$	$1.60e-2$	$2.30e-2$	$3.51e-2$	$4.97e-2$	$7.47e-2$	$1.39e-1$	$1.90e-1$
AGJ	$1.36e+0$	$2.01e+0$	$2.65e+0$	$3.45e+0$	$5.43e+0$	$1.02e+1$	$2.23e+1$	$4.76e+1$	$1.07e+2$
AEJ	$9.66e-5$	$4.23e-4$	$1.73e-3$	$4.18e-3$	$1.22e-2$	$2.93e-2$	$6.75e-2$	$1.59e-1$	$3.77e-1$
IC	$6.32e-4$	$1.18e-3$	$1.89e-3$	$2.36e-3$	$3.00e-3$	$3.90e-3$	$4.42e-3$	$5.47e-3$	$6.02e-3$
Argon									
SGT	$3.45e-1$	$4.66e+0$	$7.90e+1$	$2.88e+3$	$*$	$*$			
SET	$4.68e-1$	$2.96e+0$	$1.63e+1$	$9.97e+1$	$*$	$*$			
AGT	$2.72e+0$	$9.21e+0$	$2.22e+1$	$5.72e+1$	$2.46e+2$	$*$			
AET	$2.95e-3$	$9.63e-3$	$5.94e-2$	$1.08e+0$	$*$	$*$			
SGJ	$5.29e-1$	$1.38e+0$	$5.88e+0$	$*$	$*$	$*$	$*$	$*$	$*$
SEJ	$4.68e-1$	$3.07e+0$	$3.01e+1$	$*$	$*$	$*$	$*$	$*$	$*$
AGJ	$3.18e+0$	$7.23e+0$	$1.57e+1$	$4.13e+1$	$1.72e+2$	$*$	$*$	$*$	$*$
AEJ	$3.23e-3$	$1.66e-2$	$8.20e-2$	$5.99e-1$	$9.28e+0$	$*$	$*$	$*$	$*$
IC	$1.83e-2$	$3.59e-2$	$5.32e-2$	$7.34e-2$	$9.57e-2$	$1.15e-1$	$1.31e-1$	$1.44e-1$	$1.65e-1$
Burgers'									
SGT	$9.44e-1$	$2.44e+1$	$0.80e+3$	$*$	$*$	$*$			
SET	$4.65e-1$	$1.26e+1$	$3.32e+2$	$*$	$*$	$*$			
AGT	$3.28e+0$	$1.61e+1$	$6.23e+1$	$*$	$*$	$*$			
AET	$7.14e-3$	$3.05e-2$	$3.28e-1$	$*$	$*$	$*$			
SGJ	$8.80e-1$	$6.07e+0$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
SEJ	$4.58e-1$	$1.15e+1$	$*$	$*$	$*$	$*$	$*$	$*$	$*$
AGJ	$3.55e+0$	$1.20e+1$	$4.34e+1$	$*$	$*$	$*$	$*$	$*$	$*$
AEJ	$7.51e-3$	$4.75e-2$	$3.58e-1$	$*$	$*$	$*$	$*$	$*$	$*$
IC	$2.69e-2$	$5.36e-2$	$8.65e-2$	$1.19e-1$	$1.44e-1$	$1.73e-1$	$2.09e-1$	$2.55e-1$	$2.85e-1$
Brusselator									
SGT	$2.35e+0$	$6.54e+0$	$0.95e+2$	$*$	$*$	$*$			
SET	$8.65e-2$	$.18e-1$	$2.02e-1$	$*$	$*$	$*$			
AGT	$1.94e+0$	$4.64e+0$	$7.11e+0$	$1.03e+1$	$1.37e+1$	$1.77e+1$			
AET	$4.23e-4$	$1.30e-3$	$3.48e-2$	$2.11e+0$	$*$	$*$			
SGJ	$1.70e+0$	$2.90e+0$	$1.10e+1$	$4.79e+1$	$2.38e+2$	$1.15e+3$	$*$	$*$	$*$
SEJ.	$8.72e-2$	$1.07e-1$	$1.53e-1$	$3.35e-1$	$1.10e+0$	$4.19e+0$	$*$	$*$	$*$
AGJ	$1.43e+0$	$2.17e+0$	$3.24e+0$	$4.74e+0$	$6.61e+0$	$9.04e+0$	$1.34e+1$	$2.43e+1$	$6.90e+1$
AEJ	$4.79e-4$	$1.61e-3$	$4.51e-3$	$9.90e-3$	$1.99e-2$	$4.45e-2$	$1.23e-1$	$4.60e-1$	$2.21e+0$
IC	$7.68e-3$	$1.58e-2$	$2.35e-2$	$3.18e-2$	$4.03e-2$	$4.84e-2$	$5.61e-2$	$6.30e-2$	$7.13e-2$

consider that the ETM strategy only requires ℓ function evaluations in the Infinity Computer arithmetic, where ℓ is the order of the derivative.

Conclusion

A novel approach based on on the Infinity Computer arithmetic for calculating the Lie derivative of a function, even in the case where its analytical expression is not available, has been presented. A comparison with symbolic and automatic differentiation shows the poten-
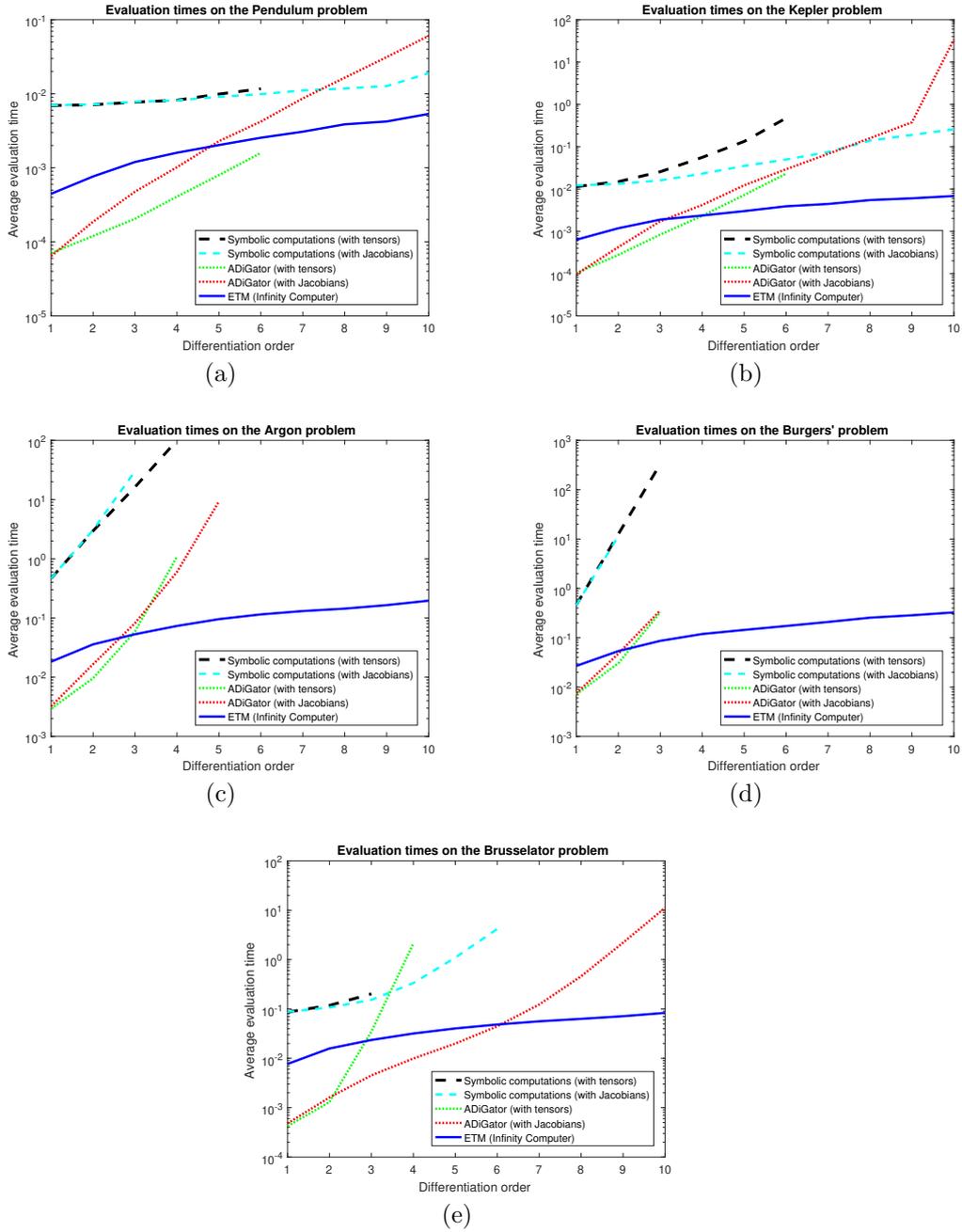
Figure 2: Average evaluation times for the computation of the first 10 Lie derivatives of 5 test functions (graphs (a)–(e), respectively) symbolically (black and cyan dashed lines), by ADiGator (green and red dotted lines) and by the strategy "ETM" on the Infinity Computer (blue line).

tiality of the proposed technique, especially for higher order derivatives and for problems of relatively high dimension. The presented strategy, in fact, only requires ℓ function evaluations in the Infinity Computer arithmetic, where ℓ is the order of the derivative. In a future research,

a more detailed comparison of the Infinity Computer with respect to different computational methodologies will be performed. The software simulator of the Infinity Computer will be optimized for this purpose using vector operations and, probably, parallel computations, allowing us to improve the computational efforts of the proposed methodology.

References

[1] J. M. Lee, Introduction to Smooth Manifolds, Vol. 218 of Graduate Texts in Mathematics, Springer, New York, 2006.

[2] A. Isidori, Nonlinear Control Systems, third edition, Communications and Control Engineering, Springer-Verlag London, 1995.

[3] F. Iavernaro, F. Mazzia, M. S. Mukhametzhanov, Y. D. Sergeyev, Conjugate-symplecticity properties of Euler-Maclaurin methods and their implementation on the infinity computer, Applied Numerical Mathematics. 155 (2020) 58–72. doi:10.1016/j.apnum.2019.06.011.

[4] F. Iavernaro, F. Mazzia, On conjugate-symplecticity properties of a multi-derivative extension of the midpoint and trapezoidal methods, Rendiconti del Seminario Matematico 76 (2) (2018) 123–134.

[5] F. Mazzia, A. Sestini, On a class of conjugate symplectic Hermite-Obreshkov one-step methods with continuous spline extension, Axioms 7 (3). doi:10.3390/axioms7030058.

[6] A. Günes Baydin, B. Pearlmutter, A. Andreyevich Radul, J. Mark Siskind, Automatic differentiation in machine learning: A survey, Journal of Machine Learning Research 18 (2018) 1–43.

[7] F. Srajer, Z. Kukelova, A. Fitzgibbon, A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning, Optimization Methods and Software 33 (4-6) (2018) 889–906. doi:10.1080/10556788.2018.1435651.

[8] K. Röbenack, J. Winkler, S. Wang, LIEDRIVERS- a toolbox for the efficient computation of lie derivatives based on the object-oriented algorithmic differentiation package ADOL-C, in: Proceedings of the 4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, EOOLT 2011, 2011, pp. 57–66.

[9] K. Röbenack, Computation of multiple Lie derivatives by algorithmic differentiation, Journal of Computational and Applied Mathematics 213 (2) (2008) 454–464. doi:10.1016/j.cam.2007.01.036.

[10] A. Walther, A. Griewank, Getting started with ADOL-C, in: U. Naumann, O. Schenk (Eds.), Combinatorial Scientific Computing, Chapman-Hall CRC Computational Science, 2012, Ch. 7, pp. 181–202.

[11] W. Yu, M. Blair, DNAD, a simple tool for automatic differentiation of fortran codes using dual numbers, Computer Physics Communications 184 (5) (2013) 1446–1452. doi:10.1016/j.cpc.2012.12.025.

[12] J. A. Fike, J. J. Alonso, Automatic differentiation through the use of hyper-dual numbers for second derivatives, in: S. Forth, P. Hovland, E. Phipps, J. Utke, A. Walther, T. J. Barth, M. Griebel, D. E. Keyes, R. M. Nieminen, D. Roose, T. Schlick (Eds.), Recent Advances in Algorithmic Differentiation, Vol. 87 of Lecture Notes in Computational Science and Engineering, Springer Berlin Heidelberg, 2012, pp. 163–173.

[13] J. Fike, J. Alonso, Automatic differentiation through the use of hyper-dual numbers for second derivatives, Lecture Notes in Computational Science and Engineering 87 LNCSE (2012) 163–173.

[14] G. Lantoine, R. Russell, T. Dargent, Using multicomplex variables for automatic computation of high-order derivatives, ACM Transactions on Mathematical Software 38 (3). doi:10.1145/2168773.2168774.

[15] P. Amodio, F. Iavernaro, F. Mazzia, M. S. Mukhametzhanov, Y. D. Sergeyev, A generalized Taylor method of order three for the solution of initial value problems in standard and infinity floating-point arithmetic, Mathematics and Computers in Simulation 141 (2017) 24–39.

[16] Y. D. Sergeyev, Solving ordinary differential equations by working with infinitesimals numerically on the infinity computer, Applied Mathematics and Computation 219(22) (2013) 10668–10681.

[17] Y. D. Sergeyev, M. S. Mukhametzhanov, F. Mazzia, F. Iavernaro, P. Amodio, Numerical methods for solving initial value problems on the Infinity Computer, International Journal of Unconventional Computing 12(1) (2016) 3–23.

[18] Y. D. Sergeyev, Higher order numerical differentiation on the Infinity Computer, Optimization Letters 5(4) (2011) 575–585.

[19] F. Mazzia, Y. D. Sergeyev, F. Iavernaro, P. Amodio, M. S. Mukhametzhanov, Numerical methods for solving ODEs on the Infinity Computer, in: Proc. of the 2nd Intern. Conf. "Numerical Computations: Theory and Algorithms", Vol. 1776, AIP Publishing, New York, 2016, p. 090033.

[20] Y. D. Sergeyev, Lagrange Lecture: Methodology of numerical computations with infinities and infinitesimals, Rendiconti del Seminario Matematico dell'Università e del Politecnico di Torino 68(2) (2010) 95–113.

[21] Y. D. Sergeyev, Independence of the grossone-based infinity methodology from non-standard analysis and comments upon logical fallacies in some texts asserting the opposite, Foundations of Science 24 (1) (2019) 153–170.

[22] G. Lolli, Metamathematical investigations on the theory of grossone, Applied Mathematics and Computation 255 (2015) 3–14.

[23] M. Margenstern, Using grossone to count the number of elements of infinite sets and the connection with bijections, p-Adic Numbers, Ultrametric Analysis and Applications 3 (3) (2011) 196–204.

[24] F. Montagna, G. Simi, A. Sorbi, Taking the Pirahã seriously, Communications in Nonlinear Science and Numerical Simulation 21 (1–3) (2015) 52–69.

[25] Y. D. Sergeyev, Numerical infinities and infinitesimals: Methodology, applications, and repercussions on two Hilbert problems, EMS Surveys in Mathematical Sciences 4(2) (2017) 219–320.

[26] Y. D. Sergeyev, Arithmetic of Infinity, Edizioni Orizzonti Meridionali, CS, 2003, 2nd ed. 2013.

[27] Y. D. Sergeyev, Computer system for storing infinite, infinitesimal, and finite quantities and executing arithmetical operations with them, USA patent 7,860,914, 2010.

[28] M. Cococcioni, M. Pappalardo, Y. D. Sergeyev, Lexicographic multi-objective linear programming using grossone methodology: Theory and algorithm, Applied Mathematics and Computation 318 (2018) 298–311. doi:10.1016/j.amc.2017.05.058.

[29] M. Cococcioni, A. Cudazzo, M. Pappalardo, Y. D. Sergeyev, Solving the lexicographic multi-objective mixed-integer linear programming problem using branch-and-bound and grossone methodology, Communications in Nonlinear Science and Numerical Simulation (2020) 84,105177doi:https://doi.org/10.1016/j.cnsns.2020.105177.

[30] R. De Leone, G. Fasano, Y. D. Sergeyev, Planar methods and grossone for the conjugate gradient breakdown in nonlinear programming, Computational Optimization and Applications 71 (2018) 73–93.

[31] M. Gaudioso, G. Giallombardo, M. S. Mukhametzhanov, Numerical infinitesimals in a variable metric method for convex nonsmooth optimization, Applied Mathematics and Computation 318 (2018) 312–320.

[32] L. Lai, L. Fiaschi, M. Cococcioni, Solving mixed pareto-lexicographic multi-objective optimization problems: The case of priority chains, Swarm and Evolutionary Computation (In Press).

[33] Y. D. Sergeyev, D. E. Kvasov, M. S. Mukhametzhanov, On strong homogeneity of a class of global optimization algorithms working with infinite and infinitesimal scales, Communications in Nonlinear Science and Numerical Simulation 59 (2018) 319–330.

[34] R. D. Leone, G. Fasano, M. Roma, Y. D. Sergeyev, Iterative grossone-based computation of negative curvature directions in large-scale optimization, Journal of Optimization Theory and Applications 186 (2), in press.

[35] A. Zhigljavsky, Computing sums of conditionally convergent and divergent series using the concept of grossone, Applied Mathematics and Computation 218(16) (2012) 8064–8076.

[36] P. Amodio, L. Brugnano, F. Iavernaro, F. Mazzia, On the use of the infinity computer architecture to set up a dynamic precision floating-point arithmetic, Soft Computing, in press.

[37] A. Falcone, A. Garro, M. S. Mukhametzhanov, Y. D. Sergeyev., Representation of grossone-based arithmetic in simulink for scientific computing, Soft Computing, submitted.

[38] F. Caldarola, The exact measures of the Sierpinski d-dimensional tetrahedron in connection with a diophantine nonlinear system, Communications in Nonlinear Science and Numerical Simulation 63 (2018) 228–238.

[39] L. D'Alotto, A classification of one-dimensional cellular automata using infinite computations, Applied Mathematics and Computation 255 (2015) 15–24.

[40] Y. D. Sergeyev, A. Garro, Single-tape and multi-tape Turing machines through the lens of the Grossone methodology, Journal of Supercomputing 65(2) (2013) 645–663.

[41] L. Fiaschi, M. Cococcioni, Numerical asymptotic results in game theory using Sergeyev's Infinity Computing, Int. Journal of Unconventional Computing 14 (1) (2018) 1–25.

[42] D. Rizza, A study of mathematical determination through Bertrand's Paradox, Philosophia Mathematica 26 (2018) 375–395.

[43] D. Rizza, Numerical methods for infinite decision-making processes, Int. Journal of Unconventional Computing 14 (2) (2019) 139–158.

[44] C. S. Calude, M. Dumitrescu, Infinitesimal probabilities based on grossone, SN Computer Science, 1 (2020) 36. doi:10.1007/s42979-019-0042-8.

[45] E. Hairer, S. P. Nørsett, G. Wanner, Solving ordinary differential equations. I. Nonstiff problems., Second ed., Springer Series in Computational Mathematics **8**, Springer-Verlag, Berlin, 1993.

[46] E. Hairer, C. Lubich, G. Wanner, Geometric Numerical Integration. Structure-Preserving Algorithms for Ordinary Differential Equations, Second ed., Springer, Berlin, 2006.

[47] L. Brugnano, F. Iavernaro, Line Integral Methods for Conservative Problems, Monographs and Research Notes in Mathematics, CRC Press, Boca Raton, FL, 2016.

[48] L. F. Shampine, M. W. Reichelt, The MATLAB ODE suite, SIAM Journal on Scientific Computing 18(1) (1997) 1–22. doi:10.1137/S1064827594276424.

[49] P. Amodio, L. Brugnano, F. Iavernaro, F. Mazzia, A dynamic precision floating-point arithmetic based on the Infinity Computer framework, Lecture Notes in Comput. Sci. 11974 (2020) 289–297. doi:`10.1007/978-3-030-40616-5_22`.

[50] M. A. Patterson, M. Weinstein, A. V. Rao, An efficient overloaded method for computing derivatives of mathematical functions in MATLAB, ACM Trans. Math. Softw. 39 (3) (2013) 17:1–17:36.

[51] M. Weinstein, A. Rao, Algorithm 984: ADiGator, a toolbox for the algorithmic differentiation of mathematical functions in MATLAB using source transformation via operator overloading, ACM Transactions on Mathematical Software 44 (2). doi:10.1145/3104990.

[52] B. W. Bader, T. G. Kolda, Efficient MATLAB computations with sparse and factored tensors, SIAM Journal on Scientific Computing 30 (1) (2007) 205–231. doi:10.1137/060676489.