Available online at www.sciencedirect.com
ScienceDirect
Journal of Computational and Applied Mathematics 213 (2008) 454–464
JOURNAL OF COMPUTATIONAL AND APPLIED MATHEMATICS
www.elsevier.com/locate/cam

Computation of multiple Lie derivatives by algorithmic differentiation

Klaus Röbenack

Department of Mathematics, Institute of Scientific Computing, Technische Universität Dresden, 01062 Dresden, Germany

Received 21 July 2006; received in revised form 10 January 2007

Abstract

Lie derivatives are often used in nonlinear control and system theory. In general, these Lie derivatives are computed symbolically using computer algebra software. Although this approach is well-suited for small and medium-size problems, it is difficult to apply this technique to very complicated systems. We suggest an alternative method to compute the values of iterated and mixed Lie derivatives by algorithmic differentiation.
© 2007 Elsevier B.V. All rights reserved.

MSC: 93B29; 93B40; 93C10

Keywords: Lie derivatives; Algorithmic differentiation; Series expansion

1. Introduction

Many methods in nonlinear control and system theory require Lie derivatives. The most well-known area of application is controller design by feedback linearization [16,23]. For nonlinear systems of moderate complexity, Lie derivatives can easily be computed symbolically using computer algebra packages such as Mathematica, Maple, MuPAD, or Macsyma. In fact, several research groups developed toolboxes to solve problems in nonlinear control symbolically using Lie derivatives [3,9,20,21]. Symbolically computed solutions provide insights into the structure of the solution, parameter dependencies, and can easily be exported as source code of the programming languages C and Fortran.

With todays fast computer systems, symbolic computation can be applied to more complicated system as in the past. However, the applicability of symbolic computation in nonlinear control is limited due to enormous increase of the sizes of higher order derivatives [17,18]. Moreover, the models of complex systems are usually generated by means of computer-aided methods. The resulting model is not an explicit formula but an algorithm that may contain subprocedures, loops and conditional statements. It is often not possible to carry out symbolic computation for this type of model.

The computation of Lie derivatives essentially means we have to calculate derivatives. For higher order derivatives, the computational effort basically increases exponentially due to product and chain rule. In this case, numeric differentiation by divided differences is not applicable due to cancellation and truncation errors. An alternative differentiation technique called *algorithmic* or *automatic differentiation* avoids these disadvantages (see [2,10,6] and references cited there).

E-mail address: klaus@roebenack.de.

0377-0427/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.cam.2007.01.036

Higher order Lie derivatives are defined by certain finite recursions. This allows a very elegant implementation for computer algebra systems. Unfortunately, this approach cannot directly be conferred to algorithmic differentiation, i.e., one has to develop a completely different computation method. In fact, this has already been done for iterated Lie derivatives, where all Lie derivatives are computed along the same vector field [27,26]. The algorithmic differentiation tools used there must provide Taylor arithmetic. With minor modifications, these approaches can also be used to calculate certain (very special) types of mixed Lie derivatives. However, the computation of arbitrary mixed Lie derivatives by algorithmic differentiation has not been addressed in the literature. In this paper, we propose an approach to overcome this problem.

The paper is structured as follows. In Section 2 we recall the definition of Lie derivatives and discuss the relation with dynamical systems. The differentiation technique used in this paper is explained in Section 3. In Section 4, algorithmic differentiation is employed to compute iterated and mixed Lie derivatives. Our approach is applied to an example in Section 5. Finally, the conclusions are given in Section 6.

2. Lie derivatives and dynamical systems

Consider a map $\boldsymbol{h} : \mathscr{M} \to \mathbb{R}^p$ and a vector field $\boldsymbol{f} : \mathscr{M} \to \mathbb{R}^n$ defined on an open subset $\mathscr{M} \subseteq \mathbb{R}^n$. We assume that all maps occurring in this paper are sufficiently smooth. Moreover, let $\boldsymbol{\varphi}_t^{\boldsymbol{f}}$ denote the flow of vector field $\boldsymbol{f}$ at time t. The Lie derivative of map $\boldsymbol{h}$ along the vector field $\boldsymbol{f}$ at the point $\boldsymbol{x} \in \mathscr{M}$ is defined by

$$L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}) = \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{h}(\boldsymbol{\varphi}_t^{\boldsymbol{f}}(\boldsymbol{x}))\bigg|_{t=0}. \tag{1}$$

This definition is very similar to the classical Lie derivative of a scalar field known from differential geometry [15]. Our definition can be regarded as a simultaneous Lie derivative of the p scalar fields $h_1, \ldots, h_p$ of which $\boldsymbol{h}$ consists of. This notation is occasionally used in control theory and especially useful for software implementations [21,24]. Note that the Lie derivative (2) of a vector-valued map $\boldsymbol{h}$ should not be confused with the Lie derivative of a vector field.

By the chain rule, the Lie derivative (1) can equivalently be written as

$$L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}'(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x}), \tag{2}$$

where $\boldsymbol{h}'$ denotes the Jacobian matrix of $\boldsymbol{h}$. The Lie derivative $L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x})$ has the same dimension as the map $\boldsymbol{h}$ itself. Therefore, we repeat this process. If we compute multiple Lie derivatives along the same vector field $\boldsymbol{f}$, we get the recursion

$$L_{\boldsymbol{f}}^k\boldsymbol{h}(\boldsymbol{x}) = \frac{\partial L_{\boldsymbol{f}}^{k-1}\boldsymbol{h}(\boldsymbol{x})}{\partial \boldsymbol{x}}\boldsymbol{f}(\boldsymbol{x}) \quad \text{with } L_{\boldsymbol{f}}^0\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}(\boldsymbol{x}). \tag{3}$$

For a further vector field $\boldsymbol{g} : \mathscr{M} \to \mathbb{R}^n$ we obtain mixed Lie derivatives such as

$$L_{\boldsymbol{g}}L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}) = \frac{\partial L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x})}{\partial \boldsymbol{x}}\boldsymbol{g}(\boldsymbol{x}). \tag{4}$$

Denoting the flow of the vector field $\boldsymbol{g}$ at time s by $\boldsymbol{\varphi}_s^{\boldsymbol{g}}$, this Lie derivatives can be written as

$$L_{\boldsymbol{g}}L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}) = \frac{\mathrm{d}^2}{\mathrm{d}t\,\mathrm{d}s}\boldsymbol{h}(\boldsymbol{\varphi}_t^{\boldsymbol{f}}(\boldsymbol{\varphi}_s^{\boldsymbol{g}}(\boldsymbol{x})))\bigg|_{t=s=0}. \tag{5}$$

Consider an autonomous state-space system

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}), \quad \boldsymbol{y} = \boldsymbol{h}(\boldsymbol{x}), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \in \mathscr{M}, \tag{6}$$

where we consider $\boldsymbol{y}$ as an output. The Lie derivative $L_{\boldsymbol{f}}\boldsymbol{h}$ is the time derivative of the output curve along the dynamics of (6), i.e.,

$$\dot{\boldsymbol{y}}(0)|_{(6)} = \frac{\mathrm{d}}{\mathrm{d}t}\boldsymbol{h}(\boldsymbol{\varphi}_t^{\boldsymbol{f}}(\boldsymbol{x}_0))\bigg|_{t=0} = \boldsymbol{h}'(\boldsymbol{x}_0)\boldsymbol{f}(\boldsymbol{x}_0) = L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}_0).$$

Table 1
Simply Maple implementation of Lie derivative (2)

```
with(linalg);
LieDerivative: = proc(f,h,x);
                 multiply(jacobian(h,x),f)
end proc;
```

If $\boldsymbol{f}$ and $\boldsymbol{h}$ are analytic, the output curve of (6) can be written as

$$\boldsymbol{y}(t)=\sum_{k=0}^{\infty} L_{\boldsymbol{f}}^{k}\boldsymbol{h}(\boldsymbol{x}_0)\frac{t^k}{k!}. \tag{7}$$

This expansion is called *Lie series* [13].

Many real-world systems can be modelled by a control system of the form

$$\dot{\boldsymbol{x}}=\boldsymbol{f}(\boldsymbol{x})+\sum_{i=1}^{m}\boldsymbol{g}_i(\boldsymbol{x})u_i,\quad \boldsymbol{y}=\boldsymbol{h}(\boldsymbol{x}),\quad \boldsymbol{x}(0)=\boldsymbol{x}_0\in\mathscr{M} \tag{8}$$

with additional vector fields $\boldsymbol{g}_1,\ldots,\boldsymbol{g}_m:\mathscr{M}\to\mathbb{R}^n$ and the control inputs $u_1,\ldots,u_m$. There are several approaches to extend the concept of the Lie series (7) to control system (8). In addition to the autonomous system's case, the output of (8) depends also on the input. We set $\boldsymbol{g}_0:=\boldsymbol{f}$ and $u_0:=1$, and define a sequence of integrals by

$$\xi_{i_0}(t)=\int_0^t u_i(\tau)\,\mathrm{d}\tau,\qquad \int_0^t \mathrm{d}\xi_{i_0}=\xi_{i_0}$$

and

$$\int_0^t \mathrm{d}\xi_{i_k}\cdots\mathrm{d}\xi_{i_0}=\int_0^t \mathrm{d}\xi_{i_k}(\tau)\int_0^\tau \mathrm{d}\xi_{i_{k-1}}\cdots\mathrm{d}\xi_{i_0}$$

with $i_0,\ldots,i_k=0,\ldots,m$. If all maps are analytic, the output curve of (8) can expressed by

$$\boldsymbol{y}(t)=\boldsymbol{h}(\boldsymbol{x}_0)+\sum_{k=0}^{\infty}\sum_{i_0,\ldots,i_k=0}^{m} L_{\boldsymbol{g}_{i_0}}\cdots L_{\boldsymbol{g}_{i_k}}\boldsymbol{h}(\boldsymbol{x}_0)\int_0^t \mathrm{d}\xi_{i_k}\cdots\mathrm{d}\xi_{i_0}. \tag{9}$$

Series (9) is called *Fliess–Chen-Series* [8,5]. In the series expansions (9) we require mixed Lie derivatives of the form $L_{\boldsymbol{g}_{i_1}}L_{\boldsymbol{g}_{i_2}}\cdots L_{\boldsymbol{g}_{i_k}}\boldsymbol{h}(\boldsymbol{x})$. This arbitrarily mixed Lie derivatives also occur in other applications. For instance, in telecommunications, signal processing and circuit design *Volterra-Series* [30,32] play an important role. In this case, the output of system (8) can also be expanded in terms of mixed Lie derivatives [16]. Similarly, these Lie derivatives are needed to determine the observation space of forced systems (8), see [23, Section 3.2].

For symbolic computation, Lie derivatives are computed according to (2). Table 1 shows a straightforward implementation in Maple using the linear algebra package `linalg`. For higher order Lie derivatives we simply apply the function `LieDerivative` repeatedly. For this recurrence, from a programmer's point of view it makes no difference whether the Lie derivatives are computed along the same vector field as in (3) or along different vector fields as in (4).

3. Algorithmic differentiation

In many areas of engineering and science one needs derivative values of given functions. Typical applications are optimization and sensitivity analysis. The most common way to compute derivatives is symbolic differentiation using computer algebra systems. The derivative of a composite function is calculated by a systematic application of elementary differentiations rules together with the chain rule. As a result, one gets a symbolic expression.

For very complicated systems, symbolic differentiation might fail, e.g., if the function under consideration is not given by an explicit expression but by an algorithm containing conditional statements, branches and subroutines.

Furthermore, the sizes the resulting symbolic expressions often increase significantly w.r.t. the order of the derivative. On the other hand, numeric differentiation by finite differences is not reliable for higher order derivatives.

The disadvantages of symbolic and numeric differentiation can be circumvented with so-called *algorithmic* or *automatic differentiation* [10]. Like symbolic differentiation, elementary differentiation rules are systematically applied in connection with the chain rule. All intermediate values are evaluated instantaneously at the specific argument, i.e., we always deal with floating point numbers instead of symbolic expressions.

The most important area, where algorithmic differentiation is currently applied, is optimization. There, the code describing the function is often very large, but one usually needs first and occasionally second order derivatives only. In nonlinear control, the situation is reciprocal. The functions describing the dynamical system are reasonable sized, but modern control algorithms require higher order derivatives [16,23].

For the computation of higher order derivatives we introduce the concept of Taylor arithmetic. Consider a curve

$$\boldsymbol{x}(t) = \boldsymbol{x}_0 + \boldsymbol{x}_1 t + \boldsymbol{x}_2 t^2 + \cdots + \boldsymbol{x}_d t^d + \mathcal{O}(t^{d+1}) \tag{10}$$

with $\boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n$ and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d \in \mathbb{R}^n$. We map this curve via a smooth map $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ into a curve

$$z(t) = \boldsymbol{F}(\boldsymbol{x}(t)) = z_0 + z_1 t + z_2 t^2 + \cdots + z_d t^d + \mathcal{O}(t^{d+1}) \tag{11}$$

with the Taylor coefficients $z_k = (1/k!)(\partial^k/\partial t^k)z(t)|_{t=0}$ for $k = 0, \ldots, d$. Clearly, each Taylor coefficient z_k depends on the coefficients $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k$ of the curve (10). The first Taylor coefficients of (11) are given by

$$z_0 = \boldsymbol{F}(\boldsymbol{x}_0) \quad \text{and} \quad z_1 = \boldsymbol{F}'(\boldsymbol{x}_0)\boldsymbol{x}_1. \tag{12}$$

The next Taylor coefficients cannot anymore be expressed in terms of matrix calculus. We will use the following tensor notation [11,27]. Consider a sequence $\boldsymbol{v}_1, \ldots, \boldsymbol{v}_k \in \mathbb{R}^n$ of k vectors with $\boldsymbol{v}_j = (v_{j,1}, \ldots, v_{j,n})^{\mathrm{T}}$ for $j = 1, \ldots, k$. With Einstein's summation convention we define

$$\boldsymbol{F}^{(k)}(\boldsymbol{x})\boldsymbol{v}_1\boldsymbol{v}_2 \cdots \boldsymbol{v}_k := \frac{\partial^k F_i(\boldsymbol{x})}{\partial x_{i_1}\partial x_{i_2} \cdots \partial x_{i_k}} v_{1,j_1} v_{2,j_2} \cdots v_{k,j_k}, \tag{13}$$

which is a multi-linear map. Using (13), the next Taylor coefficients of (11) can be written as

$$\begin{aligned}
z_2 &= \boldsymbol{F}'(\boldsymbol{x}_0)\boldsymbol{x}_2 + \tfrac{1}{2}\boldsymbol{F}''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_1,\\
z_3 &= \boldsymbol{F}'(\boldsymbol{x}_0)\boldsymbol{x}_3 + \boldsymbol{F}''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_2 + \tfrac{1}{6}\boldsymbol{F}'''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_1,\\
z_4 &= \boldsymbol{F}'(\boldsymbol{x}_0)\boldsymbol{x}_4 + \boldsymbol{F}''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_3 + \tfrac{1}{2}\boldsymbol{F}''(\boldsymbol{x}_0)\boldsymbol{x}_2\boldsymbol{x}_2 + \tfrac{1}{2}\boldsymbol{F}'''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_2 + \tfrac{1}{24}\boldsymbol{F}^{(4)}(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_1,\\
z_5 &= \boldsymbol{F}'(\boldsymbol{x}_0)\boldsymbol{x}_5 + \boldsymbol{F}''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_4 + \boldsymbol{F}''(\boldsymbol{x}_0)\boldsymbol{x}_2\boldsymbol{x}_3 + \tfrac{1}{2}\boldsymbol{F}'''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_3\\
&\quad + \tfrac{1}{2}\boldsymbol{F}'''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_2\boldsymbol{x}_2 + \tfrac{1}{6}\boldsymbol{F}^{(4)}(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_2 + \tfrac{1}{120}\boldsymbol{F}^{(5)}(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_1\boldsymbol{x}_1.
\end{aligned} \tag{14}$$

In the scalar-valued case, the formula for the Taylor coefficients goes back to Faa di Bruno [7]. Generalizations for vector-valued functions are given in [22,14].

Algorithmic differentiation allows an direct calculation of the Taylor coefficients z_k for $k = 0, \ldots, d$. In particular, an explicit computation of the high dimensional derivative tensors $\boldsymbol{F}^{(k)}(\boldsymbol{x}_0) \in \mathbb{R}^{m \times n^k}$ can be avoided. If the map $\boldsymbol{F}$ consists of a sequence of some elementary operations (e.g., the functions and operations defined in the C library `math.h`), we can propagate the Taylor coefficients through these operations. Some of the rules to compute the Taylor coefficients for elementary functions are given in Table 2, see [10,1]. Taylor arithmetic can easily be implemented using operator overloading techniques. The computation of these Taylor coefficients is supported by the algorithmic differentiation packages ADOL-C [11], TADIFF [1] and FADBAD + + [31].

In Table 2, the coefficients are basically computed recursively by certain convolutions. In some cases, the recurrence depends on previous results. Therefore, to compute a Taylor coefficient z_k we have to compute the previous coefficients $z_0, \ldots, z_{k-1}$. Hence, the computational effort to obtain all Taylor coefficients increases at most quadratically compared to the evaluation of the pure function value. Using fast convolution techniques, the number of operations can even be reduced from $\mathcal{O}(d^2)$ to $\mathcal{O}(d \log d)$, see [10, p. 219]. The very moderate computational effort results from the restriction

Table 2
Computation of Taylor coefficients

Operations	Taylor coefficients
$z = x \pm y$	$z_k = x_k + y_k$
$z = xy$	$z_k = \sum_{i=0}^{k} x_i y_{k-i}$
$z = x/y$	$z_k = (x_k - \sum_{i=1}^{k} y_i z_{k-i})/y_0, \quad y_0 \neq 0$
$z = \sqrt{x}$	$z_k = (x_k - \sum_{i=1}^{k-1} y_i z_{k-i})/2z_0, \quad k \geqslant 1, x_0 > 0, z_0 = \sqrt{x_0}$
$z = \exp(x)$	$z_k = (\sum_{i=0}^{k-1} (k-i) z_i x_{k-i})/k, \quad k \geqslant 1, z_0 = \exp(x_0)$
$z = \ln(x)$	$z_k = (x_k - \sum_{i=1}^{k-1} i z_i x_{k-i}/k)/z_0 \quad k \geqslant 1, x_0 > 0, z_0 = \ln(x_0)$

to univariate Taylor series. The calculation of multivariate series expansion is significantly more expensive, see [25]. In fact, some algorithmic differentiation tools use univariate Taylor series to compute multivariate derivatives [10,12].

4. Computation of Lie derivatives

We derive methods to compute multiple Lie derivatives by algorithmic differentiation. The calculation of iterated Lie derivatives is straightforward. The computation of arbitrarily mixed Lie derivatives is more complicated.

4.1. Iterated Lie derivatives

We explain the computation of iterated Lie derivatives $L_{\boldsymbol{f}}^k \boldsymbol{h}(\boldsymbol{x})$ of a map $\boldsymbol{h}$ along the vector fields $\boldsymbol{f}$. First, we calculate the Taylor coefficients $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d \in \mathbb{R}^n$ of the solution (10) of the initial value problem

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t)), \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \in \mathscr{M}. \tag{15}$$

To achieve this, we concurrently consider the mapping

$$\boldsymbol{z}(t) = \boldsymbol{f}(\boldsymbol{x}(t)) = \boldsymbol{z}_0 + \boldsymbol{z}_1 t + \boldsymbol{z}_2 t^2 + \cdots + \boldsymbol{z}_{d-1} t^{d-1} + \mathcal{O}(t^d)$$

with the Taylor coefficients $\boldsymbol{z}_0, \ldots, \boldsymbol{z}_{d-1} \in \mathbb{R}^n$. Since $\boldsymbol{x}(t) = \boldsymbol{\varphi}_t^{\boldsymbol{f}}(\boldsymbol{x}_0)$ is a solution of (15), we have $\boldsymbol{z} = \dot{\boldsymbol{x}}$. Comparing the coefficients of $\dot{\boldsymbol{x}}$ and $\boldsymbol{z}$ yields

$$\boldsymbol{x}_{k+1} = \frac{1}{k+1} \boldsymbol{z}_k \quad \text{for } k = 0, \ldots, d-1. \tag{16}$$

This relation can be used to compute the Taylor coefficients of (10) step-by-step. More precisely, we start with the given initial value $\boldsymbol{x}_0 \in \mathscr{M}$ and compute the function value $\boldsymbol{z}_0 = \boldsymbol{f}(\boldsymbol{x}_0)$. Using (16) we calculate the Taylor coefficient $\boldsymbol{x}_1 = \boldsymbol{z}_0$. Knowing $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ we apply the Taylor arithmetic to obtain the coefficient $\boldsymbol{z}_1$. Using (16) again, we get $\boldsymbol{x}_2 = \boldsymbol{z}_1/2$. We proceed until the last step is reached, where $\boldsymbol{z}_{d-1}$ is calculated from $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{d-1}$ by Taylor arithmetic and (16) yields $\boldsymbol{x}_d = \boldsymbol{z}_{d-1}/d$.

Knowing the Taylor coefficients $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_d$ of the curve (10), we apply Taylor arithmetic to the map $\boldsymbol{h}$ in order to calculate the Taylor coefficients of system's (6) output curve

$$\boldsymbol{y}(t) = \boldsymbol{h}(\boldsymbol{x}(t)) = \boldsymbol{y}_0 + \boldsymbol{y}_1 t + \boldsymbol{y}_2 t^2 + \cdots + \boldsymbol{y}_d t^d + \mathcal{O}(t^{d+1}). \tag{17}$$

Matching the coefficients of the series expansions (7) and (17) results in

$$L_{\boldsymbol{f}}^k \boldsymbol{h}(\boldsymbol{x}_0) = k! \boldsymbol{y}_k \quad \text{for } k = 0, \ldots, d, \tag{18}$$

by what we obtain the function values of the iterated Lie derivatives (3) at $\boldsymbol{x}_0$.

4.2. Mixed Lie derivatives

In this part we compute mixed Lie derivatives of the map $\boldsymbol{h}$ such as (4). Consider k vector fields $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k : \mathscr{M} \to \mathbb{R}^n$ with the associated flows $\boldsymbol{\varphi}_{s_1}^{f_1}, \ldots, \boldsymbol{\varphi}_{s_k}^{f_k}$. Repeating the limit (5) we get

$$L_{f_1} \cdots L_{f_k} \boldsymbol{h}(\boldsymbol{x}_0) = \frac{\partial^k}{\partial s_1 \cdots \partial s_k} \boldsymbol{h}(\boldsymbol{\varphi}_{s_k}^{f_k} \circ \cdots \circ \boldsymbol{\varphi}_{s_1}^{f_1}(\boldsymbol{x}_0))\bigg|_{s_1=\cdots=s_k=0} \tag{19}$$

for $\boldsymbol{x}_0 \in \mathscr{M}$. For each independent variable s_i the expression on the right-hand side is differentiated only once. Hence, limit (19) remains unchanged if we replace each flow $\boldsymbol{\varphi}_{s_i}^{f_i}$ by its first order approximation

$$\widehat{\boldsymbol{\varphi}}_{s_i}^{f_i}(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{f}_i(\boldsymbol{x}) s_i. \tag{20}$$

The concatenation of the approximate flows defines a vector-valued function

$$\boldsymbol{w}(s) \equiv \boldsymbol{w}(s_1, \ldots, s_k) = \widehat{\boldsymbol{\varphi}}_{s_k}^{f_k} \circ \cdots \circ \widehat{\boldsymbol{\varphi}}_{s_1}^{f_1}(\boldsymbol{x}_0) \tag{21}$$

having the multivariate series expansion

$$\boldsymbol{w}(s) = \boldsymbol{x}_0 + \sum_{i_1=1}^{k} \boldsymbol{w}_{i_1} s_{i_1} + \sum_{i_1=1}^{k} \sum_{i_2=i_1}^{k} \boldsymbol{w}_{i_1 i_2} s_{i_1} s_{i_2} + \cdots \tag{22}$$

with certain coefficient vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{11}, \boldsymbol{w}_{12}, \ldots \in \mathbb{R}^n$. Using the tensor notation introduced in Section 3, the first derivatives of $\boldsymbol{h}$ along (21) have the form

$$\frac{\partial}{\partial s_1} \boldsymbol{h}(\boldsymbol{w}(s))\bigg|_{s=0} = \boldsymbol{h}'(\boldsymbol{x}_0)\boldsymbol{w}_1, \tag{23a}$$

$$\frac{\partial^2}{\partial s_1 \partial s_2} \boldsymbol{h}(\boldsymbol{w}(s))\bigg|_{s=0} = \boldsymbol{h}'(\boldsymbol{x}_0)\boldsymbol{w}_{12} + \boldsymbol{h}''(\boldsymbol{x}_0)\boldsymbol{w}_1\boldsymbol{w}_2, \tag{23b}$$

$$\frac{\partial^3}{\partial s_1 \partial s_2 \partial s_3} \boldsymbol{h}(\boldsymbol{w}(s))\bigg|_{s=0} = \boldsymbol{h}'(\boldsymbol{x}_0)\boldsymbol{w}_{123} + \boldsymbol{h}''(\boldsymbol{x}_0)\boldsymbol{w}_1\boldsymbol{w}_{23} + \boldsymbol{h}''(\boldsymbol{x}_0)\boldsymbol{w}_2\boldsymbol{w}_{13} + \boldsymbol{h}''(\boldsymbol{x}_0)\boldsymbol{w}_3\boldsymbol{w}_{12} + \boldsymbol{h}'''(\boldsymbol{x}_0)\boldsymbol{w}_1\boldsymbol{w}_2\boldsymbol{w}_3. \tag{23c}$$

These derivatives could directly be computed with multivariate Taylor arithmetic (see [25]). However, we want to compute the derivatives in (23) using univariate Taylor series of a curve (17). The first derivative (23a) is the directed derivative of $\boldsymbol{h}$ in the direction $\boldsymbol{w}_1$. For $k = 1$, if we set $s_1 = t$ and $\boldsymbol{x}(t) = \boldsymbol{w}(t)$, the first Taylor coefficients of (17) yield the Lie derivative

$$\boldsymbol{y}_1 = \frac{\partial}{\partial t} \boldsymbol{h}(\boldsymbol{w}(t))\bigg|_{t=0} = \boldsymbol{h}'(\boldsymbol{x}_0)\boldsymbol{w}_1 = \boldsymbol{h}'(\boldsymbol{x}_0)\boldsymbol{f}_1(\boldsymbol{x}_0) = L_{f_1}\boldsymbol{h}(\boldsymbol{x}_0).$$

Now, we consider the second order derivative (23b). The expressions in (23b) are very similar to that of the univariate Taylor series, namely

$$\boldsymbol{y}_2 = \boldsymbol{h}'(\boldsymbol{x}_0)\boldsymbol{x}_2 + \tfrac{1}{2}\boldsymbol{h}''(\boldsymbol{x}_0)\boldsymbol{x}_1\boldsymbol{x}_1, \tag{24}$$

see (14). Unfortunately, the derivative tensor $\boldsymbol{h}''(\boldsymbol{x}_0)$ is combined with the same direction $\boldsymbol{x}_1$ twice instead of with two different directions as in (23b). Therefore, the coefficient of a second order univariate Taylor expansion cannot directly be used to compute the mixed derivative (23b). In particular, if we simply set $s_1 = s_2 = t$ and $\boldsymbol{x}(t) = \boldsymbol{w}(t, t)$, we obtain the Taylor coefficient (24) of (17) with $\boldsymbol{x}_1 = \boldsymbol{w}_1 + \boldsymbol{w}_2$ and $\boldsymbol{x}_2 = \boldsymbol{w}_{11} + \boldsymbol{w}_{12} + \boldsymbol{w}_{22}$, which is not the desired derivative (23b).

In theory, derivative (23b) could be calculated from linear combinations of the vector $\boldsymbol{y}_2$ for a set of appropriate Taylor coefficients of (22) as suggested in [10,12]. Unfortunately, this approach is not applicable because the Taylor

Table 3
Exponents for the computation of mixed derivatives

k	n_1	n_2	n_3	n_4	n_5	d
1	1					1
2	2	3				5
3	4	6	7			17
4	8	12	14	15		49
5	16	24	28	30	31	129

coefficients of (22) are defined (indirect) by the concatenation of flows, but are not given explicitly. However, if we set $s_1 = t^2$ and $s_2 = t^3$, the curve $\boldsymbol{x}(t) = \boldsymbol{w}(t^2, t^3)$ becomes

$$\boldsymbol{x}(t) = \boldsymbol{x}_0 + \boldsymbol{w}_1 t^2 + \boldsymbol{w}_2 t^3 + \boldsymbol{w}_{11} t^4 + \boldsymbol{w}_{12} t^5 + \boldsymbol{w}_{22} t^6 + \cdots$$

and the fifth order Taylor coefficient $\boldsymbol{y}_5$ of (17) yields the correct result (23b), see last line of (14). From (19) and (23b) we conclude $L_{f_1} L_{f_2} \boldsymbol{h}(\boldsymbol{x}_0) = \boldsymbol{y}_5$.

This approach can be generalized for arbitrary order k. Then, for appropriate positive integers $n_1, \ldots, n_k$ we have

$$\left.\frac{\partial^k}{\partial s_1 \cdots \partial s_k} \boldsymbol{h}(\boldsymbol{w}(s_1, \ldots, s_k))\right|_{s=0} = \left.\frac{1}{d!} \frac{\partial^d}{\partial t^d} \boldsymbol{h}(\boldsymbol{w}(t^{n_1}, \ldots, t^{n_k}))\right|_{t=0} \tag{25}$$

with $d = \sum_{i=1}^{k} n_i$. The right-hand side is the dth Taylor coefficient $\boldsymbol{y}_d$ of (17) with $\boldsymbol{x}(t) = \boldsymbol{w}(t^{n_1}, \ldots, t^{n_k})$. From (19) we get the desired Lie derivative

$$L_{f_1} \cdots L_{f_k} \boldsymbol{h}(\boldsymbol{x}_0) = \boldsymbol{y}_d. \tag{26}$$

The integers $n_1, \ldots, n_k$ can be found in advance. The first exponents are listed in Table 3. We see a significant increase in the order d of the derivative. The 49th order Taylor coefficient can still be computed with algorithmic differentiation. In case of the 129th Taylor coefficient we could expect accuracy problems.

We can summarize the computation of the kth mixed Lie derivative as follows:

(1) Initialization: $\boldsymbol{x}_0 \in \mathscr{M}$ is given, set $\boldsymbol{x}_i = \boldsymbol{0}$ for $i = 1, \ldots, d$.
(2) We compute a series expansion of the concatenation of approximate flows (20). For $i = 1, \ldots, k$ we do:
   (a) For given Taylor coefficients $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_{d-n_i}$ we use algorithmic differentiation to compute the Taylor coefficients $z_0, \ldots, z_{d-n_i}$ of $z(t) = \boldsymbol{f}_i(\boldsymbol{x}(t))$.
   (b) We update the Taylor coefficients of $\boldsymbol{x}$ according to $\boldsymbol{x}(t) \mapsto \widehat{\boldsymbol{\varphi}}^{f_i}_{t^{n_i}}(\boldsymbol{x}(t)) = \boldsymbol{x}(t) + \boldsymbol{f}_i(\boldsymbol{x}(t)) t^{n_i}$:

$$\begin{aligned} \boldsymbol{x}_{n_i} &= \boldsymbol{x}_{n_i} + z_0, \\ \boldsymbol{x}_{n_i+1} &= \boldsymbol{x}_{n_i+1} + z_1, \\ &\vdots \\ \boldsymbol{x}_d &= \boldsymbol{x}_d + z_{d-n_i}. \end{aligned}$$

(3) For given Taylor coefficients $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_d$ we use algorithmic differentiation to compute the Taylor coefficient $\boldsymbol{y}_d$ of (17). This is the mixed Lie derivative (26).

In the first part we start with the constant curve $\boldsymbol{x}(t) \equiv \boldsymbol{x}_0$. In each step of the second part we add the Taylor coefficients $\boldsymbol{f}_i(\boldsymbol{x}(t)) t^{n_i}$. In part (a) we calculate the series expansion of $\boldsymbol{f}_i(\boldsymbol{x}(t))$. The multiplication with t^{n_i} in part (b) corresponds to a shift of the Taylor coefficients by n_i positions. This implies that only the Taylor coefficients up to order $d - n_i$ are needed before the shift.

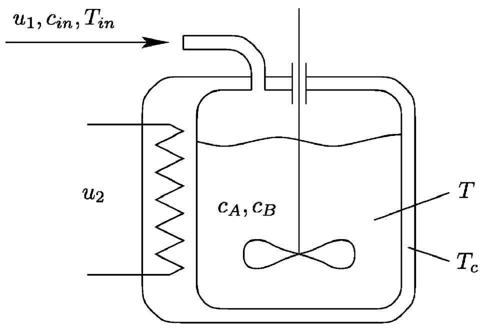

Fig. 1. Chemical reactor.

5. Example

We consider the well-studied nonlinear benchmark problem of a chemical reactor with the van der Vusse scheme $A \to B \to C, 2A \to D$ (see Fig. 1). The model

$$\begin{aligned}\dot{c}_A &= r_A(c_A, T) + (c_{\text{in}} - c_A)u_1,\\ \dot{c}_B &= r_B(c_A, c_B, T) - c_B u_1,\\ \dot{T} &= h(c_A, c_B, T) + \alpha(T_{\text{c}} - T) + (T_{\text{in}} - T)u_1,\\ \dot{T}_{\text{c}} &= \beta(T - T_{\text{c}}) + \gamma u_2 \end{aligned} \tag{27}$$

has been studied in [19,4,29,28]. System (27) comprises material and enthalpy balances, where c_A and c_B denote the concentrations of substance A and B, respectively. Moreover, T and T_{c} denote the temperatures in the reactor and in the cooling jacket, respectively. The reaction rates r_A and r_B and the contribution to the enthalpy h are described by

$$\begin{aligned}r_A(c_A, T) &= -c_A k_1(T) - c_A^2 k_2(T),\\ r_B(c_A, c_B, T) &= (c_A - c_B)k_1(T),\\ h(c_A, c_B, T) &= -\delta((c_A \Delta H_{AB} + c_B \Delta H_{BC})k_1(T) + c_A^2 \Delta H_{AD} k_2(T))\end{aligned}$$

with Arrhenius type functions

$$k_i(T) = k_{i0} \exp\left(\frac{-E_i}{T + T_0}\right) \quad \text{for } i = 1, 2.$$

The first control input u_1 is the normalized flow rate, whereas the second control input u_2 is the cooling power. The concentration and temperature of the inflow are denoted by c_{in} and T_{in}, respectively. The quantities α, β, γ, δ, k_{10}, k_{20}, E_1, E_2, ΔH_{AB}, ΔH_{BC}, ΔH_{AD} and T_0 are real constants. As suggested in [29,28] we use the output

$$\boldsymbol{y} = \boldsymbol{h}(\boldsymbol{x}) = \begin{pmatrix} T \\ \dfrac{c_{\text{in}} - c_A}{c_B} \end{pmatrix}, \tag{28}$$

where the first component is the measured temperature and the second component is the inverse selectivity.

If we set $u_1 = u_2 = 0$, system (27) defines a vector field $\boldsymbol{f}$. First, we want to compute iterated Lie derivatives $L_{\boldsymbol{f}}^k \boldsymbol{h}$. We computed these Lie derivatives with the computer algebra system Maple 10 (Maplesoft/Waterloo Maple Inc.). To compare the run time with algorithmic differentiation, we converted the results into C code using the Maple package `CodeGeneration`. Since output (28) is two-dimensional, each Lie derivative $L_{\boldsymbol{f}}^k \boldsymbol{h}$ yields two expressions. With the option `optimize` Maple extracts common subexpressions and generates a longer list of relatively short expressions. Table 4 shows the length of the generated C code. Without optimization, the expressions' sizes increase exponentially. However, the optimization for $L_{\boldsymbol{f}}^7 \boldsymbol{h}$ required several hours computation time.

Table 4
Length of the C code of iterated Lie derivatives $L_f^k \boldsymbol{h}$

Order k	`optimize = false`	`optimize = true`
1	290 bytes	283 bytes
2	1.4 KB	872 bytes
3	9.0 KB	2.5 KB
4	75.5 KB	7.1 KB
5	756.4 KB	21.6 KB
6	8.5 MB	64.3 KB
7	–	187.3 KB

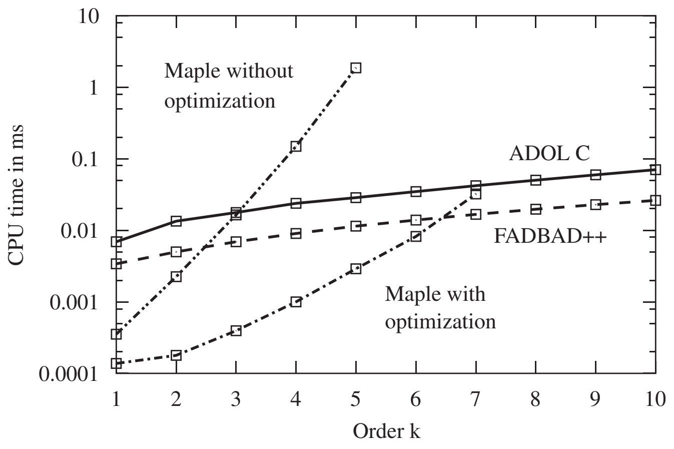

Fig. 2. CPU time for the evaluation of iterated Lie derivatives $L_f^k \boldsymbol{h}$.

Next, we compare the run time of the generated C code with a direct computation of the Lie derivatives using the algorithmic differentiation tools ADOL-C 1.10.1 [11] as well as FADBAD + + 1.4 [31] and the approach described in Section 4.1. The tests were carried out on a Fedora Core 4 Linux system with an AMD Althon 64 3500+ processor. We used the GNU C + + compiler GCC 4.0.2 with the code optimization option `-O3`. The results are shown in Fig. 2. The calculation of lower order Lie derivatives is faster with symbolic differentiation. An optimization of the code generation reduces the run time significantly. We were not able to compile the 8.5 MB source of $L_f^6 \boldsymbol{h}$ (even if we disable the compiler optimization). For higher order Lie derivatives, algorithmic differentiation might be the better choice. With FADBAD + + we have a speed-up factor of approximately 2.5 compared to ADOL-C. Lie derivatives of order $k \geqslant 8$ can only be computed with algorithmic differentiation.

Finally, we consider the calculation of mixed Lie derivatives $L_{f_1} \cdots L_{f_k} \boldsymbol{h}$. System (27) is a multi-input multi-output control system of form (8) with $n = 4$ and $m = p = 2$. We define the vector fields $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_k$ by $\boldsymbol{f}_i(\boldsymbol{x}) = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{g}_1(\boldsymbol{x}) u_{1i} + \boldsymbol{g}_2(\boldsymbol{x}) u_{2i}$ with real numbers $u_{1i}, u_{2i} \in \mathbb{R}$ for $i = 1, \ldots, k$. In other words, the different vector fields are generated from (27) via a selection of different values for the input. The test was carried out under the same conditions as in case of the iterated Lie derivatives. The measured evaluation time is shown in Fig. 3. Maple with optimized C code output yields the best results. If Maple is used without optimization, the run time of the mixed Lie derivatives is of the same order of magnitude as ADOL-C for $k = 4, 5$. To use FADBAD + + for $k = 4, 5$ we have to increase the length of the arrays in which the Taylor coefficients are stored. This can be achieved either by changing the macro `MaxLength` in the header file `tadiff.h` or by the command line option `-D MaxLength= ...` of the C + + compiler GCC. Due to the already very high order of the required derivatives ($d = 49$ for $k = 4$ and $d = 129$ for $k = 5$), the computation of mixed Lie derivatives of order $k > 5$ using univariate Taylor arithmetic is not feasible.

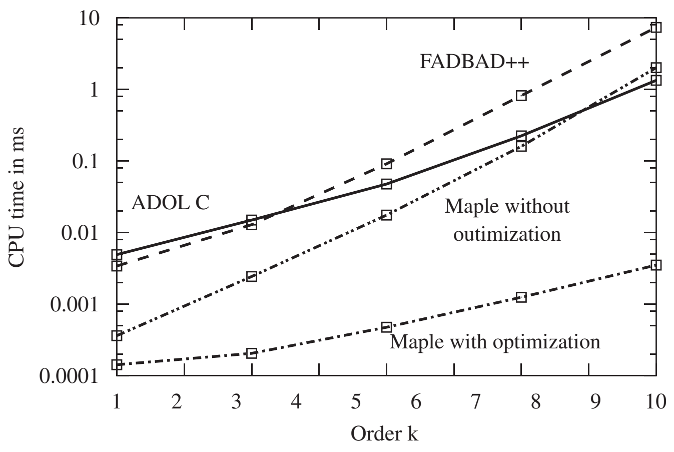

Fig. 3. CPU time for the evaluation of mixed Lie derivatives $L_{f_1} \cdots L_{f_k} \boldsymbol{h}$.

6. Conclusions

We considered the computation of multiple Lie derivatives. For a symbolic computation, the used computed algebra system must essentially be able to compute first order derivatives. For multiple Lie derivatives one simply repeats the computation procedure. Iterated Lie derivatives can easily be implemented by recursive programming. The underlying procedure is the same both for iterated and for mixed Lie derivatives.

Using algorithmic differentiation tools that provide univariate Taylor arithmetic, the computation of iterated Lie derivatives is straightforward. Although a symbolic computation might result in faster code for lower order derivatives, the usage of algorithmic differentiation is advantageous for higher order Lie derivatives and very complicated systems.

In this paper, we extended existing results on algorithmic differentiation to the calculation of mixed Lie derivatives. This computation required a completely different framework as iterated Lie derivatives. The speed of algorithmic differentiation to compute Taylor coefficients is basically compensated by the enormous increase w.r.t. the required derivative order. The developed approach is only feasible up to order 4 and occasionally up to order 5. However, our method can basically be applied to arbitrarily complicated systems. In particular, our approach is applicable if the system is described by a computation procedure containing branches and loops.

References

[1] C. Bendtsen, O. Stauning, TADIFF, a flexible C + + package for automatic differentiation, Technical Report IMM-REP-1997–07, TU of Denmark, Deparment of Mathematical Modelling, Lungby, 1997.

[2] M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, SIAM, Philadelphia, PA, 1996.

[3] J. Birk, M. Zeitz, Computer-aided design of nonlinear observers, in: Proceedings of the IFAC-Symposium Nonlinear Control System Design, Capri, Italy, 1989.

[4] H. Chen, A. Kremling, F. Allgöwer, Nonlinear predictive control of a CSTR benchmark problem, in: Proceedings of the Third European Control Conference ECC'95, Roma, Italy, 1995, pp. 3247–3252.

[5] K.-T. Chen, Integration of paths, geometric invariants and a generalized Baker–Hausdorff formula, Ann. Math. 65 (1) (1957) 163–178.

[6] G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (Eds.), Automatic Differentiation: From Simulation to Optimization, Springer, New York, 2002.

[7] F. Faa de Bruno, Note sur une nouvelle formule de calcule differentiel, Quart. J. Math. 1 (1856) 359–360.

[8] M. Fliess, M. Lamnabhi, F. Lannabhi-Lagarrigue, An algebraic approach to nonlinear functional expansion, IEEE Trans. Circuits and Systems 30 (8) (1983) 554–570.

[9] J. C. Gómez, Using symbolic computation for the computer aided design of nonlinear (adaptive) control systems, in: 14th IMACS World Congress on Computational and Applied Mathematics, Atlanta, USA, July 11–15, 1994.

[10] A. Griewank, Evaluating Derivatives—Principles and Techniques of Algorithmic Differentiation, Frontiers in Applied Mathematics, vol. 19, SIAM, Philadelphia, PA, 2000.

[11] A. Griewank, D. Juedes, J. Utke, ADOL-C: a package for automatic differentiation of algorithms written in C/C + +, ACM Trans. Math. Software 22 (1996) 131–167.

[12] A. Griewank, J. Utke, A. Walther, Evaluating higher derivative tensors by forward propagation of univariate Taylor series, Math. Comput. 69 (2000) 1117–1130.
[13] W. Gröbner, Die Lie-Reihen und ihre Anwendung, Deutscher Verlag der Wissenschaften, Berlin, 1967.
[14] M. Hardy, Combinatorics of partial derivatives, Electron. J. Combin. 13 (2006) 1–13.
[15] C.J. Isham, Modern Differential Geometry for Physicists, second ed., World Scientific, Singapore, 2001.
[16] A. Isidori, Nonlinear Control Systems: An Introduction, third ed., Springer, London, 1995.
[17] B. de Jager, The use of symbolic computation in nonlinear control: is it viable?, IEEE Trans. Automat. Control 40 (1) (1995) 84–89.
[18] B. de Jager, Symbolic computation in nonlinear control system modelling and analysis, in: 10th IEEE International Symposium on Computer Aided Control System Design jointly with the 1999 Conference on Control Applications (1999 IEEE CCA/CACSD), Kohala Cost, Island of Hawai, USA, August 22–27, 1999.
[19] K.-U. Klatt, S. Engell, Rührkesselreaktor mit Parallel- und Folgereaktion, in: S. Engell (Ed.), Nichtlineare Regelungen: Methoden, Werkzeuge, Anwendungen, VDI-Berichte, vol. 1026, VDI-Verlag, Düsseldorf, 1993, pp. 101–108.
[20] A. Kugi, K. Schlacher, R. Novaki, Symbolic Computation for the Analysis and Synthesis of Nonlinear Control Systems, Software Studies, vol. 2, WIT-Press, Southampton, 1999, pp. 255–264.
[21] H.G. Kwatny, G.L. Blankenship, Nonlinear Control and Analytical Mechanics: A Computational Approach, Birkhäuser, Boston, 2000.
[22] R.L. Mishkov, Generalization of the formula of Faa di Bruno for a composite function with a vector argument, Internat. J. Math. Math. Sci. 24 (7) (2000) 481–491.
[23] H. Nijmeijer, A.J. van der Schaft, Nonlinear Dynamical Control Systems, Springer, Berlin, 1990.
[24] V. Polyakov, R. Ghanadan, G. L. Blankenship, Symbolic numerical computational tools for nonlinear and adaptive control, in: Proceedings of the IEEE/IFAC Joint Symposium on Computer-Aided Control System Design, Tucson, Arizona, 1994, pp. 117–122.
[25] J. D. Pryce, J. K. Reid, AD01, a Fortran 90 code for automatic differentiation, Technical Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Computing and Information Systems Department, 1998.
[26] K. Röbenack, Computation of Lie derivatives of tensor fields required for nonlinear controller and observer design employing automatic differentiation, Proc. Appl. Math. Mech. 5 (1) (2005) 181–184.
[27] K. Röbenack, K.J. Reinschke, The computation of Lie derivatives and Lie brackets based on automatic differentiation, Z. Angew. Math. Mech. 84 (2) (2004) 114–123.
[28] R. Rothfuß, Anwendung der flachheitsbasierten Analyse und Regelung nichtlinearer Mehrgrößensysteme, VDI-Forschrittsberichte, vol. 664, Reihe 8: Meß-, Steuerungs- und Regelungstechnik, VDI-Verlag, Düsseldorf, 1997.
[29] R. Rothfuß, R. Rudolph, M. Zeitz, Flatness based control of a nonlinear chemical reactor model, Automatica 32 (10) (1996) 1433–1439.
[30] W.J. Rugh, Nonlinear System Theory: The Volterra/Wiener Approach, The Johns Hopkins University Press, Baltimore, MD, 1981.
[31] O. Stauning, Flexible automatic differentiation using templates and operator overloading in C++, Talk Presented at the Automatic Differentiation Workshop at Shrivenham Campus, Cranfield University, June 2003.
[32] P. Wambacq, W. Sansen, Distortion Analysis of Analog Integrated Circuits, Kluwer Academic Publishers, Dordrecht, 1998.