LIEDRIVERS — A Toolbox for the Efficient Computation of Lie Derivatives Based on the Object-Oriented Algorithmic Differentiation Package ADOL-C

Klaus Röbenack Jan Winkler Siqian Wang

Technische Universität Dresden, Faculty of Electrical and Computer Engineering, Institute of Control Theory,
{klaus.roebenack,jan.winkler,siqian.wang}@tu-dresden.de

Abstract

Lie derivatives are widely used in mathematics and physics. They are usually computed symbolically using computer algebra software. This symbolic computation might fail for very complicated expressions. Moreover, symbolic differentiation becomes more difficult if the function to be differentiated is not described explicitly as a function but by an algorithm. This is a situation occuring quite often in modeling languages. In this contribution we present an approach for calculating Lie derivatives based on algorithmic differentiation using the software package ADOL-C avoiding the drawbacks of symbolic differentiation.

Keywords Lie derivatives, algorithmic differentiation

1. Introduction

Lie derivatives play an important role in mathematics as well as physics [18, 26, 43]. Many methods in control engineering and system theory require Lie derivatives as well [20, 25]. To get an intuitive understanding, Lie derivatives can be considered as total derivatives of certain fields along the solution of a differential equation [23].

Assume we have described a physical system as lumped parameter model resulting in a set of nonlinear ordinary differential equations

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{F}(\boldsymbol{x}, \boldsymbol{u}), \quad \boldsymbol{y} = \boldsymbol{H}(\boldsymbol{x}, \boldsymbol{u}) \tag{1}$$

with the state $\boldsymbol{x}$, the input $\boldsymbol{u}$, the output $\boldsymbol{y}$ and appropriate maps $\boldsymbol{F}$ and $\boldsymbol{H}$. If system (1) is formulated in terms of a modeling language, this description is usually only employed for simulation and optimization. However, operator overloading techniques known from object-oriented programming allow the simultaneous usage of the description of (1) for control-related tasks such as controller and observer design [32]. More precisely, several algorithms in controller design rely on Lie derivatives [20, 25]. Similarly, several approaches for observer design can be formulated in terms of Lie derivatives [11, 14, 38]. Using operator overloading, these derivatives can efficiently be computed using an alternative differentiation technique called *algorithmic differentiation* [17]. We present a toolbox to carry out these calculations. Our implementation is based on the widely used algorithmic differentiation package ADOL-C (Automatic Differentiation by OverLoading in C++), cf. [16, 46].

The paper is structured as follows. In Section 2 we remind the reader of some definitions concerning Lie derivatives. Section 3 addresses the calculation of derivatives. The toolbox is presented in Section 4. The application of Lie derivatives in nonlinear control is the topic of Section 5. We use the toolbox for an example system in Section 6 and draw some conclusions in Section 7.

2. Basic Definitions from Differential Geometry

2.1 Tensors, Fields and Flows

First, we would like to recall some facts from differential geometry [3, 19, 23]. We restrict ourselves to the n-dimensional real vector space $\mathbb{V} = \mathbb{R}^n$. The elements of a vector space are *vectors*, and we represent the elements of $\mathbb{V}$ as column vectors. We denote the set of linear maps from the vector space $\mathbb{V}$ to another vector space $\mathbb{W}$ by $L(\mathbb{V}, \mathbb{W})$. Formally, the dual space of $\mathbb{V}$, denoted by $\mathbb{V}^*$, is the set of all linear functionals over $\mathbb{V}$, i.e., $\mathbb{V}^* = L(\mathbb{V}, \mathbb{R})$. The dual space $\mathbb{V}^*$ is also a vector space, whose elements are called *covectors*. Due to Riesz's representation theorem we can represent the covectors by row vectors.

A multi-linear map (i.e., a map that is linear separately in each variable)

$$(\mathbb{V}^*)^p \times \mathbb{V}^q \to \mathbb{R} \tag{2}$$

is called a *p-covariant q-contravariant tensor*, or (p, q)-*tensor*. The set $\mathbb{V}^p_q$ of (p, q)-tensors over $\mathbb{V}$ is a vector space itself. In case of $p = q = 0$ we set $\mathbb{V}^0_0 \equiv \mathbb{R}$, i.e., a $(0, 0)$-tensor is a scalar. In case of $p = 0$ and $q = 1$ the map (2) simplifies to a linear map $\mathbb{V} \to \mathbb{R}$, which belongs by definition to the dual space of $\mathbb{V}$, i.e., $\mathbb{V}^0_1 = \mathbb{V}^*$. Hence, $(0, 1)$-tensors are covectors. In case of $p = 1$ and $q = 0$

4th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools. September, 2011, ETH Zürich, Switzerland.
Copyright is held by the author/owner(s). The proceedings are published by Linköping University Electronic Press. Proceedings available at:
http://www.ep.liu.se/ecp/056/
EOOLT 2011 website:
http://www.eoolt.org/2011/

the map (2) simplifies to another linear map $\mathbb{V}^* \to \mathbb{R}$, which belongs to the dual space $(\mathbb{V}^*)^*$ of the dual space $\mathbb{V}^*$. Since the vector space $\mathbb{V}$ is finite dimensional, the so-called bidual space is isomorphic to the original vector space $\mathbb{V}$, i.e., $\mathbb{V}_0^1 = (\mathbb{V}^*)^* \cong \mathbb{V}$. Therefore, the $(1,0)$-tensors are vectors.

The following consideration can be carried out for smooth manifolds. Since the paper deals mainly with computational issues, which will always be carried out in local coordinates, we can restrict ourselves to an open subset $\mathcal{M} \subseteq \mathbb{V}$ of $\mathbb{R}^n$. The *tangent space* at $\boldsymbol{x} \in \mathcal{M}$, denoted by $T_{\boldsymbol{x}}\mathcal{M}$, is isomorphic to $\mathbb{V}$, i.e., $T_{\boldsymbol{x}}\mathcal{M} \cong \mathbb{V}$. (See [23, Chap. 3] for the construction of the geometric tangent space of $\mathbb{R}^n$.) The dual space of $T_{\boldsymbol{x}}\mathcal{M}$ is called *cotangent space* and denoted by $T_{\boldsymbol{x}}^*\mathcal{M} \cong \mathbb{V}^*$. In the paper, we will deal with the following special cases: If $\mathcal{S}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}_0^0 \cong \mathbb{R}$, $\mathcal{S}$ is a *scalar field*. If $\mathcal{S}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}_0^1 = T_{\boldsymbol{x}}\mathcal{M} \cong \mathbb{V}$, $\mathcal{S}$ is a *vector field*. If $\mathcal{S}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}_1^0 = T_{\boldsymbol{x}}^*\mathcal{M} \cong \mathbb{V}^*$, $\mathcal{S}$ is a *covector field*. Covector fields are also called *differential forms* of degree 1 (*1-forms*). An *exact* differential form is the gradient of a scalar field.

A vector field $\boldsymbol{f}$, which maps each point $\boldsymbol{x} \in \mathcal{M}$ to the corresponding tangent space $\boldsymbol{f}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}$ (see Figure 1), can be associated with the differential equation

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t)) \ . \tag{3}$$

The *flow* $\boldsymbol{\varphi}_t$ of the vector field $\boldsymbol{f}$ is the general solution of (3), i.e., the curve $\boldsymbol{\varphi}_t(\boldsymbol{x}_0)$ is the solution of (3) with the initial condition $\boldsymbol{x}(0) = \boldsymbol{x}_0 \in \mathcal{M}$. Moreover, the flow is a one-parametric family of local diffeomorphisms on $\mathcal{M}$ having the structure of a transformation group with $\boldsymbol{\varphi}_0(\boldsymbol{x}_0) = \boldsymbol{x}_0$ and $\boldsymbol{\varphi}_s(\boldsymbol{\varphi}_t(\boldsymbol{x}_0)) = \boldsymbol{\varphi}_{s+t}(\boldsymbol{x}_0)$ for all sufficiently small $|t|, |s|$. The flow of (3) has the following series expansion:

$$\boldsymbol{\varphi}_t(\boldsymbol{x}) = \boldsymbol{x} + \boldsymbol{f}(\boldsymbol{x})t + \tfrac{1}{2}\boldsymbol{f}'(x)\boldsymbol{f}(\boldsymbol{x})t^2 + \mathcal{O}(t^3) \ , \tag{4}$$

which reflects the fact that $\boldsymbol{f}(\boldsymbol{x})$ is tangent to $\boldsymbol{\varphi}_t(\boldsymbol{x})$ in the point $\boldsymbol{x}$.

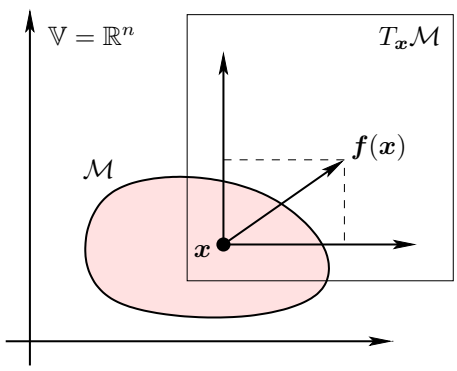

Figure 1. Tangent space and vector field $\boldsymbol{f}$.

2.2 Lie Derivatives

In this section we will define Lie derivatives of scalar, vector and covector fields.

2.2.1 Lie Derivative of a Scalar Field

Consider the differential equation (3) with the vector field $\boldsymbol{f} : \mathcal{M} \to \mathbb{R}^n$ and the associated flow $\boldsymbol{\varphi}_t$. The *Lie derivative of the scalar field* $h : \mathcal{M} \to \mathbb{R}$ is defined as

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

Using the series expansion (4) of the flow, with

$$\begin{aligned} L_{\boldsymbol{f}}h(\boldsymbol{x}) &= \frac{\mathrm{d}}{\mathrm{d}\,t}h(\boldsymbol{\varphi}_t(\boldsymbol{x}))\bigg|_{t=0} \\ &= h'(\boldsymbol{\varphi}_t(\boldsymbol{x}))\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t(\boldsymbol{x})\bigg|_{t=0} \\ &= h'(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x}) \end{aligned} \tag{6}$$

we obtain the explicit computation rule (6) as the scalar product between the gradient $\mathrm{d}\,h = h'$ and the vector field $\boldsymbol{f}$. In other words: $L_{\boldsymbol{f}}h$ is the directional derivative of h in the direction of $\boldsymbol{f}$ (see Figure 2). The Lie derivative $L_{\boldsymbol{f}}h$ is again a scalar field. Therefore, we can repeat this process and define multiple Lie derivatives along the same vector field $\boldsymbol{f}$ by

$$L_{\boldsymbol{f}}^{k+1}h(\boldsymbol{x}) = L_{\boldsymbol{f}}L_{\boldsymbol{f}}^{k}h(\boldsymbol{x}) = \frac{\partial L_{\boldsymbol{f}}^{k}h(\boldsymbol{x})}{\partial \boldsymbol{x}}\,\boldsymbol{f}(\boldsymbol{x}) \tag{7}$$

with $L_{\boldsymbol{f}}^{0}h(\boldsymbol{x}) = h(\boldsymbol{x})$.

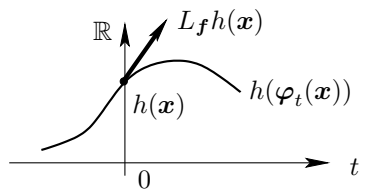

Figure 2. Lie derivative $L_{\boldsymbol{f}}h$ of a scalar field h

2.2.2 Lie Derivative of a Vector Field

We need some preliminary remarks before we introduce the Lie derivative of a vector field. Consider a smooth map $\boldsymbol{\psi} : \mathcal{M} \to \mathcal{M}$, which maps a point $\boldsymbol{x} \in \mathcal{M}$ into the point $\boldsymbol{\psi}(\boldsymbol{x}) \in \mathcal{M}$. The *differential* of *push-forward* is a linear map between the associated tangent spaces $\boldsymbol{\psi}_* : T_{\boldsymbol{x}}\mathcal{M} \to T_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M}$, which can be represented by the Jacobian matrix $\boldsymbol{\psi}'$ of $\boldsymbol{\psi}$. If $\boldsymbol{\psi}$ is a diffeomorphism, the inverse map $\boldsymbol{\psi}^{-1}$ of $\boldsymbol{\psi}$ exists and is continuously differentiable. Then, the linearization of the inverse map $\boldsymbol{\psi}^{-1}$ defines another map $\boldsymbol{\psi}^* := \boldsymbol{\psi}_*^{-1} : T_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M} \to T_{\boldsymbol{x}}\mathcal{M}$ called *pull-back*. Clearly, the Jacobian of the inverse map is the inverse Jacobian matrix of the original map. In case of $\boldsymbol{\psi} = \boldsymbol{\varphi}_t$, we obtain from (4) a series expansion of push-forward

$$\boldsymbol{\varphi}_{t*}(\boldsymbol{x}) = I + \boldsymbol{f}'(\boldsymbol{x})t + \mathcal{O}(t^2) \ , \tag{8}$$

where I denotes the identity matrix. Due to the group property, the inverse map of the flow is given as the flow in reverse time, i.e., $\boldsymbol{\varphi}_t^{-1} = \boldsymbol{\varphi}_{-t}$. This implies $\boldsymbol{\varphi}_t^* = \boldsymbol{\varphi}_{t*}^{-1} = \boldsymbol{\varphi}_{-t*}$ having the series expansion

$$\boldsymbol{\varphi}_t^*(\boldsymbol{z}) = \boldsymbol{\varphi}_{-t*}(\boldsymbol{z}) = I - \boldsymbol{f}'(\boldsymbol{z})t + \mathcal{O}(t^2) \tag{9}$$

with $\boldsymbol{z} = \boldsymbol{\varphi}_t(\boldsymbol{x})$.

We are now able to introduce the *Lie derivative of a vector field* $\boldsymbol{g}$ along $\boldsymbol{f}$ as

$$L_{\boldsymbol{f}}\boldsymbol{g}(\boldsymbol{x}) = \mathrm{ad}_{\boldsymbol{f}}\,\boldsymbol{g}(\boldsymbol{x}) = \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t^*\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \quad . \tag{10}$$

This Lie derivative is called *Lie bracket* and often denoted by $\mathrm{ad}_{\boldsymbol{f}}\,\boldsymbol{g} = [\boldsymbol{f}, \boldsymbol{g}]$. Using the series expansions (4) and (9) we obtain

$$\begin{aligned}
\mathrm{ad}_{\boldsymbol{f}}\,\boldsymbol{g}(\boldsymbol{x}) &= \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t^*\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \\
&= \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_{-t*}\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \\
&= \lim_{t\to 0}\frac{\boldsymbol{\varphi}_{-t*}\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x})) - \boldsymbol{g}(\boldsymbol{x})}{t} \qquad (11)\\
&= \boldsymbol{g}'(\boldsymbol{x})\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}'(\boldsymbol{x})\boldsymbol{g}(\boldsymbol{x}) \ . \qquad (12)
\end{aligned}$$

The reason for the seemingly complicated construction (10) becomes apparent in Eq. (11). In general, the derivative of a function is defined as the limit of the difference quotient. Formally, the two terms $\boldsymbol{g}(\boldsymbol{x}) \in T_{\boldsymbol{x}}\mathcal{M}$ and $\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x})) \in T_{\boldsymbol{\varphi}_t(\boldsymbol{x})}\mathcal{M}$ required for the difference quotient belong to different vector spaces. Using pull-back we map $\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x}))$ back from $T_{\boldsymbol{\varphi}_t(\boldsymbol{x})}\mathcal{M}$ to $T_{\boldsymbol{x}}\mathcal{M}$ and obtain $\boldsymbol{\varphi}_t^*\boldsymbol{g}(\boldsymbol{\varphi}_t(\boldsymbol{x})) \in T_{\boldsymbol{x}}\mathcal{M}$. Then, we can compute the difference in (11) since both objects lie in the same tangent space.

The Lie derivative $\mathrm{ad}_{\boldsymbol{f}}\,\boldsymbol{g}$ of a vector field is a vector field again. Higher order Lie derivatives of a vector field $\boldsymbol{g}$ are recursively defined by

$$\mathrm{ad}_{\boldsymbol{f}}^{k+1}\,\boldsymbol{g}(\boldsymbol{x}) = [\boldsymbol{f}, \mathrm{ad}_{\boldsymbol{f}}^{k}\,\boldsymbol{g}](\boldsymbol{x}) \tag{13}$$

with $\mathrm{ad}_{\boldsymbol{f}}^{0}\,\boldsymbol{g} = \boldsymbol{g}(\boldsymbol{x})$.

2.2.3 Lie Derivative of a Covector Field

Consider a smooth map $\boldsymbol{\psi} : \mathcal{M} \to \mathcal{M}$ with its push-forward $\boldsymbol{\psi}_* : T_{\boldsymbol{x}}\mathcal{M} \to T_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M}$ and a covector field $\boldsymbol{\omega}$, i.e., $\boldsymbol{\omega}(\boldsymbol{x}) \in T^*_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M}$ for $\boldsymbol{x} \in \mathcal{M}$. The *dual* or *adjoint* map $\boldsymbol{\psi}^* : T_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M} \to T_{\boldsymbol{x}}\mathcal{M}$ defined by $\langle \boldsymbol{\psi}^*\boldsymbol{\omega}, \boldsymbol{z}\rangle = \langle \boldsymbol{\omega}, \boldsymbol{\psi}_*\boldsymbol{z}\rangle$ for all $\boldsymbol{z} \in T_{\boldsymbol{x}}\mathcal{M}$ and $\boldsymbol{\omega}(\boldsymbol{x}) \in T^*_{\boldsymbol{\psi}(\boldsymbol{x})}\mathcal{M}$ is called *pull-back* of the covector field $\boldsymbol{\omega}$. For $\boldsymbol{\psi} = \boldsymbol{\varphi}_t$ we obtain from (4) the following series expansion of pull-back of the covector field $\boldsymbol{\omega}$ for the flow:

$$\begin{aligned}
\boldsymbol{\varphi}_t^*\boldsymbol{\omega}(\boldsymbol{x}) &= \boldsymbol{\omega}(\boldsymbol{x})\boldsymbol{\varphi}_{t*}(\boldsymbol{x}) \\
&= \boldsymbol{\omega}(\boldsymbol{x})(I + \boldsymbol{f}'(\boldsymbol{x})t + \mathcal{O}(t^2)) \ .
\end{aligned} \tag{14}$$

The *Lie derivative of the covector field* $\boldsymbol{\omega}$ is defined by

$$L_{\boldsymbol{f}}\boldsymbol{\omega}(\boldsymbol{x}) = \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t^*\boldsymbol{\omega}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \quad . \tag{15}$$

The definitions (10) and (15) of the Lie derivatives of a vector and a covector field look exactly the same. However, pull-back acts on different mathematical objects (on a vector field in (10) and on a covector field in (15)) and therefore on different spaces.

Using the series expansions (4) and (14) we obtain by

$$\begin{aligned}
L_{\boldsymbol{f}}\boldsymbol{\omega}(\boldsymbol{x}) &= \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\varphi}_t^*\boldsymbol{\omega}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \\
&= \left.\frac{\mathrm{d}}{\mathrm{d}\,t}\boldsymbol{\omega}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\boldsymbol{\varphi}_{t*}(\boldsymbol{\varphi}_t(\boldsymbol{x}))\right|_{t=0} \\
&= \boldsymbol{\omega}(\boldsymbol{x})\boldsymbol{f}'(\boldsymbol{x}) + \boldsymbol{f}^T(\boldsymbol{x})\left(\frac{\partial\boldsymbol{\omega}^T(\boldsymbol{x})}{\partial\boldsymbol{x}}\right)^T \qquad (16)
\end{aligned}$$

an explicit computation rule for the Lie derivative (15).

As mentioned in Section 2.1, the gradient $\mathrm{d}\,h$ of a scalar field h is a covector field. Therefore, the Lie derivative $L_{\boldsymbol{f}}\,\mathrm{d}\,h$ can be calculated by (16) with $\boldsymbol{\omega} = \mathrm{d}\,h$. Alternatively, the Lie derivative $L_{\boldsymbol{f}}\,\mathrm{d}\,h$ can also be obtained from the gradient of the Lie derivative $L_{\boldsymbol{f}}h$:

$$L_{\boldsymbol{f}}\,\mathrm{d}\,h(\boldsymbol{x}) = \mathrm{d}\,L_{\boldsymbol{f}}h(x) \ . \tag{17}$$

Eq. (17) can be stated as follows: The exterior derivative commutes with the Lie derivative.

3. Computation of Derivatives

3.1 General Differentiation Techniques

The *(Fréchet) derivative* of a smooth map $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ in the point $\boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n$ is a linear map $A \in L(\mathbb{R}^n, \mathbb{R}^m)$, for which

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{F}(\boldsymbol{x}_0) + A(\boldsymbol{x} - \boldsymbol{x}_0) + o(\|\boldsymbol{x} - \boldsymbol{x}_0\|)$$

holds for all $\boldsymbol{x}$ in a neighbourhoud of $\boldsymbol{x}_0$. The linear map A can be identified with an $m \times n$-matrix $A = \boldsymbol{F}'(\boldsymbol{x}) \in \mathbb{R}^{m\times n}$, the so-called *Jacobian matrix*.

In this paper we assume that the map $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ is given as a computer program (such as a function, procedure, method etc.), i.e., $\boldsymbol{F}$ is described as a finite sequence of elementary functions and operations. The derivative of $\boldsymbol{F}$ can be computed systematically using elementary differentiation rules in combination with the chain rule. This differentiation process can be carried out by computer algebra systems such as MATHEMATICA [48], MAPLE [10] or MAXIMA [2]. The result of this *symbolic computation* is a symbolic expression.

The computer algebra system might not be able to carry out a symbolic differentiation if the function under consideration is very complicated. Moreover, symbolic computation is usually not (directly) possible if the function to be differentiated is not given explicitly but by algorithms containing branches, loops and subroutines. Moreover, the sizes of symbolic expressions usually increase significantly w.r.t. the order of the derivative.

Another widely used method to compute derivative values is *numeric differentiation* by divided differences. Depending on the step size, the resulting derivative will be affected by truncation or cancellation errors. Even for an optimal step size, the derivative value will have a significantly reduced precision compared to the function value.

The disadvantages of symbolic and numeric differentiation can be circumvented by an alternative technique

called *algorithmic* or *automatic differentiation* [17]. Similar to symbolic differentiation, elementary differentiation rules are applied systematically. In contrast to symbolic differentiation, all intermediate results are not symbolic expressions but numerical values, i.e., the intermediate values are immediately evaluated and stored as floating point numbers.

3.2 Symbolic Computation of Lie Derivatives

Symbolically, Lie derivatives of scalar, vector and covector fields are computed based on Eqs. (6), (12) and (16). Hence, one needs to compute gradients or Jacobian matrices. These operations are well-supported by the usual computer algebra systems. For example in MAPLE, the Lie derivative (6) of the scalar field h along the vector field $\boldsymbol{f}$ can be implemented as follows:

```
with(linalg);
LieScalar:=proc(f,h,x)
 multiply(jacobian(h,x),f)
end_proc;
```

Higher order Lie derivatives can be calculated easily using finite recursions such as (7) or (13). The symbolic computation of Lie derivatives was implemented in [7, 22, 27, 36] for MATHEMATICA and in [12, 21, 24, 37] for MAPLE. However, the symbolic calculation of Lie derivatives is not restricted to commercial systems. The open source computer algebra system MAXIMA contains toolboxes for tensor calculus, which were developed for computations in general relativity and support the calculation of arbitrary types of Lie derivatives [42]. Alternatively, the Lie derivatives (6), (12) and (16) can also be implemented in MAXIMA using the build-in linear algebra package. A possible implementation of the Lie derivative (6) reads as follows:

```
load("linearalgebra");
LieScalar(f,h,x):=jacobian([h],x).f;
```

3.3 Algorithmic Differentiation

3.3.1 Forward Mode

Assume we represent a vector-valued function $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$, $\mathcal{M} \subseteq \mathbb{R}^n$ with

$$\boldsymbol{z} = \boldsymbol{F}(\boldsymbol{x}) \tag{18}$$

by appropriate C++ code. Input values $\boldsymbol{x}_0 \in \mathcal{M}$, output values (i.e., function values) $\boldsymbol{z}_0 \in \mathbb{R}^m$ as well as intermediate values are stored as floating point numbers, usually with the build-in type `double`. Considering $\boldsymbol{x}$ and $\boldsymbol{z}$ as curves, we can compute the time derivative of (18) using the chain rule:

$$\dot{\boldsymbol{z}} = \boldsymbol{F}'(\boldsymbol{x})\,\dot{\boldsymbol{x}} \ . \tag{19}$$

For a given direction $\dot{\boldsymbol{x}}$ of $\mathbb{R}^n$ we obtain the directional derivative $\dot{\boldsymbol{z}}$ of $\boldsymbol{F}$. The tangent value $\dot{\boldsymbol{z}}$ can be obtained calculating the tangent values of all intermediate values occurring during the function evaluation. We repleace the floating point type `double` by a new class, e.g. `ddouble`, containing the function value and additionally the derivative value:

```
class ddouble
{
 public:
  double val; // function value
  double der; // derivative value
};
```

All differentiable functions (e.g. $\sin$, $\cos$, $\exp$, $\log$) that act on the type `double` must be replaced by appropriate methods for the class `ddouble` such that in addition to the function value one also computes the derivative value using elementary differentiation rules in connection with the chain rule:

```
ddouble sin (ddouble x)
{
 ddouble z;
 z.val = sin(x.val);
 z.der = x.der*cos(x.val);
}
```

In a similar way we have to provide the usual binary operation (e.g. $+$, $-$, $*$, $/$) for the new class `ddouble`. In case of multiplication we also have to take the product rule into account:

```
ddouble operator * (ddouble x, ddouble y)
{
 ddouble z;
 z.val = x.val*y.val;
 z.der = x.val*y.der+y.val*x.der;
}
```

This approach to algorithmic differentiation is called *forward mode* because the program flow of function evaluation and derivative computation have the same orientation.

3.3.2 Reverse Mode

In the reverse mode, the elementary statements are differentiated in reverse order, i.e., from the end to the beginning of the program. Therefore, the implementation is much more complicated. The reverse mode can be interpreted as a generalization of the backpropagation algorithm known from neuronal networks [47].

Mathematically, for a given covector (i.e., row vector) $\bar{\boldsymbol{z}} \in (\mathbb{R}^m)^*$, the reverse mode yields a covector $\bar{\boldsymbol{x}} \in (\mathbb{R}^n)^*$ with

$$\bar{\boldsymbol{x}} = \bar{\boldsymbol{z}}\,\boldsymbol{F}'(\boldsymbol{x}) \ , \tag{20}$$

which can be seen as a weighted derivative. For $m < n$, especially in case of a functional, i.e., $m = 1$, the reverse mode is more efficient than the forward mode. Therefore, the reverse mode is widely used in optimization [15].

3.3.3 Taylor Arithmetic

Assume the function $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ is sufficiently smooth to map a truncated Taylor series

$$\boldsymbol{x}(t) = \boldsymbol{x}_0 + \boldsymbol{x}_1 t + \boldsymbol{x}_2 t^2 + \cdots + \boldsymbol{x}_d t^d + \mathcal{O}(t^{d+1}) \tag{21}$$

with $\boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n$, $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_d \in \mathbb{R}^n$ into a curve

$$\begin{aligned} \boldsymbol{z}(t) &= \boldsymbol{F}(\boldsymbol{x}(t)) \\ &= \boldsymbol{z}_0 + \boldsymbol{z}_1 t + \boldsymbol{z}_2 t^2 + \cdots + \boldsymbol{z}_d t^d + \mathcal{O}(t^{d+1}) \end{aligned} \tag{22}$$

with $\boldsymbol{z}_0, \ldots, \boldsymbol{z}_d \in \mathbb{R}^m$. Clearly, each Taylor coefficient $\boldsymbol{z}_k$ depends only on the Taylor coefficients $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k$. The Taylor coefficient $\boldsymbol{z}_k$ can be calculated using the forward mode of automatic differentiation.

In reverse mode of automatic differentiation, one can compute the coefficient matrices $A_0, \ldots, A_d \in \mathbb{R}^{m \times n}$ of the Jacobian path

$$\mathbf{F}'(\mathbf{x}(t)) = A_0 + A_1 t + \ldots + A_d t^d + \mathcal{O}(t^{d+1}) \quad . \tag{23}$$

The matrices $A_0, \ldots, A_d$ are the partial derivatives of the Taylor coefficients of the curves $\mathbf{x}$ and $\mathbf{z}$. We have the following identity [8]:

$$\frac{\partial \mathbf{z}_j}{\partial \mathbf{x}_i} = \frac{\partial \mathbf{z}_{j-i}}{\partial \mathbf{x}_0} = \begin{cases} A_{j-i} & \text{for} \quad j \geq i \\ 0 & \text{otherwise.} \end{cases} \tag{24}$$

As illustrated, the curves $\boldsymbol{x}$ and $\boldsymbol{z}$ associated with a function $\boldsymbol{F}$ are represented by the coefficients of their corresponding truncated Taylor series expansions (21) and (22). Thus, it is necessary to introduce another datatype holding the values of these coefficients. As an example, one might define

```
#include <vector>
class tdouble:public std::vector<double>;
```

and implement the operations required for an appropriate handling of these truncated series expansions.

3.4 Implementations and Tools

As sketched in Section 3.3, algorithmic differentiation can easily be implmented if the programming language under consideration supports operator overloading. For C++, this approach is implemented in several algorithmic differentiation packages such as ADOL-C [16], FADBAD [5], FADBAD++ [41], TADIFF [6], AUTODIFF [39], CppAD [1]. Operator overloading is also supported in Fortran 90/95 and used by the tools AUTO_DERIV [40] and TaylUR [45]. Moreover, there are also tools that use operator overloading in Matlab for algorithmic differentiation, e.g. ADMAT [44] and TOMLAB/MAD [13].

It should be mentioned that algorithmic differentiation can also be implemented based on source code transformation, especially for programming languages such as C and Fortran 77 which do not support operator overloading. For more details on algorithmic differentiation tools and techniques we refer to [4, 17].

3.5 Differentiation Package ADOL-C

The software package ADOL-C provides routines for evaluation of derivatives of scalar or vector functions defined by computer programs written in the programming languages C or C++. The resulting derivative evaluation routines can be used in C, C++, Fortran, or any other language that can be linked against C [16]. ADOL-C uses operator overloading as described in Section 3.3, however with some additional features that will be discussed in the next section.

3.5.1 Tape generation

The fundamental working principle of ADOL-C is that in a first step the code representing e.g. a scalar valued function $f : \mathbb{R} \to \mathbb{R}$ is marked as a so called active section. Input and output variables are assigned as variables of type `adouble`, similar to the type `ddouble` discussed above:

```
adouble x,z;      // Active variables
double x0;        // point of expansion
double z0;        // function value

trace_on(tag);    // Begin of active section

x <<= x0;         // Assignment of independents
z = f(x);         // Evaluation
z >>= z0;         // Assignment of dependents

trace_off();      // End of active section
```

This code creates a data structure called *tape* holding the trace of the function evaluation. This tape is used in the following for the calculation of the derivatives using so called *drivers*, e.g. `gradient`, `jacobian`, `hessian`, cf. next section. It is important to understand that the tape has only to be created once a time for an aribitrary input value x_0 while repetitive calls of the drivers for different input values differing from the values the tape was generated from may follow. This holds as long as there are no user defined quadratures and all comparisons involving `adouble`s yield the same result. The drivers acting on these tapes provide a C as well as C++ interface, i.e., once a tape is generated it can also be used in environments that do not support C++. The tape is referenced by the integer `tag`.

3.5.2 Drivers

The package ADOL-C provides several commands called *drivers* for evaluation of different types of derivatives. Some of these drivers are sketched in the following:

Drivers for forward and reverse mode Given a tape of the sufficiently smooth function $\boldsymbol{F} : \mathcal{M} \to \mathbb{R}^m$ and the Taylor coefficients $X = (\boldsymbol{x}_0, \ldots, \boldsymbol{x}_d)$ of the series expansion (21) of the curve $\boldsymbol{x}$ of the independent variable one can compute the Taylor coefficients $Z = (\boldsymbol{z}_0, \ldots, \boldsymbol{z}_d)$ as given in (22) using the ADOL-C function `forward`:

```
int forward(tag,m,n,d,keep,X,Z)
short int tag;        // tape tag of F
int m;                // number of dependent variables m
int n;                // number of independent variables n
int d;                // highest derivative degree d
int keep;             // flag for reverse sweep
double X[n][d+1];     // Taylor coefficients (x_0, ..., x_d)
double Z[m][d+1];     // Taylor coeffcients (z_0, ..., z_d)
```

The keep flag must be set to `keep=1` if a further calculation using the reverse mode is intended. The 2-dimensional arrays X and Z are allocated as arrays of pointers.

The reverse mode can be employed for an efficient computation of the coefficient matrices $A_0, \ldots, A_d \in \mathbb{R}^{m \times n}$ of the Jacobian path (23). They are obtained by calling

the ADOL-C function `reverse` and stored in the 3-dimensional array `A`:

```
int reverse(tag,m,n,d,A)
short int tag;          // tape tag of F
int m;                  // number of dependent variables m
int n;                  // number of independent variables n
int d;                  // highest derivative degree d
double A[m][n][d+1];    // resulting matrices A_0, ..., A_d
```

Drivers for ordinary differential equations Given an initial value problem

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}(\boldsymbol{x}(t)) \quad \text{with} \quad \boldsymbol{x}(0) = \boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n \tag{25}$$

of the ordinary differntial equation (3). We want to compute a Taylor series expansion (21) of a solution of (25), i.e., of the flow of the vector field $\boldsymbol{f}$ passing through $\boldsymbol{x}_0$:

$$\boldsymbol{x}(t) = \boldsymbol{\varphi}_t(\boldsymbol{x}_0) \ . \tag{26}$$

On the other hand, we can treat $\boldsymbol{f}$ as a functions mapping the curve (21) into the curve (22) via

$$\boldsymbol{z}(t) = \boldsymbol{f}(\boldsymbol{x}(t)) \ . \tag{27}$$

Because of $\boldsymbol{f} : \mathcal{M} \subseteq \mathbb{R}^n \to \mathbb{R}^n$, both curves $\boldsymbol{x}$ and $\boldsymbol{z}$ belong to spaces of the same dimension. However, since $\boldsymbol{f}$ is a vector field, the curve $\boldsymbol{z}$ in the image of $\boldsymbol{f}$ must belong to the tangent space $T_{\boldsymbol{x}}\mathcal{M}$, i.e., we have the identify $\boldsymbol{z}(t) = \dot{\boldsymbol{x}}(t)$. Therefore, the Taylor coefficients of the series expansions (21) and (22) must satiesfy the identity

$$\boldsymbol{x}_{k+1} = \frac{1}{1+k}\boldsymbol{z}_k \ . \tag{28}$$

Knowing certain Taylor coefficients $\boldsymbol{x}_0, \ldots, \boldsymbol{x}_k$ of the solution of (25), one can use the forward mode to compute the Taylor coefficients $\boldsymbol{z}_0, \ldots, \boldsymbol{z}_k$ of the curve (27). Eq. (28) yields the next Taylor coefficient $\boldsymbol{x}_{k+1}$. To carry out this recursion, ADOL-C provides the function `forode`:

```
int forode(Tape_F, n, d, X)
short Tape_F;           // tape tag of vector field f
short n;                // dimension n
short d;                // highest degree d
double X[n][d+1];       // Taylor coefficients x_0, ..., x_d
```

The initial value $\boldsymbol{x}_0$ has to be stored in the zeroth position of the array `X`, i.e., in `X[0][0]`, ..., `X[n-1][0]`.

Carrying out a reverse sweep after `forode` we obtain the coefficient matrices $A_0, A_1, \ldots, A_{d-1} \in \mathbb{R}^{n\times n}$ similar to (23). Recall that these matrices can be interpreted as partial derivatives between Taylor coefficients as described in (24). Taking the dependencies resulting from (28) into account we can calculate the total derivatives

$$B_k = \frac{\mathrm{d}\boldsymbol{x}_{k+1}}{\mathrm{d}\boldsymbol{x}_0} \in \mathbb{R}^{n\times n} \tag{29}$$

for $k = 0, \ldots, d-1$ with the ADOL-C function `accode`:

```
int accode(n, d-1, A, B)
short n;                // dimension n
short d;                // highest degree d
double A[n][n][d];      // partial derivatives A_0, ..., A_{d-1}
double B[n][n][d];      // total derivatives B_0, ..., B_{d-1}
```

Note that the total derivatives (29) are the coefficient matrices of the series expansion of push-forward (8) of the flow:

$$\boldsymbol{\varphi}_{t*}(\boldsymbol{x}_0) = I + B_0 t + B_1 t^2 + B_2 t^3 + \cdots \ .$$

ADOL-C provides much more drivers, e.g. for optimization and nonlinear equations (calculation of gradients, Jacobians and Hessians) and higher derivative tensors which are not discussed here. For details refer to [16, 46].

3.5.3 Calculation of Lie derivatives using ADOL-C

In contrast to the symbolic computation the calculation of the Lie derivatives employing automatic differentiation is based directly on the definitions (6), (10) and (15). In order to use ADOL-C for the calculation of Lie-derivatives one has to combine certain calls of the drivers discussed above. This is illustrated by example of the Lie derivative $L_{\mathbf{f}}^k h(\mathbf{x})$ of a scalar field $h : \mathcal{M} \to \mathbb{R}$ along a vector field $\boldsymbol{f} : \mathcal{M} \to \mathbb{R}^n$. Using the ADOL-C driver `forode`, in a first step, one computes the Taylor coefficients $\boldsymbol{x}_1, \ldots \boldsymbol{x}_d \in \mathbb{R}^n$ of the series expansion (21) of (26) for a given initial value $\boldsymbol{x}_0 \in \mathcal{M} \subseteq \mathbb{R}^n$. In a second step we have to compute the Taylor coefficients $y_0, y_1, \ldots, y_d \in \mathbb{R}$ of the series expansion

$$y(t) = y_0 + y_1 t + \cdots y_d t^d + \mathcal{O}(t^{d+1}) \tag{30}$$

of the curve

$$y(t) = h(\boldsymbol{\varphi}_t(\boldsymbol{x}_0)) \ . \tag{31}$$

This can easily be done with the driver `forward`. Recall that the first order Lie derivative (5) is the total derivative of h along the flow of $\boldsymbol{f}$. Taking higher order Lie derivatives into account yields a so-called Lie series

$$\sum_{k=0}^{\infty} L_{\boldsymbol{f}}^k h(\boldsymbol{x}_0)\frac{t^k}{k!} = h(\boldsymbol{\varphi}_t(\boldsymbol{x}_0)) \ . \tag{32}$$

Matching the coefficients of (30) and (31) allows an explicit computation of the function values of the Lie derivatives by

$$L_{\boldsymbol{f}}^k h(\boldsymbol{x}_0) = k!\, y_k \quad \text{for} \quad k = 0, \ldots, d \ . \tag{33}$$

Assuming that the recorded tapes of $\boldsymbol{f}$ and h are referenced by the integers `Tape_f` and `Tape_h`, respectively, one has:

```
// Taylor coefficients X = (x_k)
forode(Tape_f,n,d,X);

// Taylor coefficients Y = (y_k)
forward(Tape_h,m,n,d,X,Y);

// Lie-Derivatives up to order d
double Lfh[d+1];
int fak = 1;
for (i=0; i <= d; i++) {
   Lfh[i] = y[i]*fak;
   fak *= (i+1);
}
```

In a similar way one can compute the gradients of Lie derivatives, the Lie derivatives of a covector field and Lie brackets. For example, consecutive calls of `reverse` and `accode` provide the coefficient matrices of the series expansion (8) required for the computation of Lie derivatives of covector fields, see Eqs. (14) and (16).

4. Toolbox of Lie-Derivatives

The drivers provided for calculating Lie derivatives of scalar, vector, covector fields and gradient of Lie derivatives of scalar or vector fields are prototyped in the header <lie_tool.h>. Drivers utilizing C or C++ are provided all of them working on the tapes generated previously as described above, i.e., no repetitive direct evaluation of the function source code is required. In the following we give an overview about the routines provided by the Lie-package.

4.1 Lie Derivatives of a Scalar Field

In order to compute the Lie derivatives of the scalar field h along the vector field $\boldsymbol{f}$ at the point $\boldsymbol{x} = \boldsymbol{x}_0 \in \mathbb{R}^n$, we need the tape numbers of active sections of $\boldsymbol{f}$ and h, the number of independent variables n and the highest derivative degree d. The values of the Lie derivatives $L_{\boldsymbol{f}}^0 h(\mathbf{x}_0), \ldots, L_{\boldsymbol{f}}^d h(\mathbf{x}_0)$ will be stored in a one-dimensional array (vector) of size $d+1$. The C function `Lie_scalarc` has the following arguments:

```
int Lie_scalarc(Tape_F,Tape_H,n,x0,d,res)
short Tape_F;        // tape tag of vector field f
short Tape_H;        // tape tag of scalar field h
short n;             // dimension n
double x0[n];        // vector x0
short d;             // highest degree d
double res[d+1];     // Lie derivatives
```

Now, consider a vector-valued maps $\boldsymbol{h} : \mathbb{R}^n \to \mathbb{R}^m$. The component maps $h_1, \ldots, h_m : \mathbb{R}^n \to \mathbb{R}$ of which $\boldsymbol{h}$ consists of are scalar fields. We understand the Lie derivative of $\boldsymbol{h}$ along $\boldsymbol{f}$ as follows

$$L_{\boldsymbol{f}}\boldsymbol{h}(\boldsymbol{x}) = \begin{pmatrix} L_{\boldsymbol{f}} h_1(\boldsymbol{x}) \\ \vdots \\ L_{\boldsymbol{f}} h_p(\boldsymbol{x}) \end{pmatrix}, \tag{34}$$

which is essentially a simplified notation for the Lie derivatives $L_{\boldsymbol{f}} h_1, \ldots, L_{\boldsymbol{f}} h_m$ of the component maps. This notation is occasionally used in nonlinear control and the associated software implementations [22, 27]. Higher order Lie derivatives $L_{\boldsymbol{f}}^k \boldsymbol{h}$ are definied by the same recursion as in (7). Note that the Lie derivative (34) of the vector-valued map $\boldsymbol{h}$ should not be confused with the Lie derivative of a vector field.

The Lie derivatives of $\boldsymbol{h}$ along $\boldsymbol{f}$ at the point $\boldsymbol{x}_0$ can be computed with the C function `Lie_scalarcv`. The resulting values of the Lie derivatives $L_{\boldsymbol{f}}^0 \boldsymbol{h}(\boldsymbol{x}_0), \ldots, L_{\boldsymbol{f}}^d \boldsymbol{h}(\boldsymbol{x}_0)$ will be stored in a two-dimensional array (i.e., a matrix) of size $m \times (d+1)$:

```
int Lie_scalarcv(Tape_F,Tape_H,n,m,x0,d,res)
short Tape_F;        // tape tag of vector field f
short Tape_H;        // tape tag of vector map h
short n;             // dimension n
short m;             // dimension m
double x0[n];        // vector x0
short d;             // highest degree d
double res[m][d+1];  // lie derivatives
```

In addition, we implemented the C++ wrapper function `Lie_scalar` which supports both calling conventions, i.e., for $m = 1$ and arbitrary $m \geq 1$. Details on the computational algorithms can be found in [29, 34].

4.2 Gradients of Lie Derivatives of a Scalar Field

The gradients

$$\mathrm{d}\, h(\boldsymbol{x}_0), \mathrm{d}\, L_{\boldsymbol{f}} h(\boldsymbol{x}_0), \ldots, \mathrm{d}\, L_{\boldsymbol{f}}^d h(\boldsymbol{x}_0) \tag{35}$$

of Lie derivatives (7) of a scalar field h along a vector field $\boldsymbol{f}$ at $\boldsymbol{x} = \boldsymbol{x}_0$ are row-vectors. They can be computed with the C function `Lie_gradientc`, where the result will be stored in a two-dimensional array of size $n \times (d+1)$:

```
int Lie_gradientc(Tape_F,Tape_H,n,x0,d,res)
short Tape_F;        // tape tag of vector field f
short Tape_H;        // tape tag of scalar field h
short n;             // dimension n
double x0[n];        // vector x0
short d;             // highest degree d
double res[n][d+1];  // gradients of Lie derivatives
```

Different scalar fields $h_1, \ldots, h_m$ can be put together in a vector-valued map $\boldsymbol{h}$ as in Eq. (34). The classical derivative of (34) is a Jacobian matrix of size $m \times n$. The Jacobian matrices

$$\mathrm{d}\, \boldsymbol{h}(\boldsymbol{x}_0), \mathrm{d}\, L_{\boldsymbol{f}} \boldsymbol{h}(\boldsymbol{x}_0), \ldots, \mathrm{d}\, L_{\boldsymbol{f}}^d \boldsymbol{h}(\boldsymbol{x}_0) \tag{36}$$

can be computed at once with the C function `Lie_gradientcv`, where the result will be stored in a three-dimensional array of size $m \times n \times (d+1)$:

```
int Lie_gradientcv(Tape_F,Tape_H,n,m,x0,d,res)
short Tape_F;        // tape tag of vector field f
short Tape_H;        // tape tag of vector map h
short n;             // dimension n
short m;             // dimension m
double x0[n];        // vector x0
short d;             // highest degree d
double res[m][n][d+1]; // Jacobians
```

In addition, we implemented the C++ function `Lie_gradient` supporting both cases (35) and (36) at once. The computation is discussed in [33].

4.3 Lie Derivatives of a Covector Field

If we consider the elements of $\mathbb{R}^n$ as comlun-vectors, the elements of the associated dual space $(\mathbb{R}^n)^*$ can be written as row-vectors. For programs, there are no differences between a vector field $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ and a covector field $\boldsymbol{\omega} : \mathbb{R}^n \to (\mathbb{R}^n)^*$. The Lie derivative of the covector field $\boldsymbol{\omega}$ along the vector field $\boldsymbol{f}$ is given by (16). The Lie derivatives $L_{\boldsymbol{f}}^0 \boldsymbol{\omega}(\boldsymbol{x}_0), \ldots, L_{\boldsymbol{f}}^d \boldsymbol{\omega}(\boldsymbol{x}_0)$ at the point $\boldsymbol{x}_0$ can be computed with the C function `Lie_covector`. The results will be stored in a two-dimensional array (i.e., a matrix) of size $n \times (d+1)$:

```
int Lie_covector(Tape_F,Tape_W,n,x0,d,res)
short Tape_F;          // tape tag of vector field f
short Tape_W;          // tape tag of covector field ω
short n;               // dimension n
double x0[n];          // vector x0
short d;               // highest degree d
double res[n][d+1];    // Lie derivatives
```

4.4 Lie Derivatives of a Vector Field (Lie Brackets)

Let $\boldsymbol{g} : \mathbb{R}^n \to \mathbb{R}^n$ be a further vector field. The Lie derivative of the vector field $\boldsymbol{g}$ along $\boldsymbol{f}$ is also called Lie bracket and defined by (12). The Lie derivatives $\mathrm{ad}_{\boldsymbol{f}}^k\, \boldsymbol{g}(\boldsymbol{x})$ along $\boldsymbol{f}$ at the point $\boldsymbol{x}_0$ can be computed with the C function `Lie_vector`. The resulting values of the Lie derivatives $\mathrm{ad}_{\boldsymbol{f}}^0\, \boldsymbol{g}(\boldsymbol{x}_0), \ldots, \mathrm{ad}_{\boldsymbol{f}}^d\, \boldsymbol{g}(\boldsymbol{x}_0)$ will be stored in a two-dimensional array (i.e., a matrix) of size $n \times (d+1)$:

```
int Lie_vector(Tape_F,Tape_G,n,x0,d,res)
short Tape_F;          // tape tag of vector field f
short Tape_G;          // tape tag of vector field g
short n;               // dimension n
double x0[n];          // vector x0
short d;               // highest degree d
double res[n][d+1];    // Lie brackets
```

The computation of these Lie derivatives using algorithmic differentiation is explained in [28, 31, 35].

5. Lie Derivatives in Nonlinear Control

The Lie derivatives described in to previous sections are often used in nonlinear control. In this section, we will sketch some application for nonlinear control systems of the form

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x}) + \boldsymbol{g}(\boldsymbol{x})\, u, \quad y = h(\boldsymbol{x}) \tag{37}$$

with the vector fields $\boldsymbol{f}, \boldsymbol{g} : \mathcal{M} \to \mathbb{R}^n$ and the scalar field $h : \mathcal{M} \to \mathbb{R}$ defined on $\mathcal{M} \subseteq \mathbb{R}^n$. Here, $\boldsymbol{x}$ denotes the state, u the input and y the output.

5.1 Exact Input-Output Linearization

Lie derivatives can also be computed along different vector fields, which results in mixed Lie derivatives such as

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

System (37) has *relative degree* r in $\boldsymbol{x}_0 \in \mathcal{M}$, if $L_{\boldsymbol{g}} h(\boldsymbol{x}) = 0, \ldots, L_{\boldsymbol{g}} L_{\boldsymbol{f}}^{r-2} h(\boldsymbol{x}) = 0$ for all $\boldsymbol{x}$ in a neighbourhood of x_0, and $L_{\boldsymbol{g}} L_{\boldsymbol{f}}^{r-1} h(\boldsymbol{x}_0) \neq 0$. Roughly speaking, the relative degree is the lowest time derivative of the output y that depends explicitly on the input u:

$$y^{(r)} = L_{\boldsymbol{f}}^r h(\boldsymbol{x}) + L_{\boldsymbol{g}} L_{\boldsymbol{f}}^{r-1} h(\boldsymbol{x})\, u \ . \tag{38}$$

These nonlinearities can be compensated exactly with an additional stabilisation of the resulting linear dynamics using the state-feedback

$$u = -\frac{p_0 h(\boldsymbol{x}) + \cdots + p_{r-1} L_{\boldsymbol{f}}^{r-1} h(\boldsymbol{x}) + L_{\boldsymbol{f}}^r h(\boldsymbol{x})}{L_{\boldsymbol{g}} L_{\boldsymbol{f}}^{r-1} h(\boldsymbol{x})}, \tag{39}$$

where $p_0, \ldots, p_{r-1}$ are the coefficients of the desired characteristic polynomial [20, Chapter 4].

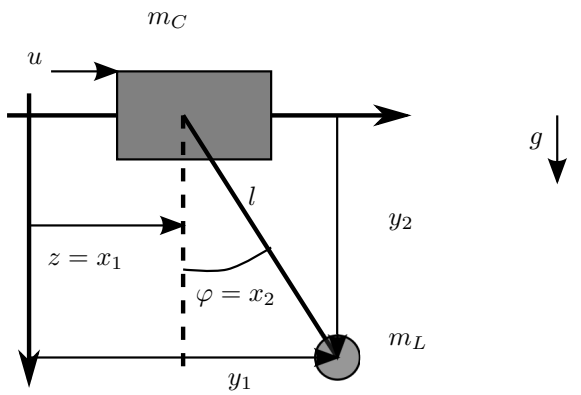

Figure 3. Gantry crane

5.2 High-Gain Observer

The matrix

$$Q(\boldsymbol{x}) := \mathbf{q}'(\boldsymbol{x}) = \begin{pmatrix} \mathrm{d}\, h(\boldsymbol{x}) \\ \mathrm{d}\, L_{\boldsymbol{f}} h(\boldsymbol{x}) \\ \cdots \\ \mathrm{d}\, L_{\boldsymbol{f}}^{n-1} h(\boldsymbol{x}) \end{pmatrix} \tag{40}$$

consistsing of gradients of Lie derivatives is called *observability matrix*. If the observability matrix is regular, we can design a high-gain observer

$$\dot{\hat{\boldsymbol{x}}} = \boldsymbol{f}(\hat{x}) + \boldsymbol{g}(\hat{\boldsymbol{x}}) u + \mathbf{k}_{HG}(\hat{\boldsymbol{x}}) \cdot (y - h(\hat{\boldsymbol{x}})) \tag{41}$$

with the state-dependend observer gain

$$\mathbf{k}_{HG}(\hat{\boldsymbol{x}}) = Q^{-1}(\hat{\boldsymbol{x}})\, (p_{n-1}, \ldots, p_0)^T \quad ,$$

see [9, 14, 30].

Note that there are many other applications of Lie derivatives in nonlinear control. For example, Lie brackets $\mathrm{ad}_{\boldsymbol{f}}^k\, \boldsymbol{g}$ are used in controllability analysis [25] and in the design of extended Luenberger observers [49].

6. Example

The usefullness of the approach is illustrated by the example of the control of a gantry crane, as shown in Figure 3.

In this underactuated system one has the mass m_c of the travelling crab, the mass m_L of the load, the length l of the cable, the earth acceleration g and the input force u. The equations of motion are given by

$$\begin{pmatrix} m_L + m_C & m_L l \cos\varphi \\ m_L l \cos\varphi & m_L l^2 \end{pmatrix} \begin{pmatrix} \ddot{z} \\ \ddot{\varphi} \end{pmatrix} + \begin{pmatrix} -m_L l \dot{\varphi}^2 \sin\varphi \\ m_L g l \sin\varphi \end{pmatrix} = \begin{pmatrix} u \\ 0 \end{pmatrix}. \tag{42}$$

Introducing the states $x_1 := z$, $x_2 := \varphi$, $x_3 := \dot{z}$, and $x_4 := \dot{\varphi}$ one obtains a state space representation of (42) which is of the form

$$\dot{\mathbf{x}} = \underbrace{\begin{pmatrix} x_3 & x_4 & * & * \end{pmatrix}^T}_{\mathbf{f}(\mathbf{x})} + \underbrace{\begin{pmatrix} 0 & 0 & * & * \end{pmatrix}^T}_{\mathbf{g}(\mathbf{x})} u \tag{43}$$

with vector fields $\mathbf{f} : \mathbb{R}^4 \to \mathbb{R}^4$ and $\mathbf{g} : \mathbb{R}^4 \to \mathbb{R}^4$.

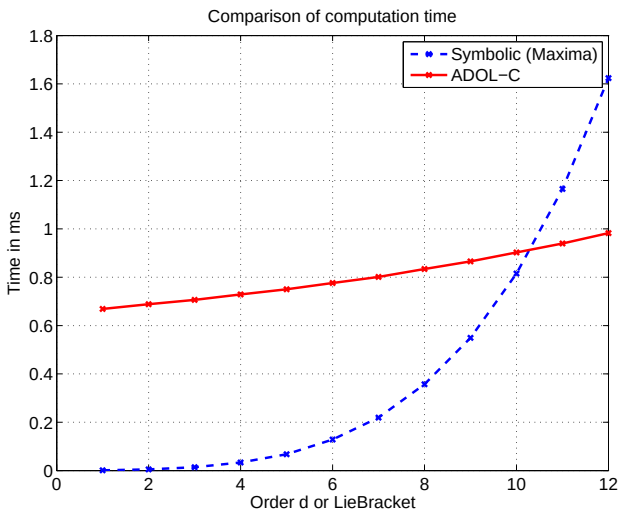

Figure 4. Comparison of computation time for Lie brackets (10) of system (43) using ADOL-C and symbolic expressions ($m_L = m_C = 1\,\text{kg}, l = 1\,\text{m}$.

The iterated Lie derivatives (Lie brackets) of the vector field $\mathbf{g}$ along the vector field $\mathbf{f}$ with $n = 4$ and $d = 12$ are calculated using ADOL-C 2.1.5 on a 2.5 GHz Phenom PC under Windows 7. The run time to compute the Lie derivatives for the model (43) of the gantry crane using ADOL-C is shown in Figure 4 (solid line). For a comparison the symbolic expressions for the calculation of the Lie brackets up to order $d = 12$ have been computed using the computer algebra system Maxima. These expressions have been exported as C and Fortan90 code after simplification using the command `trigsimp`. The generation time and sourcew code size can be found in 1. The evaluation time of the C-compiled code is given in Figure 4 (dashed line) .

Order	C-Code	Fortran90-Code	Generation
1	0.23 kB	0.16 kB	0.060 s
2	0.92 kB	0.77 kB	0.115 s
3	1.76 kB	1.54 kB	0.357 s
4	3.67 kB	3.16 kB	0.795 s
5	6.23 kB	5.46 kB	1.526 s
6	10.71 kB	9.30 kB	2.825 s
7	16.13 kB	14.13 kB	4.949 s
8	24.54 kB	21.61 kB	8.038 s
9	34.52 kB	30.77 kB	12.726 s
10	48.69 kB	43.44 kB	19.458 s
11	65.11 kB	58.63 kB	28.949 s
12	86.92 kB	79.19 kB	42.472 s

Table 1. Source code size and corresponding generation time using Maxima.

As can be seen there is some overhead using ADOL-C when computing low order Lie brackets compared to the evaluation of the compiled C-code of the symbolic expressions. However, in case of ADOL-C computational effort does only slightly increase with the order d of the Lie brackets. Furthermore one circumvents the use of large amount of C- or Fortran90 code as well as large computation times for its generation, cf. Table 1

7. Conclusions

We presented a package for computation of several kinds of Lie derivatives utilizing automatic differentiation based on the software package ADOL-C. The routines allow the convinient and efficient computation of even high order Lie derivatives of complex systems. An example illustrates the usefulness of the approach. Currently, we are working toward a native integration of the toolbox into ADOL-C.

References

[1] CppAD: A Package for Differentiation of C++ Algorithms. `http://www.coin-or.org/CppAD/`.

[2] Maxima, a computer algebra system. `http://maxima.sourceforge.net/`.

[3] R. Abraham, J. E. Marsden, and T. Ratiu. *Manifolds, Tensor Analysis, and Applications*. Springer, New York, 2nd edition, 1983.

[4] `www.autodiff.org`. Web-Portal über Automatisches Differenzieren.

[5] C. Bendtsen and O. Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM-REP-1996-17, TU of Denmark, Dept. of Mathematical Modelling, Lungby, 1996.

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

[7] G. L. Blankenship and H. G. Kwatny. Computational methods for control of dynamical systems. In N. Munro, editor, *Symbolic methods in control system analysis and design*, volume 56 of *IEE Control Engineering Series*, chapter 15. IEE Press, London, 1999.

[8] B. Christianson. Reverse accumulation and accurate rounding error estimates for Taylor series. *Optimization Methods & Software*, 1:81–94, 1992.

[9] G. Ciccarella, M. Dalla Mora, and A. Germani. A Luenberger-like observer for nonlinear systems. *Int. J. Control*, 57(3):537–556, 1993.

[10] J.-M. Cornil and P. Testud. *An Introduction to Maple V*. Springer, 2001.

[11] M. Dalla Mora, A. Germani, and C. Manes. A state observer for nonlinear dynamical systems. *Nonlinear Analysis, Theory, Methods & Applications*, 30(7):4485–4496, 1997.

[12] B. de Jager. The use of symbolic computation in nonlinear control: Is it viable? *IEEE Trans. on Automatic Control*, 40(1):84–89, 1995.

[13] S. A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software*, 32(2):195–222, jun 2006.

[14] J. P. Gauthier, H. Hammouri, and S. Othman. A simple observer for nonlinear systems — application to bioreactors. *IEEE Trans. on Automatic Control*, 37(6):875–880, 1992.

[15] R. Giering and Th. Kaminski. Recomputations in reverse mode ad. In G. Corliss, Ch. Faure, A. Griewank, Hascoët, and U. Naumann, editors, *Automatic Differentiation: From Simulation to Optimization*, chapter 33, pages 283–290.

Springer, 2002.

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

[17] A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM, 2nd edition, 2008.

[18] D. D. Holm, T. Schmah, and C. Stoica. *Geometric Mechanicas and Symmetry*. Oxford University Press, 2009.

[19] C. J. Isham. *Modern Differential Geometry for Physicists*. World Scientific, 2 edition, 2001.

[20] A. Isidori. *Nonlinear Control Systems: An Introduction*. Springer-Verlag, London, 3. edition, 1995.

[21] A. Kugi, K. Schlacher, and R. Novaki. *Symbolic Computation for the Analysis and Synthesis of Nonlinear Control Systems*, volume 2 of *Software Studies*, pages 255–264. WIT-Press, Southampton, 1999.

[22] H. G. Kwatny and G. L. Blankenship. *Nonlinear Control and Analytical Mechanics: A Computational Approach*. Birkhäuser, Boston, 2000.

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

[24] M. Lemmen, T. Wey, and M. Jelali. NSAS – ein Computer-Algebra-Packet zur Analyse und Synthese nichtlinearer systeme. Forschungsbericht Nr. 20/95, Gerhard-Mercator-Universität-GH Duisburg, Meß-, Steuer- und Regelungstechnik, 1995.

[25] H. Nijmeijer and A. J. van der Schaft. *Nonlinear Dynamical Control systems*. Springer, New York, 1990.

[26] R. Oloff. *Geometrie der Raumzeit*. Vieweg Verlag, Wiesbaden, 3. edition, 2004.

[27] V. Polyakov, R. Ghanadan, and G. L. Blankenship. Symbolic numerical computational tools for nonlinear and adaptive control. In *Proc. IEEE/IFAC Joint Symposium on Computer-Aided Control System Design*, pages 117–122, Tucson, Arizona, 1994.

[28] K. Röbenack. On the efficient computation of higher order maps $ad_f^k g(x)$ using Taylor arithmetic and the Campbell-Baker-Hausdorff formula. In Alan Zinober and David Owens, editors, *Nonlinear and Adaptive Control*, volume 281 of *Lecture Notes in Control and Information Science*, pages 327–336. Springer, 2002.

[29] K. Röbenack. Automatic differentiation and nonlinear controller design by exact linearization. *Future Generation Computer Systems*, 21(8):1372–1379, 2005.

[30] K. Röbenack. Computation of high gain observers for nonlinear systems using automatic differentiation. *Journal Dynamic Systems, Measurement, and Control*, 127(1):160–162, 2005.

[31] K. Röbenack. Computation of Lie derivatives of tensor fields required for nonlinear controller and observer design employing automatic differentiation. *Proc. in Applied Mathematics and Mechanics*, 5(1):181–184, 2005.

[32] K. Röbenack. Nonlinear controller design based on algorithmic plant description. *Mathematical and Computer Modelling of Dynamical Systems*, 13(2):193–209, 2007.

[33] K. Röbenack and K. J. Reinschke. A efficient method to compute Lie derivatives and the observability matrix for nonlinear systems. In *Proc. 2000 International Symposium on Nonlinear Theory and its Applications (NOLTA'2000), Dresden, Sept. 17-21*, volume 2, pages 625–628, 2000.

[34] K. Röbenack and K. J. Reinschke. Reglerentwurf mit Hilfe des Automatischen Differenzierens. *Automatisierungstechnik*, 48(2):60–66, 2000.

[35] K. Röbenack and K. J. Reinschke. The computation of Lie derivatives and Lie brackets based on automatic differentiation. *Z. Angew. Math. Mech.*, 84(2):114–123, 2004.

[36] R. Rothfuss and M. Zeitz. Einführung in die Analyse nichtlinearer Systeme. In S. Engell, editor, *Entwurf nichtlinearer Regelungen*, pages 3–22. Oldenbourg-Verlag, München, 1995.

[37] C. Rui, I. V. Kolmanovsky, and N. H. McClamroch. Symbolic computation in nonlinear control of multibody space systems. In *ACC'1998 (American Control Conference), Philadelphia*, 1998.

[38] J. Schaffner and M. Zeitz. Variants of nonlinear normal form observer design. In H. Hijmeijer and T. I. Fossen, editors, *New Direction in Nonlinear Observer Design*, volume 244 of *Lecture Notes in Control and Information Science*, pages 161–180. Springer-Verlag, London, 1999.

[39] H. Skaug and D Fournier. Automatic approximation of the marginal likelihood in nonlinear hierarchical models. *Computational Statistics and Data Analysis*, 51(2):699–709, 2006.

[40] S. Stamatiadis, R. Prosmiti, and S. C. Farantos. AUTO_DERIV: Tool for automatic differentiation of a FORTRAN code. *Comput. Phys. Commun.*, 127(2&3):343–355, may 2000. Catalog number: ADLS.

[41] O. Stauning and C. Bendtsen. FADBAD++: Flexible automatic differentiation using templates and operator overloading in ANSI C++. `http://www2.imm.dtu.dk/~km/FADBAD/`.

[42] V. Toth. Tensor manipulation in GPL Maxima. arXiv:cs/0503073v2 [cs.SC], 2005.

[43] V. S. Varadarajan. *Lie Groups, Lie Algebras, and Their Representation*. Springer-Verlag, 1984.

[44] A. Verma. ADMAT: Automatic differentiation for MATLAB using object oriented methods. In *SIAM workshop on object oriented methods*, pages 174–183, 1999.

[45] G. M. von Hippel. Taylur, an arbitrary-order automatic differentiation package for fortran 95. *Comput.Phys.Commun.*, 174:569–576, 2006.

[46] A. Walther, A. Griewank, and O. Vogel. ADOL-C: Automatic differentiation using operator overloading in C++. *Proc. in Applied Mathematics and Mechanics*, 2(1):41–44, 2003.

[47] P. J. Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley, 1994.

[48] S. Wolfram. *The MATHEMATICA Book*. Cambridge University Press, 1999.

[49] M. Zeitz. The extended Luenberger observer for nonlinear systems. *Systems & Control Letters*, 9:149–156, 1987.