Lie Tools Package

Version 1.03

User's Guide

Miguel Torres-Torriti

17 May 2004

Copyright and License for Non-commercial Use

Lie Tools Package is copyright:

© Miguel Torres-Torriti, 2001-2004; (migueltt@cim.mcgill.ca).

Lie Tools Package is freely available at http://www.cim.mcgill.ca/∼migueltt/ltp/ltp.html.

You may freely use this software for non-commercial purposes. It may not be used for commercial purposes without an additional license.

You can redistribute it and/oral modify it under the terms of this license (see details in the license file). You must include this license and these conditions must apply to the recipient.

This program is distributed in the hope that it will be useful, but **without any warranty**; without even the intended or implied warranty of **merchantability** or **fitness for a particular purpose**.

You use this software entirely at your own risk. If you choose to use this software, by your actions you acknowledge that any consequential damage whatever is entirely your responsibility. In no event will the copyright holder be liable to you for any damage that could arise directly or indirectly by the use of this software, or the inability to use it.

Maple® is a trademark of Waterloo Maple Inc.

Contents

1 Introduction **8**

1.1 Capabilities of the Lie Tools Package 8

1.2 Target Audience for the Lie Tools Package 9

1.3 System Requirements 9

1.4 Background Information and Reference Material 10

2 Compiling, Installing and Loading the Lie Tools Package **11**

2.1 Compiling LTP 11

2.2 Installing LTP 11

2.3 Loading LTP 11

3 Basic Notions and LTP Formalism **13**

4 Practical Applications of Lie Algebras and Groups **20**

4.1 Trajectory planning and control 20

4.2 Nonlinear filtering 23

5 Using LTP: Some Practical Examples **25**

5.1 Example 1: Simplification of Lie algebraic expressions 25

5.2 Example 1: Stabilization of a rigid body in space 26

5.2.1 Step 1: Construction of the Hall basis for the Lie algebra of indeterminates $L_4(\bar{X}_3)$ 27

5.2.2 Step 2: Calculation of the right-hand side of the Wei-Norman equation 30

5.3 Example 2: Finite dimensional realization of a nonlinear filter . . 32

6 Function Reference **35**

6.1 createLBobjects 37

6.2 phb 40

6.3 phbize 46

6.3.1 posxinphb 57

6.3.2 bracketlen 57

6.4 simpLB 59

6.5 regroupLB 61

6.6 reduceLB 63

6.7 cbhexp 68

6.8 evalLB2expr 69

6.9 calcLB 72

6.10 selectLB 75

6.11 createSubsRel 76

6.12 codeCBHcf 80

6.13 reduceLBT 81

6.14 ad 83

6.15 ead, eadr 84

6.16 pead, peadr 86

6.17 wne, wner 89

6.18 wnde 94

7 Topics for Further Improvement **98**

7.1 Generation of a k-th order CBH formula (cbhexpr) 98

7.2 Functions for setting-up and solving logarithmic equations automatically 98

7.3 Automatic controller design/synthesis functions 98

A Implementation Notes **99**

A.0.1 Highlights of some Implementation Issues 100

A.0.2 Recommendations for Improvement 100

List of Figures

1 Results for the stabilization of the rigid body. 31

2 Control inputs sequence. 38

3 Lie bracketing tree. 42

4 Flow chart for the `phb` algorithm (contd. on Fig. 5). 44

5 Flow chart for the `phb` algorithm (contd. from Fig. 4). 45

6 Flow chart for the `phbize` algorithm (contd. on Fig. 7). 52

7 Flow chart for the `phbize` algorithm (contd. from Fig. 6). 53

8 Flow chart for the `phbize` algorithm (contd. from Fig. 7). 54

9 Flow chart for the `phbize` algorithm (contd. from Fig. 7). 55

10 Flow chart for the sign removal procedure. 56

11 Flow chart for the `simpLB` function. 60

List of Tables

1 Main functions in LTP. 35

2 Auxiliary functions in LTP. 36

3 Connections between `phbize` conditions and the Lie bracket properties. 49

1 Introduction

The *Lie Tools Package* (LTP) is a collection of routines whose purpose is to make easier the task of manipulating Lie algebraic expressions arising in a variety of problems in engineering and mathematical physics, such as the solution of right-invariant differential equations evolving on Lie groups. Lie theory is a powerful tool, helpful in the analysis and design of modern nonlinear control laws for dynamical systems, nonlinear filters, and the study of particle dynamics. The practical application of Lie theory often results in highly complex symbolic expressions that are difficult to handle efficiently without the aid of a computer software tool. The aim of this package is also to facilitate and encourage further research relying on Lie algebraic computations.

LTP is implemented in Maple® and can be employed for computations involving Lie algebras of arbitrary type as it is constructed using a free Lie algebra of indeterminates as its base. The results obtained with the help of LTP can subsequently be projected onto the specific Lie algebra arising in the concrete application of interest by the use of an adequately constructed Lie algebra homomorphism.

This document is not intended to teach the user the basic concepts in Lie algebra and Lie group theory. Basic references to Lie theory can be found in Section 1.4. A brief review of the basic notions and LTP formalism is presented in Section 3.

1.1 Capabilities of the Lie Tools Package

Existing software packages, such as LiE [21] or Maple's *liesymm package* [48, 49], are very specialized and do not provide the functionality of LTP. Among other functions, LTP greatly automates and simplifies the following computations:

- Construction of ordered bases for free Lie algebras of indeterminates (Hall bases).
- Simplification of completely general Lie algebraic expressions with *symbolic coefficients*.
- Composition of exponential mappings involving indeterminates by means of Dynkin's expression for the Campbell-Baker-Hausdorff (CBH) formula.
- Construction of the Wei-Norman equations of logarithmic coordinates of flows on nilpotent Lie groups.

The above capabilities are unique to LTP and, to the best of our knowledge, are not provided by other software such as, for example, the *liealg* package,

which was recently developed by Yuly Billig and Matthias Mazzag[1], to perform specific calculations involving Kac-Moody and Virasoro algebras, and their representations.

None of the mature computer algebra systems (CAS), such as Axiom (former Scratchpad II by R. D. Jenks and D. Yun, IBM Watson Laboratories), Derive (D. R. Stoutemyer), Macsyma (Math Lab Group, MIT), Maple (B. Char, Waterloo Maple, Inc.), Mathematica (Wolfram Research, Inc.) or Reduce (A. C. Hearn), provide toolboxes with the functionality of LTP. For surveys and comparative reviews of the different CAS, the reader is referred to [6, 46], the references in [38], and the information on symbolic computation available through Internet sites, such as the comprehensive Computer Algebra Information Network (CAIN)[2] or the Symbolic Mathematical Computation Information Center[3].

1.2 Target Audience for the Lie Tools Package

LTP is intended for mathematicians, physicists, practicing engineers, as well as for classroom use to complement the theoretical aspects with practical excercises. It is assumed that the user possess some prior knowledge of the mathematical concepts in Lie algebra and group theory (for a brief review see Section 3). In this sense, LTP may assist in communicating the Lie theory concepts to students in an introductory Lie Theory or Advanced Control Systems course, but it is not intended as a stand-alone introductory tool. At the same time it is sophisticated enough to allowresearchers evaluate alternate develop control strategies, nonlinear filters or simply perform general computations based on Lie algebraic symbolic calculations.

1.3 System Requirements

LTP has the following operating system requirements:

- Maple® version 6 or higher is installed on your system.
- Recommended RAM memory: 32 MB RAM (the available memory on the computer may limit the size of the problems handled by LTP).
- For reasonable response times, a computer with a Pentium 266 MHz processor or better (or any equivalent machine) is recommended.

[1] The *liealg* package has been developed in the School of Mathematics and Statistics, Carleton University, and is available at `http://mathstat.carleton.ca/~billig/maple/`

[2] `http://www.mupad.de/CAIN/`

[3] `http://www.symbolicnet.org/`

1.4 Background Information and Reference Material

The theory of Lie algebras and groups was originally conceived by the Norwegian mathematician Sophus Lie (1842-1899) as a tool for the solution of differential equations and has since then become a discipline in its own right. Lie theory brings together the mathematical disciplines of algebra and geometry to produce results relying on group-theoretic and differential geometric developments.

Important basic references in Lie algebras and group theory are the book by V.S. Varadarajan and J.-P. Serre [43, 36]. A basic reference which is intended to serve an audience of physicists and engineers is the book by R. Gilmore [13]. For a comprehensive review of the applications of the Lie theory also see [13] and the book by J. G. F. Belinfante [3], which also presents a surver of some computational methods.

Results in Lie theory have proved essential in the study of kinematical symmetries in both classical and quantum mechanics [9, 33], the construction of nonlinear filters [7, 23], the analysis of dynamical systems, and the design of feedback control laws for nonlinear systems [32, 16, 27]. The use of Lie theory in the study of the symmetries of differential equations is described in [38] from a practical perspective. The application of Lie theory to the analysis and control of robotic systems is found in [27, 35, 32] and references therein.

Despite the attention that Lie theory has received in a variety of fields, it has been limited mainly because of the complexity of the symbolic calculations, which are often prohibitively difficult to perform by hand, and the lack of adequate software capable of handling completely general symbolic Lie algebraic expressions.

2 Compiling, Installing and Loading the Lie Tools Package

The distribution of LTP already includes a compiled version of LTP and therefore compilation and installation are not necessary unless you decide to change or add new procedures to the package. *Note that if you decide to recompile LTP, you must check that there do not already exist library files for LTP in your target directory. You must delete or move the LTP library files to another location before compiling the package.*

2.1 Compiling LTP

The following steps are required for compilation:

1. Update the `libname` directory path in the file `ltp.mws` with the directory where you wish to install your package. This is in order to not interfere with Maple's repository.
2. Execute the Maple archive command (`march`) and the `savelib` command written in `ltp.mws`.

If you wish to recompile, then erase the old repository created for `lt` and repeat the above steps. The package can be compiled from scratch by simply executing the whole worksheet, however the current `libname` path must be set appropriately.

See the Maple documentation for further details on creating packages.

2.2 Installing LTP

LTP will be installed in the directory specified by the `libname` path at the time of saving the new library (see the above section on compiling the package), therefore nothing needs to be done, except if one wishes to relocate the library to a different directory.

2.3 Loading LTP

To load the LTP package first ensure that the `libname` path has been set to include the directory where the package was stored after compilation (or to where

it was moved). To add a new directory to the `libname` path variable, simply execute: `libname:="c:/YourMapleLib/lt",libname;` or `libname:=libname,"../../lib/"` if you prefer to use relative paths rather absolute ones. However, for the relative paths approach to work fine your current directory must be the appropriate one, so either you use the Maple command `currentdir` to set the directory or you be sure to open the file by double-clicking on it, or launching `xmaple` from the directory where the file is.

Load the library by executing the command `with(lt)` at the Maple prompt "`>`". Particular functions within the library can also be accessed without loading the whole module by typing `lt[function](args)`, e.g. `lt[createLBobjects](3,4)`, followed by a semi-colon or colon, the latter for silent execution of the command (i.e. the results are not shown on the screen).

Upon loading, the symbolic Lie product operator, denoted in Maple by `&*`, is defined. With this notation, the Lie product (or Lie bracket) of two indeterminates X and Y, traditionally denoted in the mathematical texts by the bracket $[X, Y]$, would be represented in Maple as `X &* Y`.

The `&` symbol preceding the `*` indicates the operator is a *user defined* operator. Instead of `*` we could have chosen any other symbol, letter or even word, however since the $[X, Y]$ is a special type of product, it seems more natural to use the computer symbol for multiplication in the definition of our *custom-built multiplication* operator. The Lie product operator `&*` is declared as being *multilinear* (in Maple this means that the operator distributes over the addition), and non-associative, in other words the operator does not distributes over the Lie product. Note that due to the latter, it is convenient to treat the operator as an *infix* operator rather than using *prefix* notation, however this is not compulsory.

When the package is shutdown (unloaded) the `&*` operator is unassigned (freed) and its properties are removed.

3 Basic Notions and LTP Formalism

This section provides the basic notions and formalism that constitutes a general framework for calculations relevant to the behaviour and properties of dynamical systems. The LTP package relies on this formalism as it is designed to aid analysis and synthesis of systems of basically unlimited Lie algebraic structure. In lay terms, the underlying idea of this formalism is to introduce abstract, but precise algebraic constructs which, under adequately constructed mappings, project directly onto the corresponding constructs acting on manifolds on which the particular systems evolve; see for example Remark 4.1.

To this end, let $\{X_1, \dots, X_m\}$ denote a set of indeterminates. For brevity of notation, let $\bar{X}_m = (X_1, \dots, X_m)$. Let $A(\bar{X}_m)$ denote the *free associative algebra* (over $\mathbb{R}$) of noncommutative polynomials in the indeterminates $X_1, X_2, \dots, X_m$. Recall that, given a set $\bar{X}_m$, a free associative algebra on the set $\bar{X}_m$ over $\mathbb{R}$, is an associative algebra $A(\bar{X}_m)$ over $\mathbb{R}$, together with a mapping $i : \bar{X}_m \to A(\bar{X}_m)$, with the following *universal property*: for each associative algebra A_0 and each mapping $f : \bar{X}_m \to A_0$, there exists a unique homomorphism of algebras $F : A(\bar{X}_m) \to A_0$ such that $f = F \circ i$. Members of $A(\bar{X}_m)$ have the form of finite linear combinations $\sum_I a_I X_I$, where the summation is over all possible multi-indices $I = (i_1, \dots, i_k)$, with $i_j \in \{1, \dots, m\}$, for $j = 1, \dots, k$, $k \in \mathbb{N}$, in which the coefficients a_I are real numbers. Here $X_I = X_{i_1} \cdots X_{i_k}$, and $X_{I=\emptyset} = 1$, where, in general, $X_i X_j \neq X_j X_i$ as implied by noncommutativity.

Let a Lie product $[X_i, X_j]$ of two indeterminates be defined as the noncommutative polynomial $X_j X_i - X_i X_j$. With this definition of the Lie product $A(\bar{X}_m)$ becomes a Lie algebra. Let $L(\bar{X}_m)$ be the subalgebra of $A(\bar{X}_m)$ generated by $\bar{X}_m$. The elements of $L(\bar{X}_m)$ are referred to as *Lie polynomials*.

Further, let $\hat{L}(\bar{X}_m)$ denote the Lie algebra of Lie series in $X_1, \dots, X_m$. The elements of $\hat{L}(\bar{X}_m)$ are formal series of the type $\sum_{i=1}^{\infty} a_i S_i$, where a_i are coefficients in K and $S_i \in L(\bar{X}_m)$. Clearly, any element $Z \in \hat{L}(\bar{X}_m)$ can be written as a formal infinite series $\sum_I a_I X_I$ in the indeterminates $X_1, \dots, X_m$, in which X_I is some monomial in $X_1, \dots, X_m$ and $a_{I=\emptyset} = 0$.

For any element in $Z \in \hat{L}(\bar{X}_m)$ the formal power series

$$e^Z = \sum_{k=0}^{\infty} \frac{1}{k!} Z^k \tag{1}$$

is well defined because $1 \notin \hat{L}(\bar{X}_m)$. Here, Z^k are infinite series in the indeterminates $X_1, \dots, X_m$ obtained by the natural multiplication rule for the component monomials of Z, $X_I X_J = X_{I*J}$, where $I * J$ is the juxtaposition (concatenation) of the components of the multi-indices I and J. The set $\hat{G}(\bar{X}_m) = \{e^Z : Z \in \hat{L}(\bar{X}_m)\}$ is called the *set of exponential Lie series* in the indeterminates $X_1, \dots, X_m$.

Note that, due to the antisymmetry property and the Jacobi identity of the Lie product, not all the elements of a Lie algebra $L(\bar{X}_m)$ are linearly independent. A procedure to construct a basis for any Lie algebra of indeterminates, while taking into account the dependencies imposed by the antisymmetry and the Jacobi identities, involves selecting some of the Lie product of $X_1, \ldots, X_m$, which can, for example, be carried out in accordance with the rules given below, see [32, 36, 4].

Definition 3.1 - Hall basis (HB). *Let B denote the basis for $L(\bar{X}_m)$, and let B_i be the i-th element in this basis. Let the* length *(order) of a Lie product G, $l(G)$, be defined as the number of indeterminates in the expansion of G, also given recursively by:*

$$\begin{aligned} l(X_i) &= 1 \qquad\qquad i = 1, \ldots, m \\ l([G, H]) &= l(G) + l(H) \end{aligned}$$

where G and H are Lie products.

Then a Hall basis *is an ordered set of Lie products $\{B_i\}$ such that:*

1. *$X_i \in B, \ i = 1, \ldots, m$*
2. *If $l(B_i) < l(B_j)$ then $B_i < B_j$*
3. *$[B_i, B_j] \in B$ if and only if*
   (a) *$B_i, B_j \in B$ and $B_i < B_j$ and*
   (b) *either $B_j = X_k$ for some k or $B_j = [B_p, B_q]$ with $B_p, B_q \in B$ and $B_p \leq B_i$.*

The proof that a Hall basis indeed constitutes a basis for the Lie algebra $L(\bar{X}_m)$ is found in [14, 36].

Remark 3.1 *The basis presented above, although already know by P. Hall, was first introduced by M. Hall [14] and pertains to one of the possible ways in which a basis for $L(\bar{X}_m)$ could be constructed. In fact, the above construction was generalized by Scützenberger [34] by weakening the degree condition 2 in Definition 3.1. Viennot, [44], further relaxed condition 2 replacing it by:*

2'. *If $[B_i, B_j] \in B \setminus \bar{X}_m$ then $B_i \in B$ and $B_i < [B_i, B_j]$.*

The last is so general that it includes the Lyndon basis and the Širšov basis [37], which is not the case with the original bases of M. Hall; see [30, 24, 25] for a comprehensive exposition of different bases constructions.

The choice of the above, rather restrictive, basis construction was deliberate for the purpose of the LTP because it is the one most often used in the engineering literature. Nevertheless, it is worth noting that other bases construction using condition 2' instead of 2 could prove more advantageous in applications for which a particular choice of coordinate system is desirable.

With regard to the algorithmic implementation of the package, a choice of Lyndon basis would possibly offer some advantages. A Lyndon basis *is defined as a set of alphabetically ordered Lyndon words over a given alphabet A (which is defined as a set of letters). A* Lyndon word *is any nonempty, finite sequences of letters which precedes all its nontrivial proper right factors in any alphabetically ordered set on A; i.e. w is a Lyndon word if for each nontrivial factorization in terms of sub-words u and v, w = uv, the word w precedes v. From Theorem 5.1 in [30] it follows that each element of the Lyndon basis can be uniquely rewritten as an element of a Hall basis satisfying Definition 3.1 with condition 2 replaced by 2'. It is the* rewriting system *of [25] that provides a procedure that allows one to translate any Lyndon word into an element of a Hall basis, thus permitting to use words in place of their corresponding Lie bracket expressions. For example, given the alphabet* $\{1, 2, 3\}$*, the sequence of Lyndon words: 12, 112, 212, 1213, 1223, 3323, translates, in a one-to-one way, into the following Hall basis elements:* $[X_1, X_2]$*,* $[X_1, [X_1, X_2]]$*,* $[X_2, [X_1, X_2]]$*,* $[[X_1, X_2], [X_1, X_3]]$*,* $[[X_1, X_2], [X_2, X_3]]$*,* $[X_3, [X_3, [X_2, X_3]]]$ *in* $L(\bar{X}_3)$*. On the one hand, operating on words (character strings) requires less memory, but on the other hand, operations on Lie brackets (binary tree structures) can generally take less processing time than those involving words.*

Let $L_k(\bar{X}_m) \subset L(\bar{X}_m)$ denote the *free nilpotent Lie algebra of order k*, i.e. a Lie algebra that can be identified with the quotient $L(\mathcal{X}_m)/I_k$, where $I_k \subset L(\bar{X}_m)$ is the ideal spanned by all elements of the Hall basis of order strictly greater than k. Hence, $L_k(\bar{X}_m)$ can be formed by assuming that all the Lie products in $L(\bar{X}_m)$ of degree strictly greater than k are equal to zero. The above procedure can still be employed to construct bases for $L_k(\bar{X}_m)$ simply by forming all the Lie products that satisfy the above properties and whose length does not exceed k.

By the result of Campbell, Baker, and Hausdorff, known as the *CBH formula*, it follows that $\hat{G}(\bar{X}_m)$ is closed under multiplication, and is in fact a group, as it can be verified that $e^Z e^{-Z} = 1$, for any $Z \in \hat{G}(\bar{X}_m)$. Moreover, the map $exp : \hat{L}(\bar{X}_m) \to \hat{G}(\bar{X}_m)$ is a bijection from $\hat{L}(\bar{X}_m)$ onto $\hat{G}(\bar{X}_m)$. It follows that for any $Z_1, Z_2 \in \hat{L}(\bar{X}_m)$ we can compute a unique $Z_3 \in L(\bar{X}_m)$ such that

$$e^{Z_1} e^{Z_2} = e^{Z_3} \tag{2}$$

The way to compute Z_3 is also delivered by the CBH formula which, in Dynkin's

form, is given by [36, 39]:

$$
\begin{aligned}
Z_3 &= \sum_{m=1}^{\infty}\sum \frac{(-1)^{m-1}(ad_{Z_2})^{q_m}(ad_{Z_1})^{p_m}\cdots(ad_{Z_2})^{q_1}(ad_{Z_1})^{p_1}}{m\sum_{i=1}^{m}(p_i+q_i)\prod_{i=1}^{m}(p_i!q_i!)} \\
&= Z_1+Z_2+\frac{1}{2}[Z_1,Z_2]+\frac{1}{12}([[Z_1,Z_2],Z_2]-[[Z_1,Z_2],Z_1]) \\
&\quad -\frac{1}{48}([Z_2,[Z_1,[Z_1,Z_2]]]+[Z_1,[Z_2,[Z_1,Z_2]]])+\ldots
\end{aligned}
\tag{3}
$$

where the inner sum ranges over all m-tuples of pairs of nonnegative integers (p_i, q_i) such that $p_i + q_i > 0$. In (3), with the exception of the last term, the symbol ad_X denotes the mapping $ad_X : Y \mapsto [X, Y]$ for all $Y \in L(\bar{X}_m)$, which is an endomorphism of $L(\bar{X}_m)$ underlying the *adjoint representation* of $L(\bar{X}_m)$, defined as the mapping $X \mapsto ad_X$. With some abuse of notation, the last term in (3) should be understood differently and must be evaluated as follows: $ad_Z^n = 1$ if $n = 0$, $ad_Z^n = Z$ if $n = 1$, and $ad_Z^n = 0$ if $n > 1$.

It is worth noticing that the group $\hat{G}(\bar{X}_m)$ is not a Lie group because it is infinite dimensional.

As the package is primarily a tool for the analysis of dynamical systems, it will be applied in the context of groups of transformations acting on the underlying manifold on which the system evolves, see [43]. For analytic systems whose accessibility Lie algebras, [32], are finite dimensional, such groups of transformations can be given the structure of Lie groups; see [29]. It is hence helpful to define $G_k(\bar{X}_m)$, a nilpotent version of $\hat{G}(\bar{X}_m)$:

$$
G_k(\bar{X}_m) \stackrel{def}{=} \{e^Z : Z \in L_k(\bar{X}_m)\} \tag{4}
$$

The group $G_k(\bar{X}_m)$ is now a Lie group with Lie algebra $L_k(\bar{X}_m)$, see [43]. For a systematic development it is assumed here that all groups of transformations act from the right on the underlying manifolds M. With this notation, for $x \in M$, the expression xe^Z denotes the value of a group action e^Z at a point $x \in M$, [36, p. LG 4.11] or [43, p. 74].

One of the many applications of the LTP package involves the solution of differential equations defined on Lie groups. As will be explained later, the trajectories of these equations relate (through a Lie group homomorphism, see Remark 4.1) to trajectories evolving on $G_k(\bar{X}_m)$. It is hence convenient that $G_k(\bar{X}_m)$ is equipped with a coordinate system. Such a coordinate system can be constructed in terms of a Hall basis and has the advantage of being global (consisting of a single chart) since $G_k(\bar{X}_m)$ is nilpotent, see [41]. In full rigour, if $\{B_1, B_2, \ldots, B_r\}$ is the r-dimensional Hall basis for a given nilpotent Lie algebra $L_k(\bar{X}_m)$, then any element P in the Lie group $G_k(\bar{X}_m)$ has the following unique representation, [20]:

$$
P = e^{\gamma_1 B_1} e^{\gamma_2 B_2} \cdots e^{\gamma_r B_r} \tag{5}
$$

The map $P \to (\gamma_1, \gamma_2, \ldots, \gamma_r)$ establishes a global diffeomorphism between $G_k(\bar{X}_m)$ and $\mathbb{R}^r$ and is thus a global coordinate chart on $G_k(\bar{X}_m)$. The coordinate system so just introduced falls into the category of Lie-Cartan coordinate systems of the second kind [36, 32]. Here, we will refer to it using the name of γ-coordinates.

Equation (5) can be viewed as a way to represent an arbitrary group action as a composition of elementary group actions defined in terms of the elements of the Hall basis of the Lie algebra associated with the group. This fact has been exploited by Wei and Norman in the solution of right-invariant parametric differential equations evolving on $G_k(\bar{X}_m)$:

$$\begin{aligned} \dot{S}(t) &= \left(\sum_{i=1}^{m} X_i u_i(t)\right) S(t) \\ S(0) &= I \end{aligned} \tag{6}$$

where $m < \infty$ (finite), X_i are indeterminate operators independent of t that generate $L_k(\bar{X}_m)$ under the commutator product $[X_i, X_j] = X_j X_i - X_i X_j$, and u_i are scalar functions of t. Here, as $S(0) \in G_k(\bar{X}_m)$, $S(t)$ evolves on $G_k(\bar{X}_m)$.

Therefore, the solution to (6) is given by the product of exponentials:

$$S(t) = e^{\gamma_1(t)B_1} e^{\gamma_2(t)B_2} \cdots e^{\gamma_r(t)B_r} = \prod_{i=1}^{r} e^{\gamma_i(t)B_i} \tag{7}$$

where $\{B_1, B_2, \ldots, B_r\}$ is the Hall basis for the Lie algebra $L_k(\bar{X}_m)$, and the γ_i are scalar functions of time, see [41, 45]. Without the loss of generality, it may be assumed that $B_i = X_i$, for $i = 1, \ldots, m$.

Remark 3.2 *The representation (7) of the solution to equation (6) is not unique. Alternatively, the solution to (6) can be represented using the Lie-Cartan coordinates of the first kind, i.e. it is possible to write*

$$S(t) = e^{\left(\sum_{i=1}^{r} \theta_i(t) B_i\right)}$$

where $\theta_i : \mathbb{R} \to \mathbb{R}$, $i = 1, 2, \ldots, r$, are the "coordinates" of such a solution; see [22].

Remark 3.3 *For an arbitrary set of indeterminates $\bar{X}_m$ the Lie algebra $L(\bar{X}_m)$ is really infinite dimensional. A unique solution of (6), however, still exists for every Lebesgue integrable control function $u \overset{def}{=} [u_1, \ldots, u_m]$ defined on a finite interval $[0, T]$. This solution is known to evolve on $\hat{G}(\bar{X}_m)$, see [40, Prop. 3.1]. Furthermore, the solution of (6) can be written in terms of a formal power series in the indeterminates $X_1, \ldots, X_m$, denoted by $Ser(u)$, and known as the Chen-Fliess series, see [12, Theorem III.2, p. 22]. The coefficients in*

$Ser(u)$ are iterated integrals of the control functions $u_1, \ldots, u_m$ and for any t, $S(t) = Ser(u_t)$, where $S(u_t)$ denotes the Chen-Fliess series with coefficients evaluated over $[0, t]$.

It has been shown in [41] that the expression of the the solution of (6) in the form of a Chen-Fliess series $Ser(u_t)$ is equivalent to the expression in the form of a product of exponentials, i.e. for any given control function u, there exist functions $\gamma_i : [0, T] \to \mathbb{R}$, $i = 1, 2, \ldots$ such that (7) is valid with the product performed over all members of the P. Hall basis $\{B_i; i = 1, 2, \ldots\}$ for $L(\bar{X}_m)$.

A particularly convenient formalism based on chronological algebras introduced for nonstationary vector fields has been introduced in [1] and permits to re-write the Chen-Fliess series in a very compact and symbolically tractable form, in which the iterated integrals are re-expressed in terms of the chronological product *operations, see [18, 17]. The chronological calculus has also been shown useful in the calculation of the logarithm of the Chen-Fliess series, see [31]; however, the expressions derived there, although relatively simple, provide a series expansion of this logarithm which may contain linearly dependent terms.*

The γ-coordinates in (7) are shown to satisfy a set of nonlinear differential equations as is implied by the following derivation, see also [45, 32].

Differentiating (7) yields,

$$\dot{S}(t) = \frac{dS(t)}{dt} = \sum_{i=1}^{r} \dot{\gamma}_i(t) \prod_{j=1}^{i-1} e^{\gamma_j B_j} B_i \prod_{j=i}^{r} e^{\gamma_j B_j} \tag{8}$$

Multiplying both sides of (8) by $S(t)^{-1}$ from the right and using the *exponential formula* (see [43, p. 40]):

$$\begin{aligned} (e^X)Y(e^{-X}) &= Y + [X, Y] + \frac{1}{2!}[X, [X, Y]] + \frac{1}{3!}[X, [X, [X, Y]]] + \ldots \\ &= \sum_{k=0} \frac{1}{k!}(ad_X^k)Y \\ &= (e^{ad_X})Y \end{aligned} \tag{9}$$

yields,

$$\dot{S}(t)S^{-1}(t) = \sum_{i=1}^{r} \dot{\gamma}_i(t) \prod_{j=1}^{i-1} e^{\gamma_j ad_{B_j}} B_i \tag{10}$$

$$= \sum_{i=1}^{r} B_i u_i(t) \tag{11}$$

with $u_i(t) = 0$ for $i = m+1, \ldots, r$, as $S(t)$ satisfies (6).

Equating the coefficients on both sides of the last equality gives:

$$\underbrace{\begin{bmatrix} u_1(t) \\ u_2(t) \\ \vdots \\ u_r(t) \end{bmatrix}}_{u} = \underbrace{\begin{bmatrix} \xi_{11}(\gamma) & \cdots & \xi_{1r}(\gamma) \\ \vdots & \ddots & \vdots \\ \xi_{r1}(\gamma) & \cdots & \xi_{rr}(\gamma) \end{bmatrix}}_{\xi(\gamma)} \underbrace{\begin{bmatrix} \dot\gamma_1(t) \\ \dot\gamma_2(t) \\ \vdots \\ \dot\gamma_r(t) \end{bmatrix}}_{\dot\gamma}, \qquad \gamma(0) = 0 \tag{12}$$

where the $\xi_{ij}(\gamma)$ are analytic functions of the γ_i's. Clearly, $\gamma(0) = 0$ since $S(0) = I$.

It is worth noting that there exists a chain of ideals $0 \subseteq \mathcal{I}_1 \subseteq \mathcal{I}_2 \subseteq \cdots \subseteq \mathcal{I}_r = L_k(\bar{X}_m)$ where each $\mathcal{I}_n$ is exactly of dimension n. The order of the elements in the Hall basis $\{B_1, \ldots, B_r\}$ is such that is the ideal $\mathcal{I}_n$ is generated by $\{B_n, \ldots, B_r\}$, which implies that the multiplication table for $L_k(\bar{X}_m)$ satisfies:

$$[B_i, B_j] = \sum_{n=i}^{r} c_n^{ij} B_n, \qquad \text{for } i > j \tag{13}$$

It can be shown, see [45], that such a multiplication table implies that $\xi(\gamma)$ is lower triangular and invertible for all t. Hence, (12) yields the system of differential equations for the computation of the γ-coordinates in explicit form:

$$\dot\gamma(t) = \xi^{-1}(\gamma) u(t), \qquad \gamma(0) = 0 \tag{14}$$

Equation (14) will be referred to as the *Wei-Norman equation*. Its solution delivers $S(t)$ of (7) which solves (6). The explicit formulæfor the solution of (14), in terms of iterated integrals, are also given in [41]; see also [18, Thm. 4.10, p. 297].

4 Practical Applications of Lie Algebras and Groups

The practical relevance of the concepts introduced in the previous section and a few applications are discussed in this section. A rigorous exposition of the examples presented and the associated assumptions can be found in [16, 23, 32] and references therein.

4.1 Trajectory planning and control

A wide class of nonlinear control systems can be described by an ordinary differential equation which is affine in the controls:

$$\dot{x} = f_0(x)u_0 + f_1(x)u_1 + \ldots + f_m(x)u_m = f(x, u) \tag{15}$$

where $x \in \mathcal{M}$ is the state of the system, $\mathcal{M}$ is the manifold on which the system evolves, $f_i : \mathcal{M} \to T\mathcal{M}$, $i = 0, 1, \ldots, m$, are analytic vector fields defined on $\mathcal{M}$, and $u_i \in \mathbb{R}$, $i = 0, 1, \ldots, m$, are scalar, measurable, control functions. The control problem for (15), with $u \overset{def}{=} [u_0, u_1, \ldots, u_m] \in \mathbb{R}^{m+1}$, becomes challenging if $m+1 < n$. Equation (15) can be thought to represent both driftless systems, and systems with drift (if $u_0 = 1$). Practical examples can be found in [32, 27, 35] and include robotic manipulators, mobile robots, underwater vehicles, and rigid bodies in space.

With reference to systems described by (15), the theory of Lie algebras and groups is known to be helpful in the following:

- Establishing the controllability properties of the system.
- Developing control laws that stabilize the system to a given equilibrium point, or ensure tracking of a desired reference trajectory.

Chow's Theorem delivers a conclusive result for the determination of complete controllability for driftless system (15). Chow's result involves the verification of the *Lie algebra rank condition* (LARC), see [32]:

$$L(f_0, f_1, \ldots, f_m)(p) = T_pM \tag{16}$$

for any $p \in M$, where $L(f_0, f_1, \ldots, f_m)(p) \overset{def}{=} \text{span}\{f(p) \in T_pM | f \in L(f_0, f_1, \ldots, f_m)\}$ and T_pM is the tangent space to M at p.

The LARC hence requires the construction of a spanning set (ideally a basis) for the Lie algebra of vector fields $L(f_0, f_1, \ldots, f_m)$. To this end the LTP package is used as follows. For a sufficiently large k, a Hall basis $\{B_1, B_2, \ldots, B_r\}$ is first generated for $L_k(\bar{X}_{m+1})$ and then each Lie product B_i, $i = 1, 2, \ldots, r$

of this basis is mapped into a corresponding Lie product of vector fields in $L(f_0, f_1, \ldots, f_m)$ by using the *evaluation map*, defined by $Ev : X_i \to f_i$, for $i = 0, 1, \ldots, m$, which assigns f_i to X_i, $i = 0, 1, \ldots, m$, in any formal Lie product B_i, $i = 1, 2, \ldots, r$. The evaluation map becomes the *canonical Lie algebra homomorphism* if $L(f_0, f_1, \ldots, f_m) = L_k(f_0, f_1, \ldots, f_m)$, i.e. when $L(f_0, f_1, \ldots, f_m)$ is nilpotent.

For systems with drift the LARC only ensures accessibility of the system, i.e. that the reachable set at any $p \in M$ has a non-empty interior, see [32]. The computation of a basis for $L(f_0, f_1, \ldots, f_m)$ is however still useful since the dimension of the set $L(f_0, f_1, \ldots, f_m)(p)$ and the highest order of the Lie products appearing in $L(f_0, f_1, \ldots, f_m)(p)$ reveal the difficulty of controlling (15).

Assuming that system (15) is completely controllable, a variety of Lie algebraic-based control synthesis methods have been proposed in the literature, see for example [27, 32].

Pivotal to controllability considerations, the design, and the derivation of control strategies for system (15) is the capability to generate equivalent system motions in directions outside the span of the vector fields f_i, $i = 0, 1, \ldots, m$. For simplicity of exposition, assume at first that piece-wise constant switching controls are employed for this purpose. Then, such motions can be achieved by concatenation of trajectories of (15) which, at every point $x \in M$, are tangent to elements of $\mathrm{span}\{f_i, i = 0, 1, \ldots, m\}$ at $x \in M$. To this end, the LTP package proves helpful in determining the vector field, which over a given interval of time T, yields motions equivalent to any desired concatenation. More precisely, let $0 = t_0 < t_1 < t_2 < \ldots < t_s = T$ be a partition of a given interval $[0, T]$, let $\bar{\epsilon} \stackrel{def}{=} \{\epsilon_i = t_i - t_{i-1}; i = 1, \ldots, s\}$, and let $\bar{u}$ be a sequence of constant controls by $\bar{u} \stackrel{def}{=} \{u^i \in \mathbb{R}^{m+1}; i = 1, \ldots, s\}$ each of which is applied over $[t_{i-1}, t_i]$. Additionally, let $g_i(x) \stackrel{def}{=} f(x, u^i)$, $i = 1, \ldots, s$, be the vector fields constituting the right-hand sides of system (15) that correspond to u^i, $i = 1, \ldots, s$. Employing the CBH formula (3), the package can then help to determine the vector field $\bar{f}(x, \bar{u}, \bar{\epsilon})$ such that for any $p \in \mathbb{R}^n$, the solution to (15), $x^{\bar{u}}$, corresponding to the sequential application of the constant controls satisfies:

$$x^{\bar{u}}(T) = p\, e^{\epsilon_1 g_1} \cdots e^{\epsilon_s g_s} = p\, e^{T\bar{f}} \tag{17}$$

where, $e^{\epsilon g}$ denotes the flow of the differential equation $\dot{x} = g$ so that $p\, e^{\epsilon g}$ is the solution of this equation with initial condition $p \in M$, evaluated at time ϵ.

For arbitrary $\bar{u}$, $\bar{\epsilon}$, equation (17) is guaranteed to hold only if the Lie algebra of vector fields $L(f_0, f_1, \ldots, f_m)$ is nilpotent, all the vector fields involved are real, analytic, and complete, as then the CBH formula is known to hold globally; see [29, p. 95] and [43, p. 195]. When $L(f_0, f_1, \ldots, f_m)$ is not nilpotent, the package can only provide an approximate expression for $\bar{f}$, and generally, (17) will be valid only locally, i.e. for sufficiently small T. A natural way to ob-

tain such an approximation is to employ the LTP package using $L_k(\bar{X}_{m+1})$ in place of $L(\bar{X}_{m+1})$, where $L(f_0, f_1, \ldots, f_m)$ is the image of $L(\bar{X}_{m+1})$ under the evaluation map Ev. Under the same evaluation map $L_k(\bar{X}_{m+1})$ maps into $Ł_k(f_0, f_1, \ldots, f_m)$, a truncated version of $L(f_0, f_1, \ldots, f_m)$.

Remark 4.1 *The use of $L_k(\bar{X}_{m+1})$ in place of $L(\bar{X}_{m+1})$ is a valid way to obtain an approximation of $\bar{f}$ in view of the results in [40] which also provide a link between the purely abstract algebraic formalism of Section 3 and the actual solution of (15).*

More precisely, if u is a Lebesgue integrable control function on $[0, T]$, then the image of the Chen-Fliess series $Ser(u)$, under the evaluation map Ev, $Ev(Ser(u))$, is a formal series of partial differential operators acting on smooth functions defined on the manifold $\mathcal{M}$. If $\phi \in \mathcal{C}^\infty(\mathcal{M})$ then application of $Ev(Ser(u))$ to ϕ yields a formal series of $\mathcal{C}^\infty$ functions on $\mathcal{M}$ denoted $Ev(Ser(u))(\phi)$. In [40, Prop. 4.3, p. 698], this series is actually shown to converge to $\phi \circ x^u$, the composition of ϕ with the solution of system (15) corresponding to u. Specifically, it was shown that: for analytic, complete vector fields $f_0, f_1, \ldots, f_m$, any compact set $U \subset \mathbb{R}^{m+1}$ and, any compact set $K \subseteq \mathbb{R}^n$, there exists a time horizon $T > 0$ such that the formal power series $Ev(Ser(u_t))(\phi)(p)$ (evaluated at $p \in M$) actually converges uniformly to $\phi \circ x^u(t)$ for $t \in [0, T]$, where $x^u(t) : [0, T] \to \mathcal{M}$ is the solution of (15), with $x(0) = p$, for any $p \in K$, and any integrable $u : [0, T] \to U$. Furthermore, a precise upper bound was obtained in the same reference for the difference between $\phi \circ x^u$ and the N-th partial sum of the series $Ev(Ser(u_t))(\phi)(p)$ for $t \in [0, T]$:

$$|\phi \circ x^u(t) - Ev(Ser_N(u_t))(\phi)(p)| \leq D_N t^{N+1} \tag{18}$$

for all $N \in \{1, 2, 3, \ldots\}$, $p \in K$, u as defined above, and all $t \in [0, T]$, where $Ser_N(u_t)$ denotes the truncated series *obtained by considering terms only up to order N in the Chen-Fliess series $Ser(u)$, and D_N is a constant.*

In view of the last remark, an "approximation" to $\bar{f}$ can be calculated employing the LTP package for repeated application of the CBH formula to perform formal calculations associated with the composition of the formal exponential maps on $L_k(\bar{X}_{m+1})$. Precisely, if $Y_i \in L_k(\bar{X}_{m+1})$ corresponds to g_i via $Ev(Y_i) = g_i$, for $i = 1, 2, \ldots, s$, then repeated application of the CBH formula yields $\bar{Y} \in L_k(\bar{X}_{m+1})$ such that

$$e^{\epsilon_1 Y_1} \cdots e^{\epsilon_s Y_s} = e^{T\bar{Y}} \tag{19}$$

Also, it follows that $\exp(T\bar{Y})$ can be expressed as $Ser_k(\bar{u_T})$ (a partial sum $Ser(\bar{u}_T)$ containing only Lie monomials up to order k). Therefore, it is only in the sense of (18) that $\tilde{f} \stackrel{def}{=} Ev(\bar{Y})$ can be considered an approximation to $\bar{f}$. Note that (18) can be applied with ϕ as coordinate functions on M

which immediately implies increasing proximity of trajectories $p\, e^{t\tilde{f}}$ and $p\, e^{t\bar{f}}$ for $t \in [0, T]$, with increasing order k of nilpotent truncation.

Rather than using piece-wise constant controls, it is often more convenient to calculate the flows of dynamical systems such as (15) with control functions u_i, $i = 1, \ldots, m$, which are only integrable. For this purpose, a generalized CBH formula (logarithm of the Chen-Fliess series) for nonstationary vector fields would have to be employed as mentioned in Remark 3.3. The product expansion (7) and the associated formula (14), would however still be valid.

4.2 Nonlinear filtering

Lie algebraic methods originally conceived as tools for the analysis of nonlinear systems have also found application in nonlinear filtering problems; the reader is referred to [23] for a complete expository review. In the nonlinear filtering problem the objective is to estimate the state of a stochastic process $x(t)$ which cannot be measured directly, but may be inferred from measurements of a related observation process $y(t)$.

Typical filtering problems consider the following signal observation model:

$$\begin{aligned} dx(t) &= f(x(t))dt + g(x(t))dv(t), \qquad x(0) = x_0 \\ dy(t) &= h(x(t))dt + dw(t), \qquad\qquad y(0) = 0 \end{aligned} \tag{20}$$

where x, v and y, w are $\mathbb{R}^n$ and $\mathbb{R}^m$ valued processes, respectively, and v and w have components which are independent, standard Brownian processes. Furthermore, f, h and g are assumed to be smooth functions.

Essential for the estimation of the state is the conditional probability density of the state, $\rho(t, x)$, given the observation $\{y(s); 0 \le s \le t\}$. It is well known, see [10], that $\rho(t, x)$ is obtained by normalizing a function $\sigma(t, x)$ which is the solution of the Duncan-Mortensen-Zakai (DMZ) bilinear, stochastic, partial differential equation:

$$d\sigma(t, x) = L_0\sigma(t, x)dt + \sum_{i=1}^{m} L_i\sigma(t, x) \circ dy_i(t), \qquad \sigma(0, x) = \sigma_0(x) \tag{21}$$

where $\circ dy(t)$ denotes the *Fisk-Stratonovitch differential* of the observation process $y(t)$, the differential operator L_0, given by:

$$L_0 = \frac{1}{2}\sum_{i=1}^{n} \frac{\partial^2}{\partial x_i^2} - \sum_{i=1}^{n} f_i \frac{\partial}{\partial x_i} - \sum_{i=1}^{n} \frac{\partial f_i}{\partial x_i} - \frac{1}{2}\sum_{i=1}^{m} h_i^2 \tag{22}$$

is defined on the space of smooth functions $\mathcal{D}(\mathbb{R}^n)$ on $\mathbb{R}^n$ with compact support, and where L_i is the operator of multiplication by h_i, $i = 1, \ldots, m$. Here, $\sigma_0 \in$

$M_+(\mathbb{R}^n)$ is the probability density of the initial point x_0, and $M_+(\mathbb{R}^n)$ denotes the space of nonnegative bounded measures on $\mathbb{R}^n$.

A particularly useful concept associated with the DMZ equation is the *estimation Lie algebra*, as introduced in [5], which is defined as the Lie algebra generated by the differential operators $L_0, \ldots, L_m$ (the Lie product of operators is calculated in a standard way, i.e. $[X, Y]\phi = X(Y\phi) - Y(X\phi)$, for any smooth function ϕ). The structure and dimensionality of the estimation Lie algebra is directly related to the existence of a finite dimensional recursive filter for the computation of $\rho(t, x)$, see [23]. It has been shown that if the estimation Lie algebra can be identified with a Weyl algebra of any order, then no non-constant statistics exist for the computation of the conditional density $\rho(t, x)$ with a finite dimensional filter; see references in [23]. In this context, the LTP package is helpful in the computation of the generators for the Weyl algebras as it permits to compute the Lie product in a coordinate independent fashion.

In the special case when the estimation Lie algebra is finite dimensional and solvable, (see [43] for the definition of solvability), the DMZ equation can be solved via an extension of the Wei-Norman formalism. Such a construction will be illustrated by an example employing the Lie tools package.

5 Using LTP: Some Practical Examples

The Maple code for the examples presented in this section is distributed with LTP, which may be obtained at: `http://www.cim.mcgill.ca/~migueltt/ltp/ltp.html`.

Any Lie product which is written in terms of the algebra generators only, will henceforth be referred to as a *Lie monomial*. By the property of distributivity over scalar multiplication, an *arbitrary* Lie bracket is a product of a symbolic coefficient and a Lie monomial.

The main and auxiliary functions provided by LTP are summarized in Table 1, p. 35, and Table 2, p. 36, respectively. Auxiliary functions are invoked by the main functions, but are also made directly available to the user to allow for perusal of intermediate results. Such an organization of the package facilitates the addition of new functions. See Section 6, p. 35, for details on the function's syntax, their algorithmic implementation, and other aspects.

Prior to invoking any function in the package, two special variables need to be declared, under arbitrary names, to signify: the number of generators in the Lie algebra $L(\bar{X}_m)$ and its assumed order of nilpotency. The values of these variables are limited only by the available computer memory.

The examples presented in the next two sections consider a set of Lie algebra generators $\bar{X}_3 = (X_1, X_2, X_3)$ and a HB, denoted by B, for a nilpotent Lie algebra $L_4(\bar{X}_3)$ with degree of nilpotency $k = 4$. The generators $\bar{X}_3$ and the basis B are easily obtained by executing the package function `phb(3,4)`; the resulting basis B is shown in § 5.2.

5.1 Example 1: Simplification of Lie algebraic expressions

To explain some of the capabilities of the package we consider a few examples.

To simplify the following expression $x := [\alpha X_3, [\alpha X_2, (\alpha+\beta^2)X_1]])$, in which α and β are considered to be symbolic scalars, the function `y:=simpLB(x)` is invoked and returns the result: $(\alpha^3 + \alpha^2\beta^2)[X_3, [X_2, X_1]]$, as well as, but separately, the scalar part of it, $(\alpha^3 + \alpha^2\beta^2)$, and the Lie monomial $[X_3, [X_2, X_1]]$. Such an answer form facilitates further calculations; for example when the expression needs to be rewritten in terms of elements of the basis B. The latter can be accomplished by subsequently invoking the function `phbize(y[3])`, which acts on the third argument of the result.

Another example, where skillful simplification is essential, is provided by the composition of exponential mappings $e^{Z_1}e^{Z_2} = e^{Z_3}$, with Z_1 and Z_2 declared as two simple Lie polynomials: $Z_1 = a_1X_1+a_2X_2+a_3X_3$, $Z_2 = b_1X_1+b_2X_2+b_3X_3$,

and with $a_i,\ b_i,\ i = 1,2,3$, declared as symbolic scalars. Employing the CBH formula in Dynkin's form, (3), a truncation of the series for Z_3 involving brackets up to order $n = 4$ is obtained by invoking first the function `cbhexp(`Z_1`,`Z_2`,n)`. This produces a complicated expression involving 231 Lie products of indeterminates, which are further simplified by executing the function `reduceLB(`Z_3`,`B`)`. This reduces Z_3 into its expression in the Hall basis B which, in this particular case, counts only 29 components. The first 12 terms of the final result are shown below:

$$
\begin{aligned}
Z_3 \quad \texttt{:=} \quad & (a_1+b_1)X_1 + (a_2+b_2)X_2 + (a_3+b_3)X_3 + \frac{1}{2}(a_1b_2 - a_2b_1)[X_1,X_2] \\
& + \frac{1}{2}(a_1b_3 - a_3b_1)[X_1,X_3] + \frac{1}{2}(a_2b_3 - a_3b_2)[X_2,X_3] \\
& + \frac{1}{12}(a_1^2b_2 - b_1a_1b_2 + b_1^2a_2 - a_1a_2b_1)[X_1,[X_1,X_2]] \\
& + \frac{1}{12}(b_1^2a_3 - a_1a_3b_1 - b_3a_1b_1 + a_1^2b_3)[X_1,[X_1,X_3]] \\
& + \frac{1}{12}(a_1a_2b_2 - b_2^2a_1 + b_1a_2b_2 - a_2^2b_1)[X_2,[X_1,X_2]] \\
& + \frac{1}{12}(b_2a_3b_1 - a_3a_2b_1 - b_3a_1b_2 + a_1a_2b_3)[X_2,[X_1,X_3]] \\
& + \frac{1}{12}(b_2^2a_3 - a_3a_2b_2 - b_2a_2b_3 + a_2^2b_3)[X_2,[X_2,X_3]] \\
& + \frac{1}{12}(a_1a_3b_2 - b_3a_1b_2 - a_3a_2b_1 + b_1a_2b_3)[X_3,[X_1,X_2]] + \ldots
\end{aligned}
$$

5.2 Example 1: Stabilization of a rigid body in space

The usefulness of LTP for practical applications in control of dynamical systems is illustrated by an example of an underactuated rigid body in space for which, after the application of a suitable feedback transformation, the model equations are:

$$\dot{x} = f_0(x) + f_1(x)u_1 + f_2(x)u_2 \tag{23}$$

$$
\begin{aligned}
\text{where,}\quad f_0(x) \quad = \quad & (\sin(x_3)\,\sec(x_2)\,x_5 + \cos(x_3)\,\sec(x_2)\,x_6)\,\frac{\partial}{\partial x_1} \\
& + (\cos(x_3)\,x_5 - \sin(x_3)\,x_6)\,\frac{\partial}{\partial x_2} \\
& + (x_4 + \sin(x_3)\,\tan(x_2)\,x_5 + \cos(x_3)\,\tan(x_2)\,x_6)\,\frac{\partial}{\partial x_3} + a\,x_4x_5\frac{\partial}{\partial x_6}, \\
f_1(x) \quad = \quad & \frac{\partial}{\partial x_4}, \quad f_2(x) = \frac{\partial}{\partial x_5}, \quad \text{and} \;\; \dot{x} = [\dot{x}_1,\ \dot{x}_2,\ \dot{x}_3,\ \dot{x}_4,\ \dot{x}_5,\ \dot{x}_6]^T
\end{aligned}
$$

Here u_1 and u_2 are the actuating controls, a is a scalar constant, f_0 is the drift vector field, and f_1 and f_2 are the input vector fields. For details on the model

derivation see, for example [8], and references therein.

The construction of stabilizing feedback control for systems such as (23) can be carried out employing different approaches; one such approach is proposed in [26]. As a point of further interest, it is worth pointing out that model (23) does not lend itself directly to the application of the method in [26] as its controllability Lie algebra is not nilpotent. Its application is made feasible only with the reference to a suitable nilpotent "approximation" of the original system; rigorous criteria for obtaining such approximations can be found in [15]. Here, it is demonstrated that even a trivial approximation amounting to a straightforward nilpotent truncation of the controllability Lie algebra for the original system is sufficient for stabilization. The truncation is merely required to preserve controllability of the system. This is justified by the result in [20, Thm. 2]), which shows that the steering error introduced while employing a truncated version of the controllability Lie algebra is a decreasing function of the distance between the initial and target points. It follows that the steering error can be controlled by selecting an adequately small time horizon T. Both the degree of nilpotency and the horizon T can be selected on a trial and error basis by requesting periodic decrease in a Lyapunov function which is a directly verifiable criterion for the adequacy of the truncation.

In this context, system (23) is assumed to be approximated by another system of a similar structure

$$\dot{x} = g_0(x) + g_1(x)u_1 + g_2(x)u_2 \tag{24}$$

whose controllability Lie algebra, $L(g_0, g_1, g_2)$, corresponds to a nilpotent truncation of $L(f_0, f_1, f_2)$ of some finite order. The order of truncation is selected so that the truncated system is STLC (see Theorem 7.3 in [42]). To follow this process, sufficiently many elements in the basis for $L(f_0, f_1, f_2)$ need to be known, and one way to proceed is to generate bases for $L_k(\bar{X}_3)$, $k = 2, 3, \ldots$ in ascending order, to select the smallest k for which the image of $L_k(\bar{X}_3)$ (under the evaluation map Ev) is STLC.

5.2.1 Step 1: Construction of the Hall basis for the Lie algebra of indeterminates $L_4(\bar{X}_3)$

For $k = 4$, a Hall basis for $L_4(\bar{X}_3)$ is first constructed by invoking B`:=phb(3,4)`, which yields B as a list of 32 elements, conveniently denoted by B_I, $I \in \mathcal{I}$, where the set of multi- $\mathcal{I} \overset{def}{=} \{I = (i_1, \ldots, i_4) : i_j \in \{1, 2, 3\}, 1 \leq j \leq 4\}$ contains all Hall words of length not exceeding four using the alphabet $\{1, 2, 3\}$; see Remark 3.1.

To evaluate the images $g_I \overset{def}{=} Ev(B_I)$, $I \in \mathcal{I}$, of the elements in the Hall basis

B the vector fields f_0, f_1, f_2 are declared as symbolic expressions in Maple, and the function `calcLB` is invoked, remembering that $f_{i-1} = Ev(B_i)$, $i = 1, 2, 3$.

The 29 brackets computed in this way are:

$$
\begin{aligned}
g_{01}(x) &= [f_0, f_1] = [0, 0, -1, 0, 0, -a\, x_5]^T \\
g_{02}(x) &= [f_0, f_2] = [-\sin(x_3)/\cos(x_2), -\cos(x_3), -\sin(x_3)\tan(x_2), 0, 0, -a\, x_4]^T
\end{aligned}
$$

$$
\begin{aligned}
g_{001}(x) &= [f_0, [f_0, f_1]] = \\
&= \begin{bmatrix} (\cos(x_3)\, x_5 - \sin(x_3)\, x_6 + \cos(x_3)\, a\, x_5)/\cos(x_2) \\ -\sin(x_3)\, x_5 - \cos(x_3)\, x_6 - \sin(x_3)\, a\, x_5 \\ \sin(x_2)\,(\cos(x_3)\, x_5 - \sin(x_3)\, x_6 + \cos(x_3)\, a\, x_5)/\cos(x_2) \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
g_{002}(x) &= [f_0, [f_0, f_2]] = \\
&= \begin{bmatrix} \cos(x_3)\, x_4/\cos(x_2)\,(-1+a) \\ -\sin(x_3)\, x_4\,(-1+a) \\ (\cos(x_2)\, x_6 - \cos(x_3)\,\sin(x_2)\, x_4 + \cos(x_3)\,\sin(x_2)\, a\, x_4)/\cos(x_2) \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
g_{102}(x) &= [f_1, [f_0, f_2]] = [0, 0, 0, 0, 0, -a]^T \\
g_{201}(x) &= [f_2, [f_0, f_1]] = g_{10}(x) \\
g_{0102}(x) &= [[f_0, f_1], [f_0, f_2]] = [\cos(x_3)/\cos(x_2), -\sin(x_3), \cos(x_3)\,\sin(x_2)/\cos(x_2), 0, 0, 0]^T \\
g_{0001}(x) &= [f_0, [f_0, [f_0, f_1]]] = \\
&= \begin{bmatrix} -x_4/\cos(x_2)\,(\sin(x_3)\, x_5 + \cos(x_3)\, x_6 + 2\,\sin(x_3)\, a\, x_5) \\ -x_4\,(\cos(x_3)\, x_5 - \sin(x_3)\, x_6 + 2\,\cos(x_3)\, a\, x_5) \\ \left\{ \begin{array}{c} (+\cos(x_2)\, x_5\, x_5 + \cos(x_2)\, a\, x_5\, x_5 - \sin(x_2)\,\sin(x_3)\, x_5\, x_4 \ldots \\ \ldots - \sin(x_2)\,\cos(x_3)\, x_6\, x_4 - 2\,\sin(x_3)\,\sin(x_2)\, a\, x_4\, x_5 \ldots \\ \ldots + \cos(x_2)\, x_6\, x_6)/\cos(x_2) \end{array} \right\} \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
g_{0002}(x) &= [f_0, [f_0, [f_0, f_2]]] = \\
&= \begin{bmatrix} -(-\sin(x_3)\, x_4\, x_4 + \sin(x_3)\, x_4\, x_4\, a + \cos(x_3)\, x_5\, x_6 - \sin(x_3)\, x_6\, x_6)/\cos(x_2) \\ \cos(x_3)\, x_4\, x_4 - \cos(x_3)\, x_4\, x_4\, a + \sin(x_3)\, x_5\, x_6 + \cos(x_3)\, x_6\, x_6 \\ \left\{ \begin{array}{c} (-\sin(x_2)\,\cos(x_3)\, x_5\, x_6 - \sin(x_2)\,\sin(x_3)\, x_4\, x_4\, a \ldots \\ \ldots + \sin(x_2)\,\sin(x_3)\, x_4\, x_4 - x_4\, x_5\,\cos(x_2) \ldots \\ \ldots + \sin(x_2)\,\sin(x_3)\, x_6\, x_6 + 2\,\cos(x_2)\, a\, x_4\, x_5)/\cos(x_2) \end{array} \right\} \\ 0 \\ 0 \\ 0 \end{bmatrix} \\
g_{1002}(x) &= [f_1, [f_0, [f_0, f_2]]] = (-1+a) * g_{15}(x) \\
g_{2001}(x) &= [f_2, [f_0, [f_0, f_1]]] = (1+a) * g_{15}(x)
\end{aligned}
$$

and $g_I = [0, 0, 0, 0, 0, 0]^T$, for $I =$ 12, 101, 112, 202, 212, 0112, 0212, 1001, 1101,

1102, 1112, 2002, 2101, 2102, 2112, 2201, 2202, 2212.

The desired nilpotent truncation of $L(f_0, f_1, f_2)$ (valid in the neighborhood of the origin only) can thus be obtained by assuming that

$$g_{001} = g_{002} = g_{0001} = g_{0002} = [0, 0, 0, 0, 0, 0]^T, \tag{25}$$

as indeed, the values of these brackets evaluated in the neighborhood of the origin are negligibly small. Considering the latter, together with the Lie products which are zero, and the following dependencies among the above Lie products:

$$g_{201} = g_{102}, \qquad g_{1002} = (-1 + a)g_{0102}, \qquad g_{2001} = (1 + a)g_{0102}, \tag{26}$$

which correspond (via evaluation map Ev) to the following symbolic dependencies between the elements of B:

$$B_I = 0, \quad B_{201} = B_{102}, \quad B_{1002} = (-1 + a)B_{0102}, \quad B_{2001} = (1 + a)B_{0102}, \tag{27}$$

for I = 12, 001, 002, 101, 112, 202, 212, 0112, 0212, 0001, 0002, 1001, 1101, 1102, 1112, 2002, 2101, 2102, 2112, 2201, 2202, 2212, a basis for the controllability Lie algebra for system (24) can thus be defined as:

$$B_g \overset{def}{=} \{g_0, g_1, g_2, g_{01}, g_{02}, g_{102}, g_{0102}\}$$

It can be verified that $L(g_0, g_1, g_2)$ is indeed nilpotent if (25) is enforced, and that such nilpotent $L(g_0, g_1, g_2)$ corresponds to an STLC system as required.

Additionally, the ordering of this basis satisfies the condition (13), which guarantees that the Wei-Norman equation can be given in the explicit form (14).

The feedback design approach developed in [26] now calls for the computation of an open-loop piece-wise constant control $\bar{u} : [0, T] \to \mathbb{R}^3$ such that the γ-coordinates for system (24) satisfy

$$\gamma(T, \bar{u}) \in \mathcal{R}(T, U^e(p)) \tag{28}$$

where $\gamma(T, \bar{u})$ is the value of the γ-coordinates at time T and corresponding to the control $\bar{u}$. The set $U^e(p)$ is a set of admissible "extended controls" which provide for a monotonic decrease of a given Lyapunov function along the trajectories (originating at a given point p) of the extended system to (24) defined as:

$$\begin{aligned} \dot{x} &= g_0(x) + g_1(x)v_1 + g_2(x)v_2 + g_{01}(x)v_3 + g_{02}(x)v_4 \\ &\quad + g_{102}(x)v_5 + g_{0102}(x)v_6 \end{aligned} \qquad x(0) = p$$

The set $\mathcal{R}(T, U^e(x))$ is the reachable set for system (29) in the γ-coordinates space while employing controls from $U^e(p)$.

In this context, the idea behind the feedback stabilization algorithm is the following. As has been pointed out in [26], for each $\bar{u}$ satisfying (28) there exists

an extended control $v \in U^e(p)$ such that the γ-coordinates of (24) and (29) match at time T; i.e. $\gamma(T, \bar{u}) = \gamma^e(T, v)$, where γ^e are the γ-coordinates of the flow of the extended system (29). This fact immediately implies that the chosen Lyapunov function decreases (periodically) along the trajectories of the original system (23) (for a precise meaning of "periodical decrease" see [26]). To this end, the method in [26] requires the construction of the Wei-Norman equations for systems (24) and (29) where the LTP package yet again comes useful.

5.2.2 Step 2: Calculation of the right-hand side of the Wei-Norman equation

The derivation of the Wei-Norman equation is carried out in two steps. The product term in the right-hand side of (10) is first computed by invoking the LTP function `wner`, in which the basis elements B_I need to be replaced by g_I, $I = 0, 1, 2, 01, 02, 102, 0102$. Next, the coefficients corresponding to the basis elements g_I, on both sides of equation (10)–(3) are equated using the LTP function `wnde`.

More precisely, the LTP function `wner` ought to be invoked with the following parameters: `rhwne:=wner(`r`,`$k-1$`,`B`, `B_g`, lbdt)`, where $r = 7$ is the dimension of the basis B_g, $k = 4$ is the degree of nilpotency, and `lbdt` is the list of linear dependencies (27). The resulting expression is:

$$\begin{aligned} rhwne \;:=\; & \dot{\gamma}_0 f_0 + \dot{\gamma}_1 f_1 + \dot{\gamma}_2 f_2 + (\dot{\gamma}_1\gamma_0 + \dot{\gamma}_3)[f_0, f_1] + (\dot{\gamma}_2\gamma_0 + \dot{\gamma}_4)[f_0, f_2] \\ & + (\dot{\gamma}_3\gamma_2 + \dot{\gamma}_4\gamma_1 + \dot{\gamma}_5)[f_1, [f_0, f_2]] + (\dot{\gamma}_4\gamma_3 + \dot{\gamma}_5\gamma_0 a + \dot{\gamma}_6)[[f_0, f_1], [f_0, f_2]] \end{aligned}$$

The function `wnde(rhwne,`r`,`B`,lbdt)` is applied to the above result returning the matrix $\xi(\gamma)$ (see equation (12)) and the set of equations:

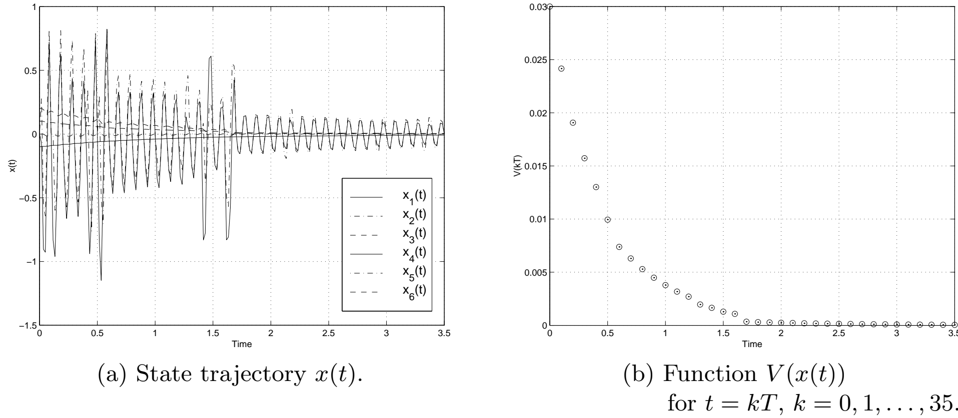
$$\begin{aligned} v_0 &= \dot{\gamma}_0 \\ v_1 &= \dot{\gamma}_1 \\ v_2 &= \dot{\gamma}_2 \\ v_3 &= \dot{\gamma}_1\gamma_0 + \dot{\gamma}_3 \\ v_4 &= \dot{\gamma}_2\gamma_0 + \dot{\gamma}_4 \\ v_5 &= \dot{\gamma}_3\gamma_2 + \dot{\gamma}_4\gamma_1 + \dot{\gamma}_5 \\ v_6 &= \dot{\gamma}_4\gamma_3 + \dot{\gamma}_5\gamma_0 a + \dot{\gamma}_6 \end{aligned}$$

The inversion of $\xi(\gamma)$ results in the following Wei-Norman equation:

$$\begin{bmatrix} \dot\gamma_0 \\ \dot\gamma_1 \\ \dot\gamma_2 \\ \dot\gamma_3 \\ \dot\gamma_4 \\ \dot\gamma_5 \\ \dot\gamma_6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -\gamma_0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -\gamma_0 & 0 & 1 & 0 & 0 \\ 0 & \gamma_0\gamma_2 & \gamma_0\gamma_1 & -\gamma_2 & -\gamma_1 & 1 & 0 \\ 0 & -a\gamma_0^2\gamma_2 & \gamma_0\gamma_3 - a\gamma_0^2\gamma_1 & a\gamma_0\gamma_2 & a\gamma_0\gamma_1 - \gamma_3 & -a\gamma_0 & 1 \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \\ v_6 \end{bmatrix}$$

with $\gamma_i(0) = 0$, $i = 0, 1, \ldots, 6$.

A feasible control $\bar u$ satisfying the inclusion (28) is found as follows. First, (29) is integrated symbolically over $[0, T]$ and solved with respect to the extended controls v_i, $i = 0, \ldots, 6$ evaluated at T to yield a symbolic expression for the reachable set $\mathcal{R}(T, U^e(p))$, now given as a set of admissible coordinate values $\gamma(T, \bar u)$ for the original system. Next, a control $\bar u$ is found by solving (28) using standard nonlinear programming techniques; see [26] for details of this calculation. Stabilization is achieved by repetitive solution of (28) as shown by simulation results in Figure 1 which correspond to an initial condition $x_0 = [-0.1\ 0\ 0.2\ 0\ 0\ 0.1]^T$. These results were obtained using a quadratic Lyapunov function $V(x) = \frac{1}{2}\|x\|^2$ and piece-wise constant controls $\bar u$ consisting of at most five switching times in any interval of length $T = 0.1$.



(a) State trajectory $x(t)$.

(b) Function $V(x(t))$ for $t = kT$, $k = 0, 1, \ldots, 35$.

Figure 1: Results for the stabilization of the rigid body.

5.3 Example 2: Finite dimensional realization of a non-linear filter

The aim here is to construct a finite dimensional realization of a nonlinear filter for the stochastic system described by (see [28, 19]):

$$\begin{aligned} dx(t) &= dv(t), \quad x(0) = x_0 \\ dy(t) &= x(t)dt + dw(t) \end{aligned} \tag{29}$$

where v and w are independent Brownian motions. As suggested in [5] such a realization can be derived by applying Lie algebra techniques to the DMZ equation for the unnormalized conditional density $\sigma(t,x)$, given the observation process $\{y(s); 0 \le s \le t\}$ for system (29). The DMZ equation here is

$$d\sigma(t,x) = L_0\sigma(t,x) + L_1\sigma(t,x) \circ dy(t),\ \sigma(0,x) = \sigma_0(x),\ \sigma_0 \in \mathbb{L}_2(\mathbb{R})$$

where the differential operators $L_0, L_1 : D(\mathbb{R}) \to \mathbb{L}_2(\mathbb{R})$ are defined by the following expressions on their common invariant domain $D(\mathbb{R})$ which is dense in $\mathbb{L}_2(\mathbb{R})$ (see [28]):

$$L_0 = \frac{1}{2}\frac{\partial^2}{\partial x^2} - \frac{x^2}{2}, \qquad L_1 = x$$

It will first be shown that the estimation Lie algebra $L_E \stackrel{def}{=} L(L_0, L_1)$ for the above problem is finite dimensional and solvable. Then, the solution of the Cauchy problem for any given $\sigma_0 \in \mathbb{L}_2(\mathbb{R})$, representing the conditional density of $x(0)$, can be written in the form of a product of exponentials, see [23]:

$$\sigma(t,x) = \prod_{i=0}^{r} e^{\gamma_i(t)L_i}\sigma_0(x) \tag{30}$$

where L_i, $i = 0,\ldots,r$ is a basis for the Lie algebra L_E. The exponential e^{tL_i} represents here a strongly continuous one-parameter semi-group operator defined on a Banach space $\mathbb{L}_2(\mathbb{R})$ and corresponding to the infinitesimal generator L_i. The last representation is only valid if the Baker-Campbell-Hausdorff-Zassenhaus formula:

$$e^{tL_i}L_j = \left(\sum_{m=0}^{\infty}\frac{t^m}{m!}(adL_i)^m L_j\right)e^{tL_i} \tag{31}$$

holds for all the L_i, L_j, $i,j = 0,\ldots,r$. As pointed out in [28] the validity of (31) is guaranteed if there exists a common, dense (in $\mathbb{L}_2(\mathbb{R})$), invariant under L_E, set of analytic vectors for the estimation Lie algebra spanned by L_i, $i = 0,\ldots,r$. Such a set can be constructed as the linear span of eigenvectors of the operator L_0.

To check the solvability of L_E, the differential operators L_0 and L_1 are first defined in Maple as follows:

```
> L0:=xi->(1/2)*diff(xi,x$2)-(1/2)*x^2*xi;
  L1:=xi->x*xi;
```

$$L0 \;:=\; \xi \to \frac{1}{2}\frac{\partial^2 \xi}{\partial x^2} - \frac{x^2}{2}\,\xi$$
$$L1 \;:=\; \xi \to x\,\xi$$

A basis for the Lie algebra of operators $L(L_0, L_1)$ is obtained by considering a free nilpotent Lie algebra $L_n(X_0, X_1)$, where n is a sufficiently high order and calculating its P. Hall basis. For example, for $n = 7$, the P. Hall basis for $L_7(X_0, X_1)$, $B = \{B_1, \ldots, B_{41}\}$ counts 41 elements and is constructed by invoking `B:=phb(2,7)`. In this process, the package also delivers explicit bracket expressions for the basis elements in B which are omitted here for reason of brevity. Identifying $L_{i-1} = Ev(B_i)$, $i = 1, 2$, the basis elements in B can thus be evaluated next by executing the LTP function `calcLBdiffop(B[i],B[1..2],[L0,L1],[x])`, for `i`$= 3, 4, \ldots, 41$, yielding:

$$L_2 \stackrel{def}{=} Ev(B_3) = [L_0, L_1] = \frac{\partial}{\partial x}$$
$$L_3 \stackrel{def}{=} Ev(B_4) = [L_0, [L_0, L_1]] = x = Ev(B_2) = L_1$$
$$L_4 \stackrel{def}{=} Ev(B_5) = [L_1, [L_0, L_1]] = -1$$

It can further be verified that the application of the evaluation map Ev to the remaining brackets in the basis `B` reveals several linear dependencies between $L_{i-1} = Ev(B_i)$, $i = 1, \ldots, 41$: $L_5 = L_{26} = L_2$, $L_{11} = L_{35} = L_1$, $L_8 = -L_{12} = -L_{20} = L_{31} = -L_{36} = -L_4$, and $L_i = 0$, for the remaining Lie products. From this calculation it follows that a basis for $L(L_0, L_1)$ can be defined as $\{L_0, L_1, L_2, L_4\} \stackrel{def}{=} \{L_0, L_1, [L_0, L_1], [L_1, [L_0, L_1]]\}$. These calculations also show that the derived Lie algebra $[L_E, L_E]$ is spanned by L_2 and L_4, and is nilpotent because $[L_2, L_4] = Ev(B_{10}) = 0$. Hence, the Lie algebra L_E is solvable, by Corollary 5.3 in [36].

The representation (30) now becomes:

$$\sigma(t, \cdot) = e^{\gamma_0(t) L_0} e^{\gamma_1(t) x} e^{\gamma_2(t) \frac{\partial}{\partial x}} e^{-\gamma_3(t)} \sigma_0 \tag{32}$$

is hence valid globally, see [45], and the functions γ_i, $i = 0, \ldots, 3$ can be computed by quadrature of the Wei-Norman equations. The analytic expression for the Wei-Norman equations can be derived by executing the sequence of commands:

```
r:=4;                   # Basis dimension.
max_bracket_order:=6; # Degree of nilpotency minus one.
```

```
wn:=wner(r,max_bracke_order,BB,B,[SR]):
wnfe:=wnde(wn,r,{},BB,{}):
F_g:=eval(wnfe[1]):
```

The symbol [SR] is a Maple list containing the dependencies between the members of the basis B after application of the evaluation map Ev as derived above. The symbol F_g assumes value of the matrix $\xi(\gamma)$ of equation (12) and is here:

$$F_g := \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1+\frac{1}{2}\gamma_0^2+\frac{1}{24}\gamma_0^4+\frac{1}{720}\gamma_0^6 & \gamma_0+\frac{1}{6}\gamma_0^3+\frac{1}{120}\gamma_0^5 & 0 \\ 0 & \gamma_0+\frac{1}{6}\gamma_0^3+\frac{1}{120}\gamma_0^5 & 1+\frac{1}{2}\gamma_0^2+\frac{1}{24}\gamma_0^4+\frac{1}{720}\gamma_0^6 & 0 \\ 0 & 0 & \gamma_1 & 1 \end{bmatrix}$$

The entries $(2,2)$ and $(3,3)$ of F_g can be clearly be recognized as the first few terms in the Taylor series expansion for cosh. Similarly, the entries $(2,3)$ and $(3,2)$ are recognized as the first few terms of the Taylor series for sinh. Now, it can be verified that if the above calculations are repeated using a Hall basis of order $n > 7$, then the entries of F_g will contain higher order terms of these Taylor series. Thus, by induction, it can be shown that these entries truly are the cosh and sinh functions, so that the Wei-Norman equations for (32) in the form (14) are given by:

$$\dot{\gamma}_0 = 1,\ \ \dot{\gamma}_1 = \cosh(\gamma_0)dy(t),\ \ \dot{\gamma}_2 = -\sinh(\gamma_0)dy(t),\ \ \dot{\gamma}_3 = \sinh(\gamma_0)\gamma_1 dy(t) = -\dot{\gamma}_2\gamma_1$$

where $u = [1\ dy(t)\ 0\ 0]^T$.

The solution of these Wei-Norman equations constitutes the *joint-sufficient statistics for the linear filtering problem* of (29). Now, Mehler's formula (see [28]) allows to obtain the explicit expression for the one parameter semi-group $e^{\gamma_0(t)L_0}$ in the form of an integral operator as follows:

$$e^{\gamma_0(t)L_0}\phi(x) = e^{tL_0}\phi(x) = \int_{-\infty}^{\infty} \frac{1}{2\pi\sinh(t)} e^{-1/2\coth(x^2+y^2)} e^{\frac{xy}{\sinh(t)}}\phi(y)dy \quad (33)$$

for any $\phi \in \mathcal{D}(\mathbb{R})$. Since $(e^{tx}\phi)(x) = e^{tx}\phi(x)$ and $(e^{t\frac{\partial}{\partial x}}\phi)(x) = \phi(x+t)$, then, finally, (32) and (33) combine into:

$$\sigma(t,x) = \int_{-\infty}^{\infty} \frac{1}{2\pi\sinh(t)} e^{-1/2\coth(x^2+y^2)} e^{\frac{xy}{\sinh(t)}} e^{\gamma_3(t)} e^{\gamma_1(t)y}\sigma_0(\gamma_2(t)+y)dy$$

which is an explicit formula for the nonlinear filter for (29).

6 Function Reference

This section contains the description of the LTP functions. The functions and their purpose are summarized in the table below.

Acronyms

CBH	- Campbell-Baker-Hausdorff formula
LB	- Lie bracket or product
LTP	- Lie Tools Package
PHB	- Philip Hall basis

Table 1: Main functions in LTP.

Function	Purpose
cbhexp	Calculates the exponent $Z_3 \in \hat{L}(\bar{X}_m)$ resulting from the composition of exponential mappings in equation (2) via the CBH formula (including brackets up to a given order k).
createLBobjects	Declares the generators $\bar{X}_m$ of the Lie algebra $L_k(\bar{X}_m)$. If needed, it also permits to declare any number of linear combinations of these generators $\sum_{i=1}^{m} a_i X_i$ with symbolic coefficients a_i. The LTP assigns a name to each linear combination allowing it to be used by other LTP functions.
phb	Declares the generators $\bar{X}_m$ of the free nilpotent Lie algebra $L_k(\bar{X}_m)$ of degree k and constructs a Hall basis for $L_k(\bar{X}_m)$.
phbize	Expresses any Lie monomial $X \in L_k(\bar{X}_m)$ in the Hall basis.
reduceLB	Reduces a general Lie polynomial $S \in L_k(\bar{X}_m)$ with symbolic coefficients to its simplest form in a given HB.
reduceLBT	Given a list of dependencies between the elements of the HB, reduces a general Lie polynomial $S \in L_k(\bar{X}_m)$ with symbolic coefficients to its simplest form.
regroupLB	Applies the distributivity properties (over addition and scalar multiplication) of the Lie product to an arbitrary Lie polynomial in $S \in L(\bar{X}_m)$ and collects its terms.
simpLB	Applies the distributivity over scalar multiplication property to a given Lie product $X \in L(\bar{X}_m)$ and returns the simplified product $\alpha Y = X$, together with its scalar symbolic component α, and the Lie monomial $Y \in L(\bar{X}_m)$.
wner	Computes the right-hand side of equation (10) and expresses it in the HB, treating $\dot{\gamma}_i$ and γ_i, $i = 1, \ldots, r$ as symbolic scalars.
wnde	Constructs the differential equation for the logarithmic coordinates γ_i given by the Wei-Norman equation (12).

Table 2: Auxiliary functions in LTP.

Function	Purpose
ad	Calculates $(ad_X^n)Y$ for $X, Y \in L(\bar{X}_m)$.
bracketlen	Returns the length $l(G)$ of a Lie product $G \in L(\bar{X}_m)$.
calcLB	Given the symbolic expressions for two vector fields in the canonical coordinate system, calculates their Lie product.
calcLBdiffop	Given the symbolic expressions for two partial differential operators, calculates their Lie product.
codeCBHcf	Generates code in either Fortran or C for the evaluation of the scalar symbolic coefficients in a given Lie polynomial $S \in L(\bar{X}_m)$.
createSubsRel	Creates Maple substitution relations for the the symbolic evaluation of controls u_i, $i = 0, \ldots, m$ in the dynamic system (15). These substitution relations can then be used to permit calculations involving systems with drift and to accommodate for piece-wise constant controls of arbitrary symbolic magnitude, as well as to allow the controls to switch at arbitrary symbolic moments in time.
ead	Computes the series expansion of $(e^X)Y(e^{-X}) = (e^{ad_X})Y$. for $X, Y \in L(\bar{X}_m)$ including brackets up to a given order.
eadr	Computes the series expansion of $(e^X)Y(e^{-X}) = (e^{ad_X})Y$. for $X, Y \in L_k(\bar{X}_m)$; re-expresses the result in the HB and further simplifies it according to a given list of dependencies involving the elements of the HB.
evalLB2expr	Returns a symbolic Maple expression for later evaluation of a Lie product of two vector fields, possibly containing symbolic scalars.
pead	Computes the product of exponentials $\prod_{i=1}^{n} e^{ad_{X_i}} X_{n+1}$ for $X, Y \in L(\bar{X}_m)$ including brackets up to a given order.
peadr	Computes the product of exponentials $\prod_{i=1}^{n} e^{ad_{X_i}} X_{n+1}$ for $X, Y \in L_k(\bar{X}_m)$; re-expresses the result in the HB and further simplifies it according to a given list of dependencies involving the elements of the HB.
posxinphb	Returns the position index i of a Lie product B_i in the HB.
selectLB	Extract, as a Maple symbolic expression for later use, the part of a given Lie polynomial $S \in L(\bar{X}_m)$ which contains brackets up to, greater than, or equal to a given order.

6.1 createLBobjects

Purpose Create (declare) Lie algebra generators and control inputs.

Syntax createLBobjects(nGen,sLen);

Description This function creates the symbolic variables representing the Lie algebra generators and the control inputs. The function assumes the variables do not exist, so if the variables exists, they will be replaced by the new ones, with the corresponding assumptions without warning to the user. Adding a line to check for the existence of already assigned variables is simple and can be done as for the function `phb`. In general, however, verifying variables redefinition should not be really necessary since the amount of variables regarding a particular problem or system is directly related to the system model, which should only be changed at the time of declaring the system generators and inputs and not at some intermediate step of the symbolic manipulations.

Arguments

nGen	Number of Lie algebra generators.
sLen	Length of the sequence of inputs, i.e. the number of switchings in the input sequence.

Examples Declaration of vector fields for a system with drift, 2 inputs and a sequence of control inputs of length 4 (i.e. four switches).

```
> createLBobjects(3,4);
Generators     Input Sequences
 f0~            u0_1~ u0_2~ u0_3~ u0_4~
 f1~            u1_1~ u1_2~ u1_3~ u1_4~
 f2~            u2_1~ u2_2~ u2_3~ u2_4~
Span of Generators for each segment of the control sequence
 f_1:=f0*u0_1+f1*u1_1+f2*u2_1
 f_2:=f0*u0_2+f1*u1_2+f2*u2_2
 f_3:=f0*u0_3+f1*u1_3+f2*u2_3
 f_4:=f0*u0_4+f1*u1_4+f2*u2_4
```

Discussion In the above example, `f0, f1, f2` represent the vector fields which generate the Lie algebra. The tilde ~ symbol at the end of each variable indicates that some assumptions have been made on these variables. The "subindex" `_i`, after each variable *var* indicates the corresponding time interval, thus the expression *var*`_i` corresponds to the particular value of *var* in the time interval `i` of the input sequence. An arbitrary input sequence with 4 switches is illustrated in Fig. 6.1.

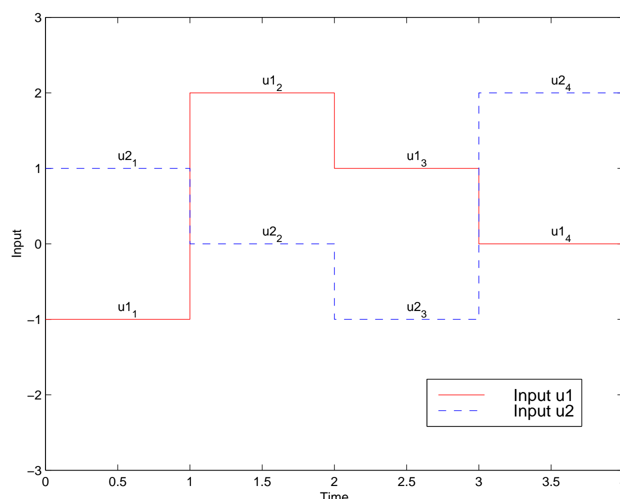


Figure 2: Control inputs sequence.

Notes Notice that the the system in the example is drift-free, however it is possible to obtain a system with drift by simply setting the controls `u0` to 1. This can be easily achieved with the Maple command `subs` for substituting expressions.

Limitations The current implementation does not allow to select a name for the generators or controls which are set to `f` and `u`, respectively.

Bugs In the current implementation the controls `u` are assumed to be of type real, even if they could be in any other field from the theoretical point of view. The reason for this is an apparent bug in Maple which returns true for the command `is(x,scalar)` for any `x`, even if `x` is assumed to be of type vector! Thus assuming `x` of type scalar causes a problem, since a generator of vector type will also be regarded by Maple as scalar and the simplification routines will fail to recognize it as a generator. To circumvent this problem, for the moment we use type real when we mean scalar, and type vector for the generators.

6.2 phb

Purpose Generate a Philip Hall basis.

Syntax B:=phb(m,k);

Description This function constructs a list containing the Philip Hall basis (PHB) for a nilpotent Lie algebra of degree k generated by m generators. The elements in the PHB are elements of the Lie algebra selected in way such that the dependencies between brackets, imposed by the anti-symmetry property and the Jacobi identity, are taken into account.

Arguments

m Number of Lie algebra generators.

k Order of nilpotency, i.e. brackets of length k+1 and higher are equal to zero.

Examples Construct a Philip Hall basis for a nilpotent algebra of order 4 generated by 3 vector fields.

```
> B:=phb(3,4);
B:=[f0~, f1~, f2~, f0~ &* f1~, f0~ &* f2~, f1~ &* f2~,
    f0~ &* (f0~ &* f1~), f0~ &* (f0~ &* f2~), f1~ &* (f0~ &* f1~),
    f1~ &* (f0~ &* f2~), f1~ &* (f1~ &* f2~), f2~ &* (f0~ &* f1~),
    f2~ &* (f0~ &* f2~), f2~ &* (f1~ &* f2~),
    (f0~ &* f1~) &* (f0~ &* f2~), (f0~ &* f1~) &* (f1~ &* f2~),
    (f0~ &* f2~) &* (f1~ &* f2~), f0~ &* (f0~ &* (f0~ &* f1~)),
    f0~ &* (f0~ &* (f0~ &* f2~)), f1~ &* (f0~ &* (f0~ &* f1~)),
    f1~ &* (f0~ &* (f0~ &* f2~)), f1~ &* (f1~ &* (f0~ &* f1~)),
    f1~ &* (f1~ &* (f0~ &* f2~)), f1~ &* (f1~ &* (f1~ &* f2~)),
    f2~ &* (f0~ &* (f0~ &* f1~)), f2~ &* (f0~ &* (f0~ &* f2~)),
    f2~ &* (f1~ &* (f0~ &* f1~)), f2~ &* (f1~ &* (f0~ &* f2~)),
    f2~ &* (f1~ &* (f1~ &* f2~)), f2~ &* (f2~ &* (f0~ &* f1~)),
    f2~ &* (f2~ &* (f0~ &* f2~)), f2~ &* (f2~ &* (f1~ &* f2~))]
```

Notes This function also declares the symbol for the Lie product operator denoted by `&*` if it was not previously assigned. This is only to ensure that `&*` and its properties (see Loading LTP in section 2.3 have been assigned in case it was manually removed. The `&*` operator is created by default at startup when the package is loaded.

Limitations There aren't any known limitations, besides the normal limitations imposed by the memory of the machine.

See Also `phbize`.

Algorithm

The Philip Hall Basis

Due to the antisymmetry property and the Jacobi identity not all the elements of the Lie algebra $\mathcal{L}(g_1, \ldots, g_m)$ generated by $g_1, \ldots, g_m$ are linearly independent. One possible method for constructing a basis which takes into account the dependencies imposed by the mentioned properties is to list all the generators $g_1, \ldots, g_m$ and select some of their Lie products according to the Philip Hall procedure described next [32].

Denote by B the basis, and let B_i be the *i-th* element in the basis. Define the length $l(G)$ of a Lie product G as the number of generators in the expansion for G or, alternatively in a recursive way:

$$
\begin{aligned}
l(g_i) &= 1 \qquad\qquad i = 1, \ldots, m \qquad (34)\\
l([G, H]) &= l(G) + l(H) \qquad (35)
\end{aligned}
$$

where G and H are Lie products.

Then a *Philip Hall basis* is an ordered set of Lie products $\{B_i\}$ satisfying:

1. $g_i \in B,\ i = 1, \ldots, m$
2. If $l(B_i) < l(B_j)$ then $B_i < B_j$
3. $[B_i, B_j] \in B$ if and only if
   (a) $B_i, B_j \in B$ and $B_i < B_j$ and
   (b) either $B_j = g_k$ for some k or $B_j = [B_p, B_q]$ with $B_p, B_q \in B$ and $B_p \leq B_i$.

For proofs that a Philip Hall basis is indeed a basis for the Lie algebra generated by $g_1, \ldots, g_m$ the reader is referred to:

J-P. Serre. Lie Algebras and Lie groups. W. A. Benjamin, New York, 1965.

M. Hall. The Theory of Groups. Macmillan, 1959.

A Philip Hall basis which is nilpotent of order k can also be constructed from the above definition by simply constructing all the Lie products that satisfy the properties in the above definition and have length not greater than k.

Implementation Notes

The implementation of the algorithm is illustrated in the flow chart of Figs. 4-5. For further details and remarks on the implementation the reader is referred to the source code.

From a practical perspective, the basis `B` can be built in such a way that only condition 3 needs to be checked, since condition 1 must be assumed true for all *initial* generators and 2 may be satisfied by performing the multiplications in an orderly manner, as briefly described next.

Condition 3 is implemented within the dashed block labeled "Create bracket $[B_i, B_j]$", shown in Fig. 5.

The bracketing procedure (i.e. the procedure for generating new Lie products or brackets) can be though of as a breeding process. We must distinguish between to groups per "breeding season" (iteration), the offspring and the parents.

On the first iteration the generators are treated as offspring and are crossed between them. On the second iteration, the offspring are called parents (since they will be crossed they will become parents), and their offspring are the new offspring. Parents are crossed only with their offspring and not between them, since this happened in the previous iteration. While offspring are crossed between them and also their parents to cover all possible combinations. All the newborns are now called offspring and the ones that were offspring are now in the group of parents. And life goes on...

Offspring are "cross-fertilized" as shown in Fig. 6.2.

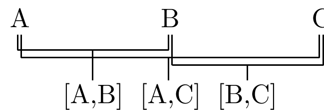


Figure 3: Lie bracketing tree.

Note that the [B,C], [C,A] and [B,A] are not valid offspring since they violate condition 2, assuming a lexicographical order is followed, i.e. A, B, C, are respectively the first, second and third generators.

Now there are two groups: parents A, B, C and offspring [A,B], [A,C], [B,C], denoted AB, AC and BC for short. Offspring will reproduce again as graphically described in the above figure, but also they will be crossed with their parents. All parents are crossed with all offspring, in the following way:

```
AxAB AxAC AxBC
BxAB BxAC BxBC
CxAB CxAC CxBC
```

Where '`x`' stands for *crossed with*, i.e. represents the Lie product operator. Note that some of the crossings must be eliminated by rule 3, namely `AxBC`.
In the flow chart of Figs. 4-5, b denotes the bracketing iteration (breeding season), g_old is the number of brackets that have been multiplied (crossed) at least once, g_tot is the number the total number of brackets including parents and offspring updated at the *end* of the iteration. Note that g_tot is note incremented as the breeding occurs since its value is necessary to close loop L3, in Fig. 5, to keep track of the new brackets the variable g_acum is used, instead, and its value will be passed to g_tot once the loop L3 is completed. The index i points to each element in the initial population, and is associated with the first term in the bracket $[B_i, B_j]$, while the index j associated with the second term in the bracket is set to start at the value of j_min according to whether the the crossings will be done between the offspring only ($j_min = i + 1$) or between the parents and the offspring ($j_min = g_old + 1$).

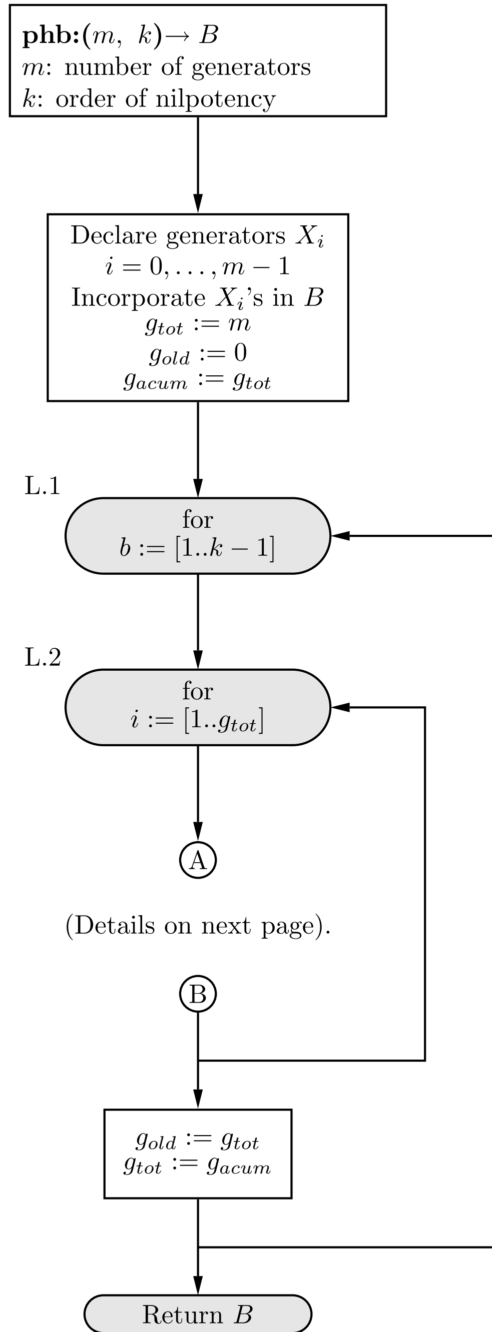


Figure 4: Flow chart for the `phb` algorithm (contd. on Fig. 5).

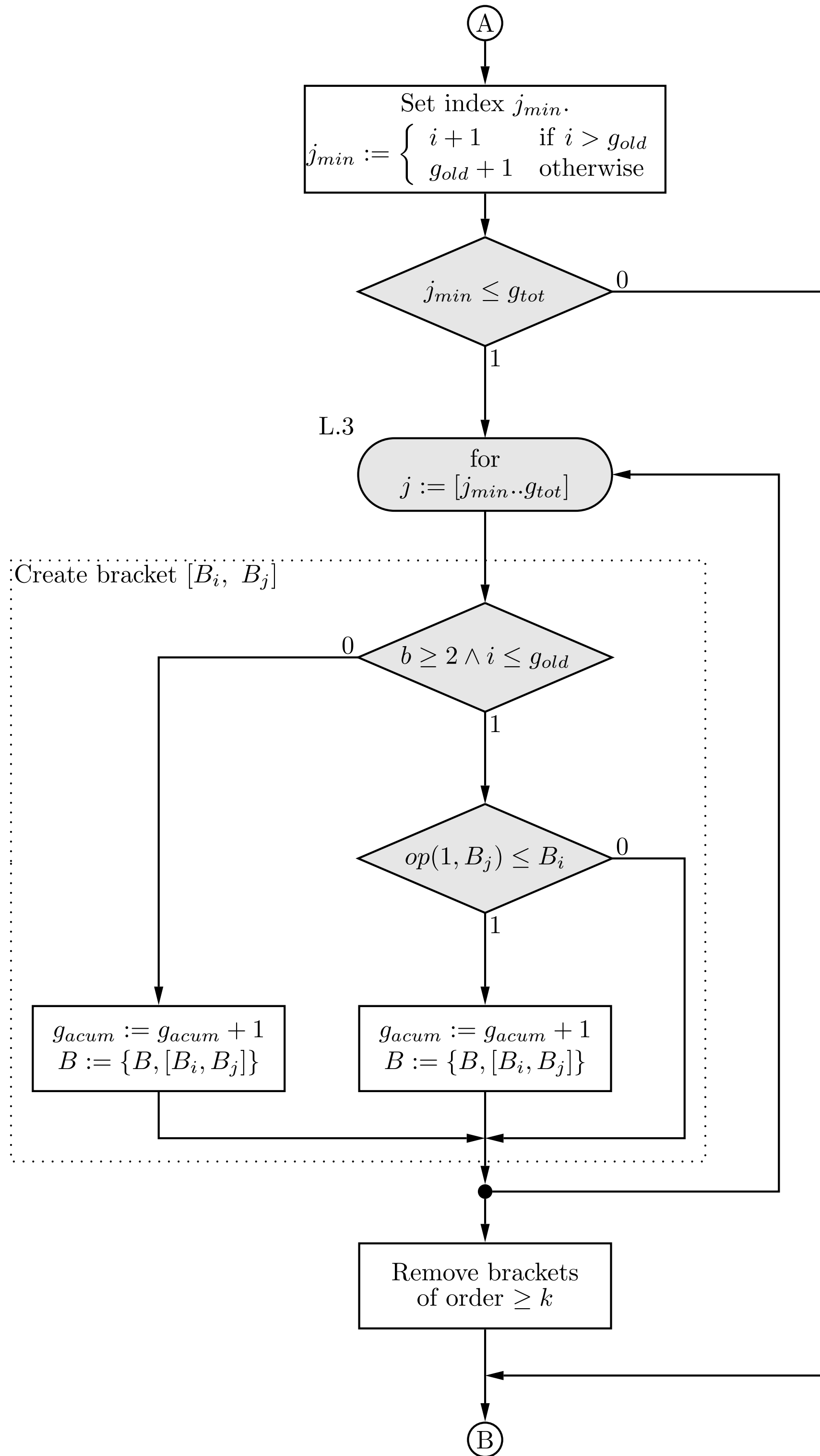


Figure 5: Flow chart for the `phb` algorithm (contd. from Fig. 4).

6.3 phbize

Purpose Transform any Lie bracket to a bracket or linear combination of brackets in the PHB. If the input Lie bracket is already an element of the PHB, then it is returned unchanged.

Syntax xr:=phbize(x,B);

Description The function `phbize`(x, B) transforms the Lie bracket x to a corresponding bracket or linear combination of brackets in the PHB, if x is not already in the PHB which must be passed in the second argument B. It is assumed that x is a pure bracket , i.e. the scalar parts have been removed (see the function `simpLB`, which allows to obtain the pure bracket as explained there).

NOTE: This function accepts arguments x of order higher than the degree of nilpotency of the Lie algebra, and which might exceed the degree of the highest order brackets in the basis. If the order of the term x exceeds the degree of nilpotency, but it is composed by the product of two elements in the PHB, the result returned will be correct. However, for more complicated brackets the result will be a partial simplification, not necessarily and element of the PHB, and thus the result must be interpreted with care.

Arguments

x	Some *pure* Lie bracket.
B	Philip Hall basis

Examples The following examples consider the PHB B shown in the example for the command phb on page 40.

```
> phbize(f0&*(f1&*f0),B);
-(f0~ &* (f0~ &* f1~))
```

The six combinations of 3-generator brackets and their equivalences in terms of PHB elements are calculated below. Notice the implicit use of the anti-commutativity law and Jacobi identity, the latter relating the first three or last three brackets.

```
> phbize(f0&*(f1&*f2),B);phbize(f2&*(f0&*f1),B);phbize(f1&*(f2&*f0),B);
> phbize(f0&*(f2&*f1),B);phbize(f2&*(f1&*f0),B);phbize(f1&*(f0&*f2),B);
           -(f2~ &* (f0~ &* f1~)) + (f1~ &* (f0~ &* f2~))
                          f2~ &* (f0~ &* f1~)
                        -(f1~ &* (f0~ &* f2~))
             (f2~ &* (f0~ &* f1~)) - (f1~ &* (f0~ &* f2~))
                        -(f2~ &* (f0~ &* f1~))
                          f1~ &* (f0~ &* f2~)
```

Algorithm This routine considers implicitly both the anti-symmetry property and the Jacobi identity, through the verification of the conditions for the construction of the PHB (cf. PHB algorithm 40), since the conversion of any bracket to an element, or a linear combination of the elements, in the PHB must result in an bracket that complies with the rules used for the construction of the PHB. As previously mentioned, the rules for the construction of the PHB take into account the anti-symmetry property and the Jacobi identity to select only the independent brackets from all possible combinations.

So the main steps of the `phbize` algorithm consist in verifying which rule or rules are violated by the Lie bracket, if it is not already in the PHB, and making the appropriate correction so that the condition is satisfied by the resulting bracket. Assuming the Lie any Lie bracket x can be decomposed into its left and right operands by respectively applying the functions lo and ro to x, i.e. $lo(x)$ and $ro(x$ return the left and right operand of x, respectively. And assuming also that the operations $lo(\cdot)$ and $ro(\cdot)$ can be composed iteratively to obtain, say the right operand of the left operand $ro(lo(x))$, then the main steps of the `phbize` procedure *in strict order* can be summarized as:

1. Verify $x \in B$.
2. Verify $lo(x) \neq ro(x)$.
3. Verify $lo(x) \in B$ and $ro(x) \in B$.
4. Verify $len(lo(x)) < len(ro(x)) \Leftrightarrow pos(lo(x), B) < pos(ro(x), B)$.
5. Verify $pos\left(lo(ro(x))\right) \leq pos\left(lo(x)\right)$.

Where the functions $len(x)$ and $pos(x, B)$ return the length (i.e. the number of operands) of the bracket x and the position of x in the basis B, respectively.

The above rules are related to the Lie bracket properties and the rules for the construction of the PHB. Particular connections can be made between the above conditions and the Lie bracket properties or the conditions in the `phb` algorithm (cf. p. 40). This relations are summarized in the Table 6.3 below.

`phbize` condition	`phb` condition	Lie product property
1	1	-
2	-	anti-symmetry
$3 \wedge 4$	3.a	anti-symmetry
4	2	anti-symmetry
5	3.b	Jacobi identity

Table 3: Connections between `phbize` conditions, `phb` construction rules and the Lie bracket properties.

All the conditions in the above table associated with the anti-symmetry property should imply the reordering (swapping) of the operands of x, with the exception of the `phbize` condition 2, which should return zero, since by the anti-symmetry property $[a,b] = [b,a] \Leftrightarrow [a,b] + [a,b] = 0 \Rightarrow [a,b] = [b,a] = 0$ only if $a = b$. In the case the bracket x does not satisfy the condition associated with the Jacobi identity, then `phbize` should return the sum of the two other terms in the Jacobi identity.

Figs. 6-7 show the flow chart illustrating the algorithm for the application for the verification of the above rules and the transformation of x into an element in the PHB B.

The functions $len(x)$ and $pos(x,B)$ mentioned above, have been implemented and called `bracketlen` and `posxinphb`. These two functions are available to the user, though their immediate use does not seem strictly necessary. Both, `bracketlen` and `posxinphb` are described in the next subsections.

Remarks At this point it is worth to make some remarks on how a procedure to construct a basis for some Lie algebra, such as the one described to construct the Philip Hall basis can be devised.

The first observation is the obvious dependency imposed by anti-symmetry property between a bracket and its commuted product. By taking into account this simply property, we are basically prescribing that we either consider $[f, g]$ or $-[g, f]$ but not both, so selecting the first bracket leads to establishing some order, in this case a lexicographical order. Thus, the ordered basis condition 1 for `phb` is somewhat implicit due to the anti-symmetry, which also leads to condition 2.

Next, if three ordered generators f, g, h are considered, it is not difficult to see that there are 6 possible ways in which their product can occur. This 6 ordered triples are fgh, fhg, hfg, hgf, ghf and gfh. Considering these combinations of f, g and h, the Jacobi identity can be written only in the next two ways:

$$\begin{array}{ccc} [f,[g,h]] & +[h,[f,g]] & +[g,[h,f]] = 0 \\ \Updownarrow & \Updownarrow & \Updownarrow \\ -[f,[h,g]] & -[h,[g,f]] & -[g,[f,h]] = 0 \end{array} \tag{36}$$

Note that the second equation simply results from the first one by applying the anti-symmetry property to the right operand of each Lie product. Note also that other forms of the Jacobi identity in which the left operand has length 2 are discarded since this violate the ordering rule 2 ($pos(lo(x), B) < len(ro(x), B)$).

So basically, all what is needed is to select the dependent and independent terms in the above Jacobi identity. The independent terms must belong to the basis B and therefore they must already be such that they satisfy the ordering and the anti-symmetry properties.

Checking the terms in (36) it is possible to identify those that do not satisfy the ordering condition 4, and the condition 5, as indicated in the equation (37), below. All the terms that do not satisfy 4 can easily be brought to a form that is the the basis B by simply swapping the elements in the right operand and making a sign correction. The exception to the last statement is the first term in the second equation of (37) which if corrected by swapping the operands of the right operand, then it violates condition 5 since $pos(lo(ro([f,[g,h]]))) = pos(g)$ is greater than $pos(lo([f,[g,h]])) = pos(f)$. Thus, the first term in (37) is the dependent term, since it cannot be brought into a form that is in the basis B by simply swapping operands in the right operand and therefore it will have to be expressed as the sum of the two terms that have been marked with a $\surd$ in (37), which are elements of the basis B.

$$\begin{aligned} &\overbrace{[f,[g,h]]}^{\times 5} \quad + [h,[\overset{\checkmark}{f},g]] \quad +[g,\overbrace{[h,f]}^{\times 4}] = 0 \\ -&[f,\overbrace{[h,g]}^{\times 4}] \quad -[h,\overbrace{[g,f]}^{\times 4}] \quad - [g,[\overset{\checkmark}{f},h]] = 0 \end{aligned} \tag{37}$$

So now the final equation for the dependent Lie product in terms of the independent basis brackets as

$$[f,[g,h]] := -[h,[f,g]] + [g,[f,h]] \tag{38}$$

which for an indeterminate bracket x can be written in the general form

$$\begin{aligned} [lo(x),[lo(ro(x)),ro(ro(x))]] :=& \\ &-[ro(ro(x)),[lo(x),lo(ro(x))]] \\ &+[lo(ro(x)),[lo(x),ro(ro(x))]] \end{aligned} \tag{39}$$

as also shown in the last process box of the flow chart in Fig. 7.

Sign removal The sign removal is not a function, but rather a process required to deal with brackets of the form $-x$, since the basis PHB B only contains the *positive* version of the brackets.

Suppose that x is indeed an element of B, then $-x$ does not need any processing, since it is evidently a bracket expressed in terms of $x \in B$, however from an implementation standpoint, checking the membership of $-x$ in B fails to return a positive answer, and the the program must be adapted to handle this case. This can easily be achieved by first checking if the symbolic variable has a minus sign in front; if so, then the sign must be stored and the membership to B of the negated bracket must be checked instead. Once some possible processes have been performed on the variable, the result is negated previous been returned to recover the sign of the original bracket passed as input to the `phbize` function (as well as some other functions of the LTP). This process of sign removal and restoration is shown in the flow chart of Fig. 10.

See Also `posxinphb(x)`, `bracketlen(x)`

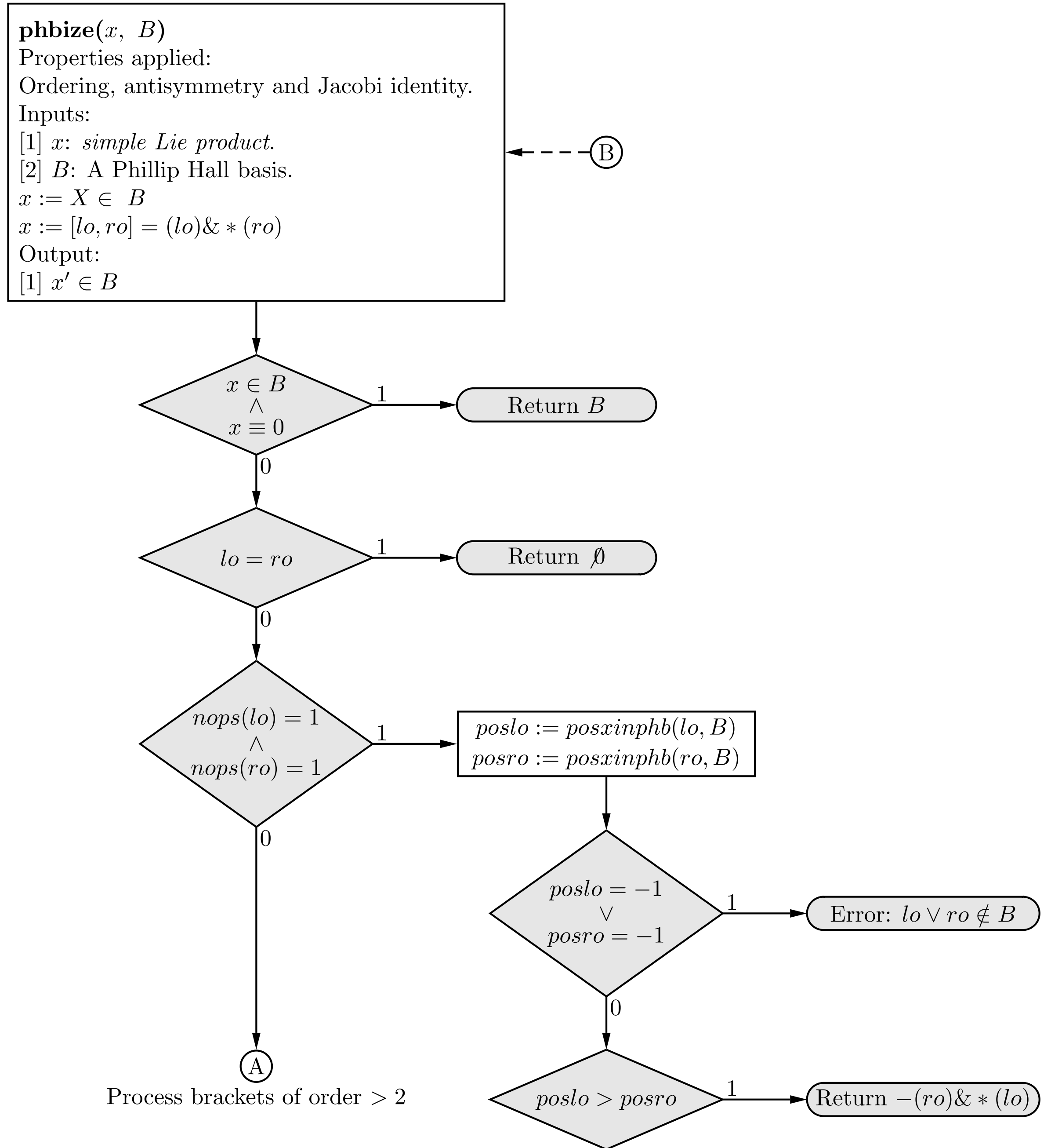


Figure 6: Flow chart for the `phbize` algorithm (contd. on Fig. 7).

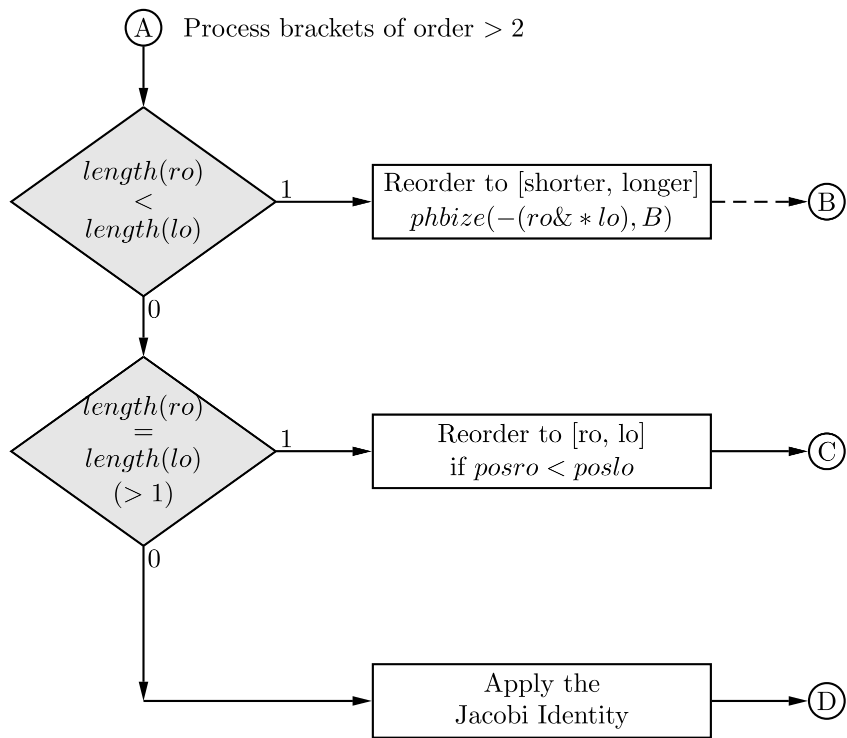


Figure 7: Flow chart for the `phbize` algorithm (contd. from Fig. 6).

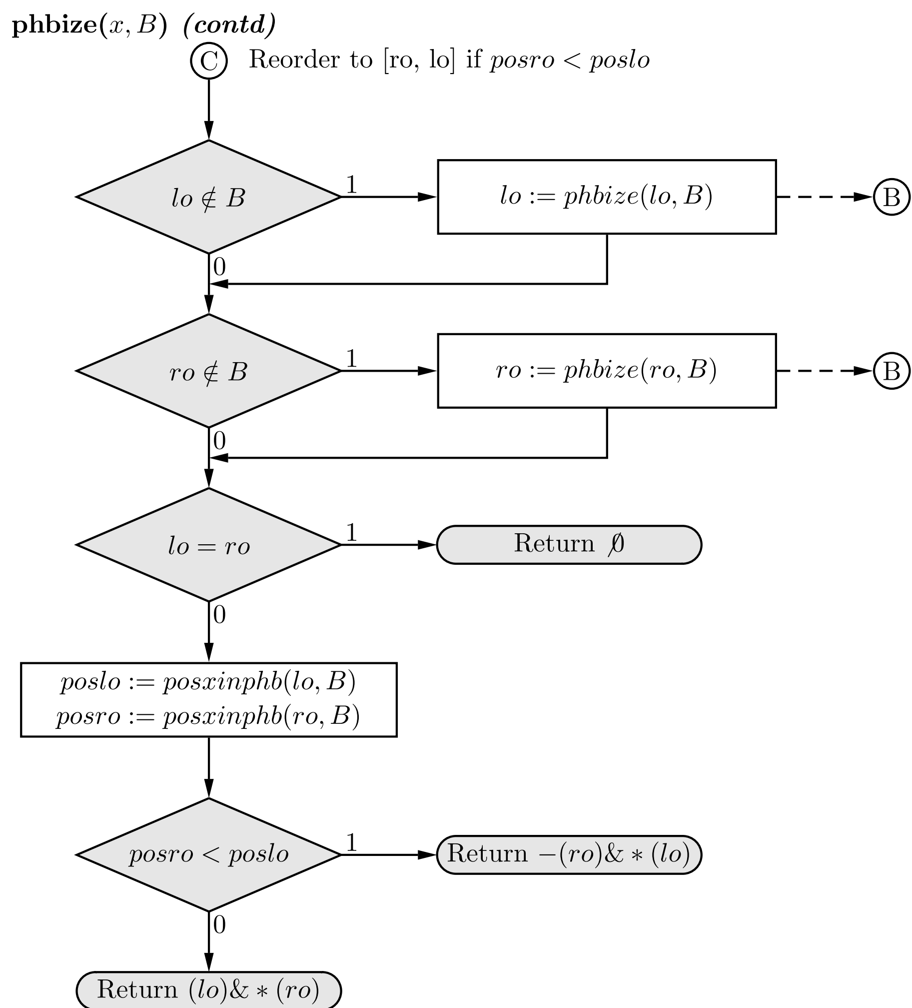


Figure 8: Flow chart for the `phbize` algorithm (contd. from Fig. 7).

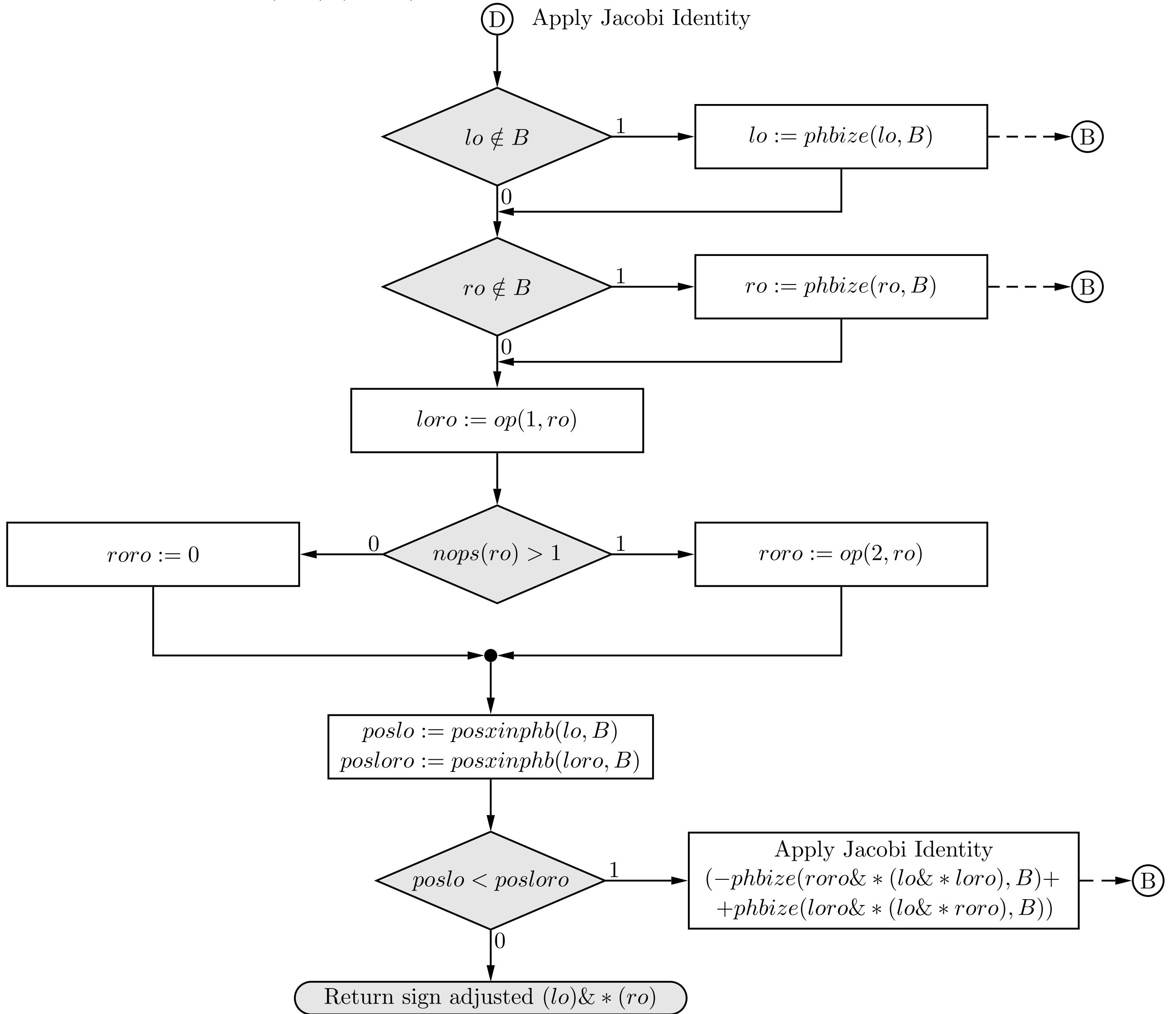


Figure 9: Flow chart for the `phbize` algorithm (contd. from Fig. 7).

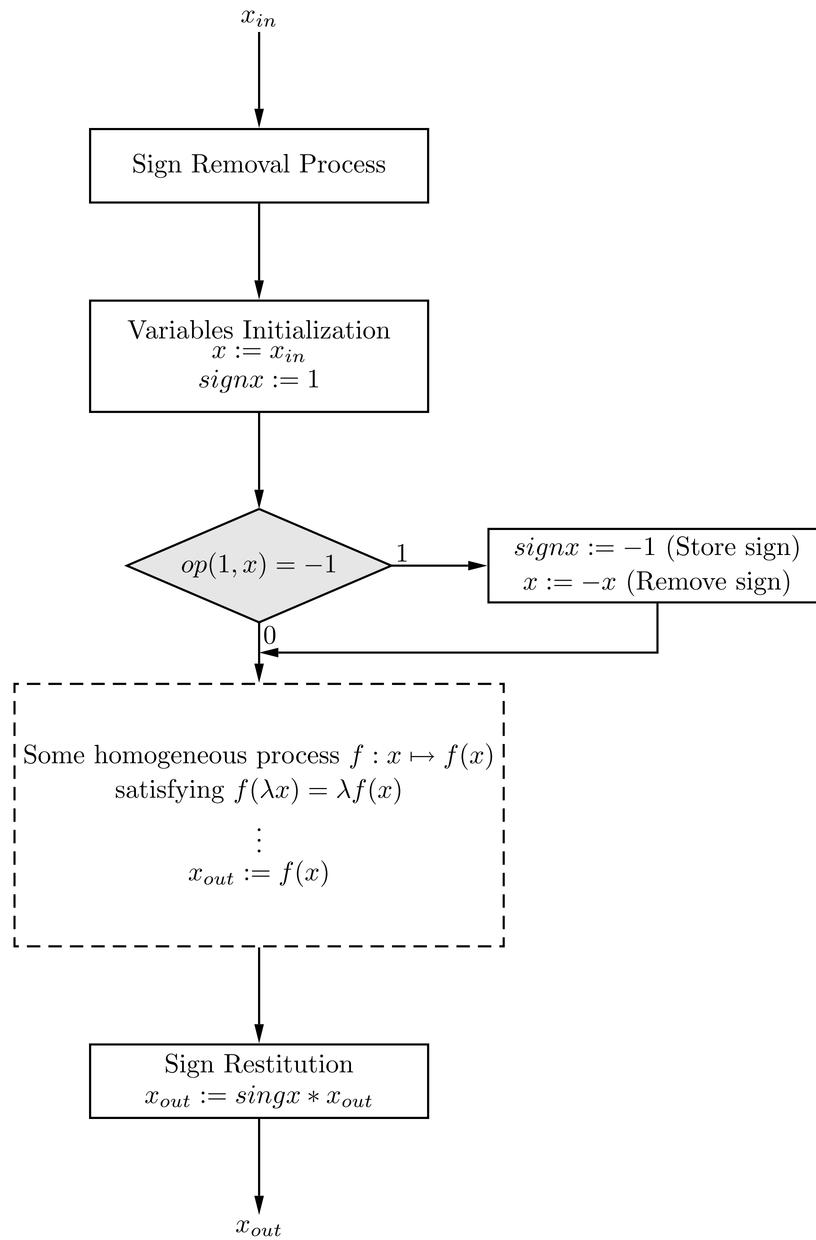


Figure 10: Flow chart for the sign removal procedure.

6.3.1 posxinphb

Purpose Return the position of the Lie bracket x in the basis B.

Syntax p:=posxinphb(x,B);

Description The command `posxinphb`, returns the position of x within the PHB, if it exists otherwise returns -1. It is assumed that x is a , i.e. the scalar parts have been removed (see function `simpLB` below which allows to obtain the pure bracket).

Arguments

x	Some *pure* Lie bracket.
B	Philip Hall basis

Examples Consider the PHB B, given in example for the function `phb` in p. 40. Then

```
> p:=posxinphb(f1~ &* (f1~ &* (f0~ &* f1~)),B);
p:=22
```

Notes This function is required by the `phbize` routine. It has been provided to the user for his/her convenience, however it is unlikely that it will really be needed.

6.3.2 bracketlen

Purpose Return the length of the Lie bracket x.

Syntax l:=bracketlen(x);

Description This function also expects a pure bracket as `posxinphb` (see `simpLB` below for the explanation of the pure bracket concept) and returns the length, also known as order or depth of the Lie bracket. The length of any generator is one, for any other Lie product x it is the number of terms (operands) in the expansion of x.

Arguments

x	Some *pure* Lie bracket.

Examples Consider the PHB B, given in example for the function `phb` in p. 40. Then the length of the third and twenty-second elements of B can be found as:

```
> l1:=bracketlen(B[3]);
> l2:=bracketlen(B[22]);
l1:=1
l2:=4
```

Notes This function is required by the `phbize` routine. It has been provided to the user for his/her convenience, however it is unlikely that it will really be needed.

6.4 simpLB

Purpose Simplify a Lie bracket according to the property of distributivity over scalar multiplication, and return the scalar part and the simplified *pure* Lie bracket.

Syntax xs:=simpLB(x);

Description The command `simpLB` simplifies any Lie bracket based on the distributivity over scalar multiplication property. This command returns the scalar terms of the Lie bracket grouped together multiplying a Lie monomial (i.e. a Lie bracket containing only generators of the Lie algebra without any scalar coefficients).

This function returns a list (a Maple sequence) of three elements: the simplified bracket in the form *scalar* ∗ *pure bracket*, the *scalar* part, the *pure bracket*, in the first, second and third positions of the list. The elements in the list can be accessed individually by appending the index selector after the name of the output variable or after the invocation of the function, i.e. if `xs:=simpLB(x)`, then `xs[2]` corresponds to the second element in the output list stored in `xs`. Similarly if only the second element is required, `simpLB` can be invoked as `z:=simpLB(x)[2]`; in this case `z` does not contain a list but only the second element of the list produced by `simpLB`.

Arguments x Any Lie bracket.

Examples As an example consider the simplification of $[\alpha f_2, [\alpha f_1, (\alpha + \beta^2) f_0]])$, which should return the scalar part $(\alpha^3 + \alpha^2\beta^2)$ and the pure bracket $[f_2, [f_1, f_0]]$. This can be easily achieved as follows:

```
> x:=(a*f2) &* ((a*f1) &* ((a+b^2)*f0));
                                                 2
      x := a~ f2~ &* (a~ f1~ &* (a~ + b~ ) f0~)
> z:=simpLB(x);
        3     2   2                             2          2
z := (a~  + a~  b~ ) (f2~ &* (f1~ &* f0~)), a~  (a~ + b~ ),

      f2~ &* (f1~ &* f0~)
> z[3];
                  f2~ &* (f1~ &* f0~)
```

Algorithm The algorithm behind the implementation of this function is illustrated in the flow chart of Fig. 11.

The main idea is to decompose each element in the Lie product into its scalar and *vector* (Lie indeterminate) part. If there is no Lie product operator `&*` standing between the operands, simply the scalar and vector parts are returned, otherwise the function recursively calls itself to further decompose the operands into scalar and vector parts. As shown in the bottom right process box of the flow chart in Fig. 11, the scalar part of x is the multiplication of the scalar parts of the left and right operands of x, while the pure bracket of x is the multiplication in the Lie product of the left and right *pure* operands of x, which are also obtained by calling `simpLB` with the respective operand as an argument, thus `simpLB` calls itself until no Lie product operator `&*` is found.

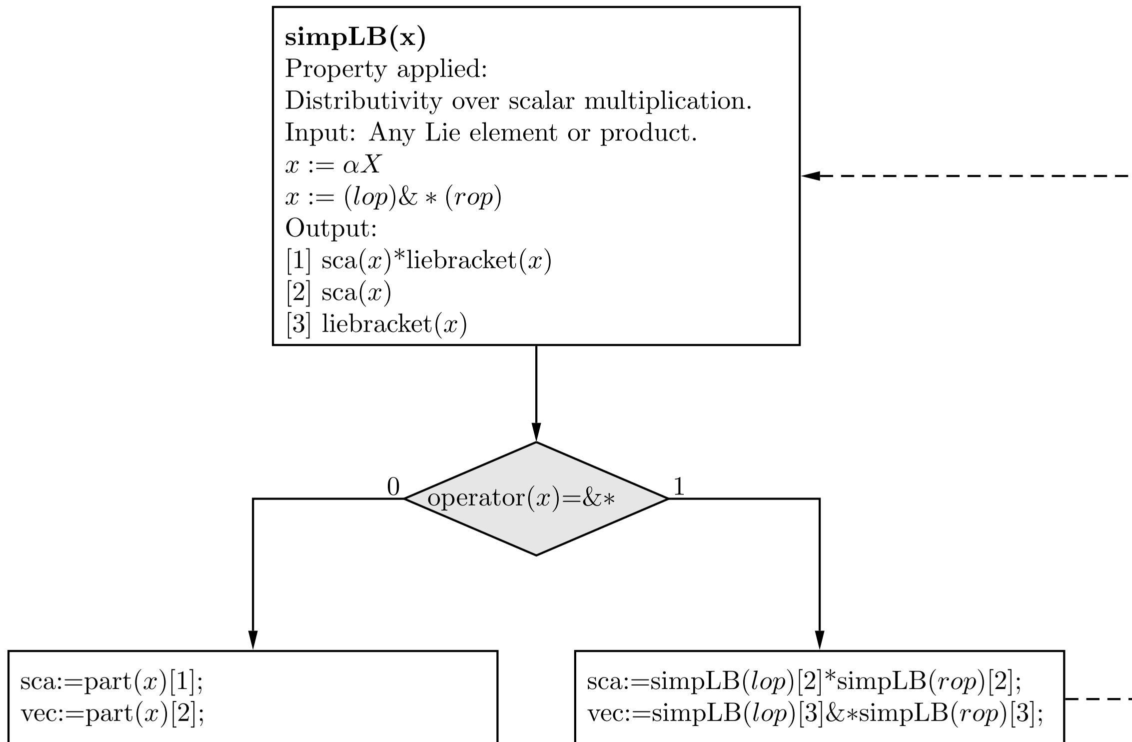


Figure 11: Flow chart for the `simpLB` function.

6.5 regroupLB

A possible alternative name for this function is `collectLB`.

Purpose Regroup (collect) summands in an expression in terms of its Lie brackets. The only properties applied are the distributivity over the Lie products and the distributivity over scalar multiplication. The anti-symmetry and Jacobi identities are not applied, however (cf. `reduceLB` instead). Neither reduces the terms to a form in the PHB, to this end use instead `reduceLB`, which is more powerful, but obviously requires a PHB.

Syntax p:=regroupLB(x);

Description The command `regroupLB` groups together brackets which are equal, in other words the scalar parts collected in terms of their common Lie bracket; i.e. the underlying properties applied are that of distributivity over the Lie products and over the scalar multiplication. The command invokes `simpLB` to obtain the scalar part and the pure (simplified) Lie bracket, the scalar parts multiplying equal brackets are summed and expressed as a new factor of the corresponding simplified Lie bracket (the common factor). This function is mostly based on finding like members in a list and conceptually simple, thus no further algorithmic details will be included.

Arguments x Some *polynomial* in the Lie brackets, i.e. a summation of Lie brackets.

Note The `regroupLB` does not apply the anti-commutativity or anti-symmetry property, neither the Jacobi identity. To reduce an expression to its simplest (shortest) form, use `reduceLB`, which is more powerful in the sense that it takes into account the anti-symmetry and Jacobi properties, however it requires the specification of a PHB.

Examples First consider as a simple example the following:

```
> z:=f0&*f1+(u0_1*f0)&*f1;
        z := (f0~ &* f1~) + (f0~ u0_1~ &* f1~)
> regroupLB(z);
              (1 + u0_1~) (f0~ &* f1~)
```

As a second, more complex example, consider the linear combination of vector fields `f_1`, `f_2` and `f_3` given in the example for the function `createLBobjects` on page 37. And let `z:=f_1&*(f_2+f_3);`, then

```
> z:=f_1&*(f_2+f_3);
  z:=(f0~ u0_1~ &* f0~ u0_2~) + (f0~ u0_1~ &* f1~ u1_2~)
      + (f0~ u0_1~ &* f2~ u2_2~) + (f0~ u0_1~ &* f0~ u0_3~)
      + (f0~ u0_1~ &* f1~ u1_3~) + (f0~ u0_1~ &* f2~ u2_3~)
      + (f1~ u1_1~ &* f0~ u0_2~) + (f1~ u1_1~ &* f1~ u1_2~)
      + (f1~ u1_1~ &* f2~ u2_2~) + (f1~ u1_1~ &* f0~ u0_3~)
      + (f1~ u1_1~ &* f1~ u1_3~) + (f1~ u1_1~ &* f2~ u2_3~)
      + (f2~ u2_1~ &* f0~ u0_2~) + (f2~ u2_1~ &* f1~ u1_2~)
      + (f2~ u2_1~ &* f2~ u2_2~) + (f2~ u2_1~ &* f0~ u0_3~)
      + (f2~ u2_1~ &* f1~ u1_3~) + (f2~ u2_1~ &* f2~ u2_3~)
> regroupLB(z4);
  (u0_1~ u0_2~ + u0_1~ u0_3~) (f0~ &* f0~)
          + (u0_1~ u1_2~ + u0_1~ u1_3~) (f0~ &* f1~)
          + (u0_1~ u2_2~ + u0_1~ u2_3~) (f0~ &* f2~)
          + (u1_1~ u0_2~ + u1_1~ u0_3~) (f1~ &* f0~)
          + (u1_1~ u1_2~ + u1_1~ u1_3~) (f1~ &* f1~)
          + (u1_1~ u2_2~ + u1_1~ u2_3~) (f1~ &* f2~)
          + (u2_1~ u0_2~ + u2_1~ u0_3~) (f2~ &* f0~)
          + (u2_1~ u1_2~ + u2_1~ u1_3~) (f2~ &* f1~)
          + (u2_1~ u2_2~ + u2_1~ u2_3~) (f2~ &* f2~)
```

The latter example clearly shows that only the distributivity over scalar multiplication is applied, since terms like `(f0~ &* f0~)` are not made zero, neither factors of, for example, `(f0~ &* f1~)` and `(f1~ &* f0~)` are collected together.

Implementation Notes

The first implementation of this function suffered of some memory problems, due to some problem in the way Maple handles the memory of sequences and lists. A work around this problem was to transform the list into an array and back into a list to eliminate certain elements in a simple ways using the `subsop` function from Maple. The details can be found in the source code.

It's worth to mention that for larger control systems and models, memory limitations might occur even if memory is *formally* allocated with the functions to declare arrays, especially if a high order CBH formula (cf. `cbhexp`, p. 68) is employed. A possible approach to face memory related obstacles is to decrease the order of the CBH used. The exponent returned by **cbhexp** is a sum of approximately $2m^n$ brackets, where m is the number of generators, n the order of the CBH formula employed. For example, in the case of three generators ($m = 3$) and a third order CBH ($n = 3$), the CBH exponent would have more than $2 \times 3^3 = 54$ elements in the sum. For the case of of $m = 3$ and $n = 4$ the number of elements increases drastically to more than 162, actually the number of summands is 231 when the elements of order smaller than 4 are taken into account.

6.6 reduceLB

Purpose Reduce an expression in terms of Lie brackets to its simplest form in terms of brackets in the PHB.

Syntax p:=reduceLB(x,B);

Description The routine `reduceLB` is similar to `regroupLB` explained above, but more powerful in the sense that it allows a greater simplification of a general Lie expression (any Lie polynomial, i.e. a summation of Lie products) by reducing every bracket to a valid product in the Philip Hall basis. The algorithmic implementation of this routine is simple and involves only three steps: regrouping the original input by invoking `regroupLB`, transforming every element in the regrouped expression to a valid PHB element by means of the `phbize` function, and regrouping the last result once more, again invoking `regroupLB`.

NOTE: This function accepts arguments x with brackets of order higher than the degree of nilpotency of the Lie algebra, and which might exceed the degree of the highest order brackets in the basis. Since this function relies on `phbize`, if there are terms in x exceeding the degree of nilpotency, but that are composed by the Lie product of two elements in the PHB, the result returned will be correct. However, for more complicated brackets the result will be a partial simplification containing brackets which do not really belong to the PHB, and thus the result must be interpreted with care.

Arguments

x	Any Lie polynomial.
B	Philip Hall basis

Example Consider the simplification of the exponent resulting from the composition of the flows corresponding to the vector fields `f_1` and `f_2` from the example for the function `createLBobjects` (p. 37. The exponent of the composition of flows can be calculated by means of the function `cbhexp` (p. 68. Assume the PHB B is the same one given in the example for the function `phb` in p. 40.

The expansion of the exponent is not shown below since its 231 terms require 7 double-spaced pages of 55 lines each, nevertheless it can be found in the examples included with the source code. Assume the PHB B is the same one given in the example for the function `phb` in p. 40.

Note in the example below how the resulting exponent `zr` has only 29 summands, compared to the 231 in the initial expression for the exponent `z`!

```
> z:=cbhexp(f_1,f_2):
> zr:=reduceLB(z,B);
  zr := (- 1/12 u2_2~ u0_1~ u1_2~ + 1/12 u0_1~ u2_1~ u1_2~

         - 1/12 u2_1~ u1_1~ u0_2~ + 1/12 u0_2~ u1_1~ u2_2~)

        (f2~ &* (f0~ &* f1~)) + (- 1/12 u0_2~ u0_1~ u1_2~ ...
```

```
                 2                   2
...  + 1/12 u0_1~  u1_2~ + 1/12 u0_2~  u1_1~

     - 1/12 u0_1~ u1_1~ u0_2~) (f0~ &* (f0~ &* f1~))

     + (- 1/2 u2_1~ u1_2~ + 1/2 u1_1~ u2_2~) (f1~ &* f2~) + (

    - 1/24 u1_2~ u1_1~ u0_1~ u2_2~ + 1/24 u2_1~ u0_2~ u1_1~ u1_2~

    ) (f1~ &* (f1~ &* (f0~ &* f2~))) + (1/12 u1_2~ u2_1~ u0_2~

     - 1/12 u2_1~ u1_1~ u0_2~ - 1/12 u2_2~ u0_1~ u1_2~

     + 1/12 u0_1~ u1_1~ u2_2~) (f1~ &* (f0~ &* f2~)) +

                      2                          2
    (1/24 u2_1~ u0_2~  u0_1~ - 1/24 u0_2~ u0_1~  u2_2~)

    (f0~ &* (f0~ &* (f0~ &* f2~)))

     + (1/2 u0_1~ u1_2~ - 1/2 u1_1~ u0_2~) (f0~ &* f1~) + (

              2
    1/12 u0_2~  u2_1~ - 1/12 u0_1~ u2_1~ u0_2~

                                             2
     - 1/12 u2_2~ u0_1~ u0_2~ + 1/12 u0_1~  u2_2~)

                                           2
    (f0~ &* (f0~ &* f2~)) + (- 1/12 u1_2~  u0_1~

     + 1/12 u0_1~ u1_1~ u1_2~ + 1/12 u0_2~ u1_1~ u1_2~

                 2
     - 1/12 u1_1~  u0_2~) (f1~ &* (f0~ &* f1~)) +

               2      2              2      2
    (1/24 u0_2~  u1_1~  - 1/24 u1_2~  u0_1~ )

                                                    2
    (f1~ &* (f0~ &* (f0~ &* f1~))) + (- 1/12 u2_1~  u0_2~

     + 1/12 u0_2~ u2_1~ u2_2~ + 1/12 u2_1~ u0_1~ u2_2~ ...
```

```
                2
...  - 1/12 u2_2~  u0_1~) (f2~ &* (f0~ &* f2~))

                                           2
     + (u2_2~ + u2_1~) f2~ + (1/12 u1_2~  u2_1~

     - 1/12 u2_1~ u1_1~ u1_2~ - 1/12 u1_2~ u1_1~ u2_2~

                2
     + 1/12 u1_1~  u2_2~) (f1~ &* (f1~ &* f2~))

     + (u1_2~ + u1_1~) f1~ + (u0_1~ + u0_2~) f0~ + (

                                                        2
    1/48 u2_1~ u0_2~ u1_1~ u1_2~ + 1/48 u0_2~ u1_1~  u2_2~

                2
     - 1/48 u1_2~  u2_1~ u0_1~ - 1/48 u1_2~ u1_1~ u0_1~ u2_2~)

    (f2~ &* (f1~ &* (f0~ &* f1~))) +

               2      2              2      2
    (1/24 u0_2~  u2_1~  - 1/24 u2_2~  u0_1~ )

    (f2~ &* (f0~ &* (f0~ &* f2~))) +

                       2                   2
    (- 1/24 u0_2~ u0_1~  u1_2~ + 1/24 u0_2~  u0_1~ u1_1~)

    (f0~ &* (f0~ &* (f0~ &* f1~))) + (

                                                         2
    - 1/48 u2_1~ u2_2~ u0_1~ u1_2~ - 1/48 u1_1~ u2_2~  u0_1~

                       2
     + 1/48 u0_2~ u2_1~  u1_2~ + 1/48 u2_1~ u0_2~ u1_1~ u2_2~)

    (f2~ &* (f1~ &* (f0~ &* f2~))) +

                     2                   2
    (1/24 u0_2~ u1_1~  u1_2~ - 1/24 u1_2~  u1_1~ u0_1~)

    (f1~ &* (f1~ &* (f0~ &* f1~))) + ...
```

```
... (1/24 u2_1~ u0_2~ u1_1~ u2_2~ - 1/24 u2_1~ u2_2~ u0_1~ u1_2~)

    (f2~ &* (f2~ &* (f0~ &* f1~))) + (

                                                       2
    - 1/48 u2_1~ u0_2~ u0_1~ u1_2~ - 1/48 u1_2~ u0_1~  u2_2~

                  2
     + 1/48 u0_2~  u2_1~ u1_1~ + 1/48 u0_2~ u1_1~ u0_1~ u2_2~)

    (f2~ &* (f0~ &* (f0~ &* f1~)))

     + (- 1/2 u2_1~ u0_2~ + 1/2 u0_1~ u2_2~) (f0~ &* f2~) +

                      2                          2
    (1/24 u0_2~ u2_1~  u2_2~ - 1/24 u0_1~ u2_2~  u2_1~)

    (f2~ &* (f2~ &* (f0~ &* f2~))) +

                        2                    2
    (- 1/24 u1_2~ u1_1~  u2_2~ + 1/24 u1_2~  u2_1~ u1_1~)

    (f1~ &* (f1~ &* (f1~ &* f2~))) +

              2                          2
    (1/24 u2_1~  u1_2~ u2_2~ - 1/24 u2_2~  u2_1~ u1_1~)

    (f2~ &* (f2~ &* (f1~ &* f2~))) + (1/12 u2_1~ u1_1~ u2_2~

                  2                    2
     - 1/12 u2_2~  u1_1~ - 1/12 u2_1~  u1_2~

     + 1/12 u1_2~ u2_1~ u2_2~) (f2~ &* (f1~ &* f2~)) +

              2      2              2      2
    (1/24 u1_2~  u2_1~  - 1/24 u1_1~  u2_2~ )

                                                2
    (f2~ &* (f1~ &* (f1~ &* f2~))) + (1/48 u0_2~  u2_1~ u1_1~

     + 1/48 u2_1~ u0_2~ u0_1~ u1_2~

                                                        2
     - 1/48 u0_2~ u1_1~ u0_1~ u2_2~ - 1/48 u1_2~ u0_1~  u2_2~)

    (f1~ &* (f0~ &* (f0~ &* f2~)))

> nops(zr);

    29
```

6.7 cbhexp

Purpose Calculate the exponent corresponding to the composition of flows of vector fields according to the CBH formula (up to order 4 brackets).

Syntax z:=cbhexp(x,y);

Description The function `cbhexp` computes and returns the exponent Z in $e^Z = e^X \circ e^Y$, the composition of flows of two vector fields X and Y, calculated according to the Campbell-Baker-Hausdorff (CBH) formula for the composition of the flows (exponential mappings) acting on the right. The exponent Z is approximated by the 4th order CBH formula given below (see[43]):

$$\begin{aligned} Z &= X + Y + \frac{1}{2}[X,Y] + \frac{1}{12}\left([X,[X,Y]] - [Y,[X,Y]]\right) \\ &\quad -\frac{1}{48}\left([Y,[X,[X,Y]]] + [X,[Y,[X,Y]]]\right) \end{aligned}$$

Arguments x, y Lie algebra generators or Lie polynomials, i.e. linear combinations of Lie elements (brackets).

Examples See example for the function `reduceLB` on page 63.

6.8 evalLB2expr

Purpose Evaluate a LB to a symbolic or Maple expression.

Syntax e:=evalLB2expr(x,vars);

Description The command `evalLB2expr`, takes a Lie monomial as input (cf. `simpLB`, p. 59 and returns two expressions with the expansion of the Lie bracket according to its definition involving the partial derivatives of the vector fields. The first expression of the output displays an *implicit* evaluation of the bracket (i.e. the derivatives are not actually computed). The second expression corresponds to a form in terms of the Jacobians that are not evaluated until the *explicit* definition of the vector fields takes place.

Arguments

x	A *pure* Lie bracket.
$vars$	Array or vector of variables in terms of which the elements of the indeterminates are defined, e.g. $vars = [x1, ..., xn]$ if $fi_j : (x1, ..., xn) \rightarrow fi_j(x1, ..., xn)$. Where fi_j is the jth element of the indeterminate fi.

Examples The following examples show respectively the *implicit* and *explicit* evaluation of a second order and a third order Lie bracket.

```
> z1:=g0&*g1;
> z2:=g1&*(g0&*g2);
> evalLB2expr(z1,x)[1];z1e:=evalLB2expr(z1,x)[2];
> evalLB2expr(z2,x)[1];z2e:=evalLB2expr(z2,x)[2];
```

```
                    z1 := g0 &* g1


                 z2 := g1 &* (g0 &* g2)


                /d     \        /d     \
                |-- g1| g0 - |-- g0| g1
                \dx    /        \dx    /


  z1e := matadd(multiply(jacobian(g1, x), g0),

        -multiply(jacobian(g0, x), g1))
```

```
/d  //d    \      /d    \   \\
|-- ||-- g2| g0 - |-- g0| g2|| g1
\dx \\dx   /      \dx   /   //

         /d    \ //d    \      /d    \   \
       - |-- g1| ||-- g2| g0 - |-- g0| g2|
         \dx   / \\dx   /      \dx   /   /


z2e := matadd(multiply(jacobian(matadd(

      multiply(jacobian(g2, x), g0), -multiply(jacobian(g0, x), g2))

      , x), g1), -multiply(jacobian(g1, x), matadd(

      multiply(jacobian(g2, x), g0), -multiply(jacobian(g0, x), g2))))
```

```
> g0:=vector([-x2,u^2*x1]);
> g1:=vector([tan(x1),tan(x2)]);
> g2:=vector([x2,1]);
> x:=[x1,x2];
```

```
                        [      2   ]
                  g0 := [-x2, u  x1]


               g1 := [tan(x1), tan(x2)]


                    g2 := [x2, 1]


                    x := [x1, x2]
```

```
> with(linalg,jacobian,multiply,matadd);
```

```
              [jacobian, multiply, matadd]
```

```
> eval(z1e);
> eval(z2e);
```

```
[             2                             2   2       2        ]
[-(1 + tan(x1) ) x2 + tan(x2), (1 + tan(x2) ) u  x1 - u  tan(x1)]
```

```
[ 2                          2    2
[u  tan(x1) - (1 + tan(x1) ) (u  x1 + 1),

        2                          2   2   ]
      -u  tan(x2) + (1 + tan(x2) ) u  x2]
```

Notes The following issues must be considered when using `evalLB2expr`:

- The input Lie bracket must be a pure bracket.
- Note the functions `jacobian`, `multiply` and `matadd` (from the `linalg` package) shouldn't be in the Maple environment before calling `evalLB2expr`, otherwise they must be unassigned using `unassign`.
- The Lie algebra generators must be free of assumptions. The vector type assumption might be a frequent cause of problems if the user forgets to remove this or any other assumption from the variable. The reason for having assumption-free variables is that the function `jacobian` from the `linalg` package cannot handle variables on which some previous assumption was made.
- To obtain a fully (explicitly) evaluated Lie bracket it is necessary to load the `linalg` package or at least the functions within this package shown in the above example.
- The invocation of the commands should follow this particular order: define the Lie monomial, calculate its expansion with `evalLB2expr`, define the corresponding generators, load the `linalg` package (jacobian, multiply and matadd commands), and finally, evaluate the expansion via `eval`.

Operator Substitution

It's worth to mention that this function is simple thanks to the capability of Maple for performing operator substitution. The code for this routine is perhaps a good example on how to perform an operator substitution.

6.9 calcLB

Purpose Calculates a LB in terms of the definitions for each generator of an algebra of vector fields.

Syntax e:=calcLB(x,listgen,listgendef,vars);

Description This function performs an explicit calculation of Lie bracket of vector fields, i.e. calculates the Jacobians of the vector fields and makes the appropriate multiplications between the Jacobians and the vector fields. To this end, the function requires a list with the names of the symbolic variables that represent the vector fields, a list with the actual definitions of the corresponding vector fields, and an array with the variables in terms of which the vector fields are defined.

Arguments

x A *pure* Lie bracket.

$listgen$ List of Lie algebra generators (indeterminates). e.g. $[f0, f1, f2]$.

$listgendef$

List with the respective definition for each indeterminate in $listgen$. It is assumed that the indeterminates are vector fields, thus the definition of each indeterminate should be stored in a vector type. e.g. Let

```
> f0d:=vector([-x2,a*x1]);
> f1d:=vector([x1*x2,x2]);
> f2d:=vector([0;2*x1^2]);
```

then $listgen$ should be $[f0d, f1d, f2d]$.

$vars$ Array or vector of variables in terms of which the elements of the indeterminates are defined, e.g. $vars = [x1, ..., xn]$ if $fi_j : (x1, ..., xn) \rightarrow fi_j(x1, ..., xn)$. Where fi_j is the jth element of the indeterminate fi.

Example The following examples show respectively the *implicit* and *explicit* evaluation of a second order and a third order Lie bracket. In the following example the Lie brackets `z1` and `z2` are defined first. Note that `z1` is defined in terms of Lie elements `f0` and `f1`, while `z2` is expressed in terms of the Lie elements `g0`, `g1` and `g2`. The explicit evaluations of the brackets `z1` and `z2` are given in `z1e` and `z2e`, respectively. Next the definitions for the `g`'s are given, but not for the `f`'s. Notice that once the `linalg` package has been loaded, the evaluation of `z2e`, using Maple's `eval`, actually calculates (fully evaluates) the Lie bracket according to the definitions of the `g`'s, while the `eval(z1e)` returns an error, obviously because the `f`'s have not been explicitly defined. However, using `calcLB` at the end of this example, the bracket `z1` is explicitly calculated. The arguments to `calcLB` are `z=[f0,f1]` a list containing the elements in terms of which the bracket `z1` is defined, a second list with the corresponding definitions of `f0` and `f1` given respectively in `g0` and `g1`, and finally the array `x` of variables in terms of which the `g`'s are defined.

```
> z1:=f0&*f1;
> z2:=g1&*(g0&*g2);
> z1e:=evalLB2expr(z1,x)[2];
> z2e:=evalLB2expr(z2,x)[2];
```

```
                            z1 := f0~ &* f1~

                         z2 := g1 &* (g0 &* g2)


  z1e := matadd(multiply(jacobian(f1~, x), f0~),

        -multiply(jacobian(f0~, x), f1~))


  z2e := matadd(multiply(jacobian(matadd(

        multiply(jacobian(g2, x), g0), -multiply(jacobian(g0, x), g2))

        , x), g1), -multiply(jacobian(g1, x), matadd(

        multiply(jacobian(g2, x), g0), -multiply(jacobian(g0, x), g2))))
```

```
> g0:=vector([-x2,u^2*x1]);
> g1:=vector([tan(x1),tan(x2)]);
> g2:=vector([x2,1]);
> x:=[x1,x2];

                                  [       2    ]
                           g0 := [-x2, u   x1]

                        g1 := [tan(x1), tan(x2)]

                             g2 := [x2, 1]

                             x := [x1, x2]

> with(linalg,jacobian,multiply,matadd);

                     [jacobian, multiply, matadd]


> eval(z1e); # Just to show that it cannot be evaluated without
>            # the proper definitions for the vector fields...
Error, (in jacobian) wrong number (or type) of arguments

> eval(z2e);

  [ 2                      2    2
  [u  tan(x1) - (1 + tan(x1) ) (u  x1 + 1),

         2                      2   2   ]
       -u  tan(x2) + (1 + tan(x2) ) u  x2]

> # z[1..2] are the first to components of the previously
> # defined z:=phb(3,4); (see the example for the phb function,
> # in which z[1..2] corresponds to the list '[f0, f1]').
> calcLB(z1,z[1..2],[g0,g1],x);

  [            2                            2   2       2        ]
  [-(1 + tan(x1) ) x2 + tan(x2), (1 + tan(x2) ) u  x1 - u  tan(x1)]
```

Notes This function invokes the procedure `evalLB2expr` in order to perform the calculation, but does not require to load the `linalg` package.

See Also `evalLB2expr`.

6.10 selectLB

Purpose Select Lie brackets of certain order in a given expression.

Syntax e:=selectLB(x,k,s)

Description Selects the Lie brackets in a given expression `x` which are of order smaller, equal or greater than `k`, according to whether the third argument `s` is less, equal or greater than zero, respectively.

Arguments

x	Expression containing Lie brackets (list, sum, etc.).
k	Bracket degree (also referred to as order or length).
s	< 0 - Select brackets of order $< k$. $= 0$ - Select brackets of order $= k$. > 0 - Select brackets of order $> k$.

Examples Consider the expression `zr` in the example for the function `reduceLB` on page 63. The the summation of brackets of order 2 can be obtained by invoking:

```
> selectLB(zr,2,0);

  (1/2 u0_1~ u2_2~ - 1/2 u2_1~ u0_2~) (f0~ &* f2~)

     + (- 1/2 u1_1~ u0_2~ + 1/2 u0_1~ u1_2~) (f0~ &* f1~)

     + (- 1/2 u2_1~ u1_2~ + 1/2 u1_1~ u2_2~) (f1~ &* f2~)
```

6.11 createSubsRel

Purpose Create substitution relations for the exponent expression resulting from the CBH composition.

Syntax e:=createSubsRel(nGen,sLen);

Description Creates substitution relations to replace the scalars in the standard result from the composition of flows via the function `cbhexp`. Namely, the scalars `u_i_k` will be replaced by `epsilon_k*u_i_k`. The system is assumed to be drift free, i.e. there is a control `u_0_k`, for all `k`, that multiplies the first vector field `f_0`. If the system has drift then set `u_0_k`=1, for all `k`. Use the substitution relation in conjunction with the command `subs` as illustrated in the example.

The function returns three lists of substitution relations, in which:

- The first list contains basic substitution relations of the `u_i_k`=`epsilon_k*u_i_k`, where `epsilon_k` represents the length of the time interval for which the controls `u_i_k` are applied.
- The second list sets `u_0_k`=1 for all `k`. This list is necessary if the system is a system with drift.
- The third list sets `epsilon_k`=`epsilon`, i.e. these substitution relations must be used if the time intervals are equal duration.

Arguments

nGen	Number of Lie algebra generators.
sLen	Length of the sequence of inputs, i.e. the number of switching in the input sequence.

Examples Consider a nilpotent Lie algebra of degree 3, generated by 3 vector fields, and a sequence of inputs of length 2. The corresponding expressions for span of generators at each instant of the input sequence is given below.

```
f_1:=f0*u0_1+f1*u1_1+f2*u2_1
f_2:=f0*u0_2+f1*u1_2+f2*u2_2
```

The composition of the above `f_1` and `f_2` using the CBH, yields after simplification the following expression:

```
cf := (1/2 u1_1~ u2_2~ - 1/2 u2_1~ u1_2~) (f1~ &* f2~)

       + (u0_1~ + u0_2~) f0~ + (u1_1~ + u1_2~) f1~ + (

      - 1/12 u1_1~ u2_1~ u0_2~ + 1/12 u0_2~ u2_1~ u1_2~ ...
   ... + 1/12 u0_1~ u1_1~ u2_2~ - 1/12 u1_2~ u0_1~ u2_2~)

      (f1~ &* (f0~ &* f2~)) + (- 1/12 u0_1~ u2_1~ u0_2~

                   2                   2
       + 1/12 u0_2~  u2_1~ + 1/12 u0_1~  u2_2~

       - 1/12 u0_2~ u0_1~ u2_2~) (f0~ &* (f0~ &* f2~)) +

       + (1/2 u0_1~ u2_2~ - 1/2 u2_1~ u0_2~) (f0~ &* f2~)

       + (u2_1~ + u2_2~) f2~

       + (1/2 u0_1~ u1_2~ - 1/2 u0_2~ u1_1~) (f0~ &* f1~) + (

                                       2
      1/12 u0_1~ u1_1~ u1_2~ - 1/12 u1_2~  u0_1~

                   2
       - 1/12 u1_1~  u0_2~ + 1/12 u0_2~ u1_1~ u1_2~)

                                         2
      (f1~ &* (f0~ &* f1~)) + (- 1/12 u2_1~  u1_2~

       + 1/12 u1_2~ u2_1~ u2_2~ + 1/12 u1_1~ u2_1~ u2_2~

                   2
       - 1/12 u2_2~  u1_1~) (f2~ &* (f1~ &* f2~)) + (

      1/12 u0_1~ u2_1~ u1_2~ - 1/12 u1_2~ u0_1~ u2_2~

       - 1/12 u1_1~ u2_1~ u0_2~ + 1/12 u0_2~ u1_1~ u2_2~)

      (f2~ &* (f0~ &* f1~)) + (- 1/12 u0_1~ u1_1~ u0_2~

                   2                   2
       + 1/12 u0_2~  u1_1~ + 1/12 u0_1~  u1_2~

       - 1/12 u0_2~ u0_1~ u1_2~) (f0~ &* (f0~ &* f1~)) + (

                                       2
      1/12 u0_1~ u2_1~ u2_2~ - 1/12 u2_2~  u0_1~

                   2
       - 1/12 u2_1~  u0_2~ + 1/12 u0_2~ u2_1~ u2_2~)

                                       2
      (f2~ &* (f0~ &* f2~)) + (1/12 u1_1~  u2_2~

       - 1/12 u1_2~ u1_1~ u2_2~ - 1/12 u1_1~ u2_1~ u1_2~

                   2
       + 1/12 u1_2~  u2_1~) (f1~ &* (f1~ &* f2~))
```

Now the substitution relations are created with `nGen`=3 and `sLen`=2.

```
> srl:=createSubsRel(nGen,sLen);

  srl := {u0_2~ = epsilon2~ u0_2~, u1_2~ = epsilon2~ u1_2~,

        u2_2~ = epsilon2~ u2_2~, u0_1~ = epsilon1~ u0_1~,

        u1_1~ = epsilon1~ u1_1~, u2_1~ = epsilon1~ u2_1~},

        {u0_2~ = 1, u0_1~ = 1},

        {epsilon2~ = epsilon~, epsilon1~ = epsilon~}
```

The final step requires the use the command `subs` to simplify the expression for `cf` using the relations in `srl` as follows:

```
> cfr:=subs(srl[3],subs(srl[2],subs(srl[1],cf)));

                     2                           2
  cfr := (1/2 epsilon~  u1_1~ u2_2~ - 1/2 epsilon~  u2_1~ u1_2~)

        (f1~ &* f2~) + 2 epsilon~ f0~

         + (epsilon~ u1_1~ + epsilon~ u1_2~) f1~ + (

                      3                           3
        - 1/12 epsilon~  u1_1~ u2_1~ + 1/12 epsilon~  u2_1~ u1_2~

                       3                           3
         + 1/12 epsilon~  u1_1~ u2_2~ - 1/12 epsilon~  u1_2~ u2_2~)

        (f1~ &* (f0~ &* f2~))

                       2                     2
         + (1/2 epsilon~  u2_2~ - 1/2 epsilon~  u2_1~) (f0~ &* f2~)

         + (epsilon~ u2_1~ + epsilon~ u2_2~) f2~

                       2                     2
         + (1/2 epsilon~  u1_2~ - 1/2 epsilon~  u1_1~) (f0~ &* f1~)

                       3                           3      2
         + (1/6 epsilon~  u1_1~ u1_2~ - 1/12 epsilon~  u1_2~

                       3      2
         - 1/12 epsilon~  u1_1~ ) (f1~ &* (f0~ &* f1~)) + (

                      3      2
        - 1/12 epsilon~  u2_1~  u1_2~

                       3
         + 1/12 epsilon~  u1_2~ u2_1~ u2_2~ ...
```

```
                    3
...  + 1/12 epsilon~  u1_1~ u2_1~ u2_2~

                    3      2
      - 1/12 epsilon~  u2_2~  u1_1~) (f2~ &* (f1~ &* f2~)) + (

                   3                            3
     - 1/12 epsilon~  u1_1~ u2_1~ + 1/12 epsilon~  u2_1~ u1_2~

                    3                            3
      + 1/12 epsilon~  u1_1~ u2_2~ - 1/12 epsilon~  u1_2~ u2_2~)

                                        3
     (f2~ &* (f0~ &* f1~)) + (1/6 epsilon~  u2_1~ u2_2~

                    3      2                3      2
      - 1/12 epsilon~  u2_2~  - 1/12 epsilon~  u2_1~ )

                                         3      2
     (f2~ &* (f0~ &* f2~)) + (1/12 epsilon~  u1_1~  u2_2~

                    3
      - 1/12 epsilon~  u1_2~ u1_1~ u2_2~

                    3
      - 1/12 epsilon~  u1_1~ u2_1~ u1_2~

                    3      2
      + 1/12 epsilon~  u1_2~  u2_1~) (f1~ &* (f1~ &* f2~))
```

Note the *sequential* (and not simultaneous) substitution, first of the relations in `srl[1]`, in the inner function call, then of those in `srl[2]` and finally those in `srl[3]`. If the system has no drift then only the first (inner-most) substitution is required. The third substitution must be done if the time intervals in the sequence of inputs are equal.

6.12 codeCBHcf

Purpose Generate code in C or Fortran for the scalar coefficients of some Lie expression in terms of Lie products (not necessarily in a PHB).

Syntax e:=codeCBHcf(expr,B,language);

Description Generates code in C or Fortran for the scalar coefficients of some Lie expression in terms of Lie products (not necessarily in a PHB).

Arguments

expr	A Lie algebra generator, bracket or Lie polynomial.
B	A Philip Hall basis or an empty list.
language	Either of the strings `C` or `fortran`.

Examples Consider the P. Hall basis `B`, of example for the function `phb` on page 40, and the Lie polynomial `zr` given in the example for the function `reduceLB` on page 63. Then, the expressions in the C language for the scalar coefficients of a given Lie polynomial, for example, given by the first and third terms in `zr` can be obtained as:

```
> codeCBHcf(op(1,zr)+op(3,zr)*epsilon,z,C);
1.  B[12] = `&*`(f2,`&*`(f0,f1))
      t0 = -u2_2*u0_1*u1_2/12.0+u0_1*u2_1*u1_2/12.0
           -u2_1*u1_1*u0_2/12.0+u0_2*u1_1~*u2_2/12.0;
2.  B[6] = `&*`(f1,f2)
      t0 = (-u2_1*u1_2+u1_1*u2_2)*epsilon/2.0;
```

Providing a P. Hall basis is not essential, however a list or set containing at least one element must be given instead, as shown by the next example.

```
> codeCBHcf(op(1,z6r)+op(3,z6r)*epsilon,[[]],fortran);
1.  B[-1] = []
      t0 = -u2_2*u0_1*u1_2/12+u0_1*u2_1*u1_2/12
     #-u2_1*u1_1*u0_2/12+u0_2*u1_1*u2_2/12
2.  B[-1] = []
      t0 = (-u2_1*u1_2+u1_1*u2_2)*epsilon/2
```

Note above that the element which would correspond to a P. Hall basis is simply a list containing an empty list, `[[]]`, but could have been also `[``]`, or any other list containing one element, but never an empty expression, such as `[]`.

6.13 reduceLBT

Purpose Substitute Lie Bracket Table relations in a Lie polynomial written in terms of PHB elements.

Syntax e:=reduceLBT(x,B,lbt);

Description This procedure is similar to `reduceLB`, it simplifies a Lie polynomial given in x to a expression in terms of elements in the PHB in B, but additionally substitutes the dependent brackets in the PHB according to Lie bracket table relations passed to the procedure in *lbt*. The table defining the dependent brackets in terms of independent ones is passed as a list. In the case that all elements in the PHB are linearly independent, then *lbt* should be an empty set or list, and the result returned by `reduceLBT` will be equal to that obtained by using `reduceLB` if there are no brackets of order higher than the degree of nilpotency. To make this clearer we stress the following characteristic of this procedure:

Unlike `reduceLB`, this procedure eliminates from the expression x all those Lie brackets of order higher than the degree of nilpotency, i.e. sets them to zero.

Arguments

x	Any Lie polynomial.
B	The Philip Hall basis.
lbt	The Lie bracket table (lbt) is a set or list of substitution relations defining the dependencies between the elements of the PHB. For example, if `B` denotes the PHB, then Lie bracket table could be defined as: `lbt:=[B[4]=B[1]+2*B[2],B[6]=0,B[7]=B[5]]` The Lie bracket table could also be an empty list or set, e.g. `lbt:={}`.

Examples Consider the Lie polynomial `x` given by:

```
                        2
x := 1/2 dg3 g1  (f0 &* (f0 &* f2))

        + dg4 g2 (f1 &* (f0 &* f1))
```

and the Lie bracket table `sr`,

```
> sr:={B[4]=B[1]+6*B[2]};

                    sr := {f0 &* f1 = f0 + 6 f1}
```

The function `reduceLBT` with the P. Hall basis given in the example for the `phb` function on page 40, yields

```
> reduceLBT(x,B,sr);

             2
1/2 dg3 g1  (f0 &* (f0 &* f2)) - dg4 g2 f0 - 6 dg4 g2 f1
```

Implementation Notes

This function basically relies only on `reduceLB` and Maple's `subs` function. The algorithm is rather simple, the procedure repeatedly tries to replace the dependent relations in x using the substitution relations in *lbt* until there are no more dependent terms (brackets) in x. At every substitution iteration, the function `reduceLB` is called to simplify the expression.

6.14 ad

Purpose Implement the adjoint operator (ad_X^n) applied n times to Y, which corresponds to the n-times Lie bracketing of X with Y.

Syntax e:=ad(x,y,n);

Description This procedure implements the adjoint operator (ad_X^n) applied n times to Y and which corresponds to the n-times Lie bracketing of X with Y, i.e.

$$\begin{aligned}(ad_X^0)Y &= (ad_X)^0Y = Y \\ (ad_X)Y &= [X,Y] \\ (ad_X^n)Y &= (ad_X^{n-1})(ad_X)Y = (ad_X^{n-1})[X,Y] \\ &= (ad_X)(ad_X^{n-1})Y = [x,(ad_X^{n-1})Y]\end{aligned}$$

Arguments

x, y	A pair of elements of the Lie algebra or Lie polynomials.
n	The number of times x will be bracketed with y.

Example The calculation of the Lie bracket $[f_1,[f_1,f_0]] = (ad_{f_1}^2)f_0$ can be performed as follows:

```
> ad(f1,f0,2);

                         f1~ &* (f1~ &* f0~)
```

6.15 ead, eadr

Purpose Compute the *Taylor* series expansion of $(e^X)Y(e^{-X}) = (e^{ad_X})Y$.
`eadr`, reduces the Lie brackets in the expression to elements in the PHB and further simplifies it according to the supplied Lie bracket table.

Syntax e:=ead(x,y,n);
e:=eadr(x,y,n,B,lbt);

Description Compute the *Taylor* series expansion of the *exponential formula* (9):

$$\begin{aligned}(e^X)Y(e^{-X}) &= Y + [X,Y] + \frac{1}{2!}[X,[X,Y]] + \frac{1}{3!}[X,[X,[X,Y]]] + \dots \\ &= \sum_{k=0} \frac{1}{k!}(ad_X^k)Y \\ &= (e^{ad_X})Y\end{aligned}$$

up to terms of order n. Details concerning the above formula can be found in [43], (see theorem 2.13.2, p. 104).

`eadr`, reduces the Lie brackets in the expression to elements in the PHB and further simplifies the expression according to the supplied Lie bracket table. If the Lie bracket table is an empty list or set, no additional simplifications are carried out.

Arguments

x, y A pair of elements of the Lie algebra or Lie polynomials.

n Order of the Taylor series expansion (n will be the order of the highest order bracket in the expansion).

`eadr` additionally requires:

B A Philip Hall basis.

lbt A Lie bracket table defining the dependencies between elements of the PHB; see definition in the section for the function `reduceLBT` on page 81. It can be an empty list or set if no dependencies between the brackets in the PHB exist.

Examples The Taylor series expansion for exponential of the operator ad_{f_1} acting on f_0, denoted by $e^{ad_{f_1}} f_0$, may be calculated as:

```
> z:=ead(f1,f0,3);

  z := f0~ + (f1~ &* f0~) + 1/2 (f1~ &* (f1~ &* f0~))

         + 1/6 (f1~ &* (f1~ &* (f1~ &* f0~)))
```

which simplified using the `reduceLB` command results in

```
> reduceLB(z,B);

  f0~ - (f0~ &* f1~) - 1/2 (f1~ &* (f0~ &* f1~))

         - 1/6 (f1~ &* (f1~ &* (f0~ &* f1~)))
```

The above result could also be obtained in single step using the procedure `eadr`, which automatically reduces the brackets to basis brackets in the PHB given in B (see example for `phb` on page 40. If a Lie bracket table is also provided then the expression will be further simplified. Note that in the following example the Lie bracket table is an empty set, thus no additional simplifications take place.

```
>  eadr(f1,f0,3,B,{});

  f0~ - (f0~ &* f1~) - 1/2 (f1~ &* (f0~ &* f1~))

         - 1/6 (f1~ &* (f1~ &* (f0~ &* f1~)))
```

See Also `ad`.

6.16 pead, peadr

Purpose Compute the product of exponentials $\prod_{i=1}^{n} e^{ad_{X_i}} X_{n+1}$.

Syntax e:=pead(n,max_bracket_order,X);
e:=peadr(n,max_bracket_order,X,B,lbt);

Description Computes the product of exponentials $\prod_{i=1}^{n} e^{ad_{X_i}} X_{n+1}$. `peadr`, reduces the Lie brackets in the expression to elements in the PHB, and further simplifies the expression according to the supplied Lie bracket table. If the Lie bracket table is an empty list or set, no additional simplifications are carried out.

Arguments

n Number of products of exponentials.

max_bracket_order
The maximum bracket order in the exponential series expansion of e^{ad_x}; see `ead`, `eadr` on page 84.

X Label or name for the (indeterminate) elements of the Lie algebra, or name of the list containing the names of each element of the Lie algebra basis.
In the case the Lie algebra basis has been constructed in terms of a P. Hall basis, *X* must contain the list of elements in the PHB that are linearly independent. If all the elements in the PHB are regarded as linearly independent, then *X* must be the name of the list that contains the PHB, and this this procedure would be invoked as `pead(n,max_bracket_order,B)` or `peadr(n,max_bracket_order,B,B,lbt)`, where `B` contains the PHB.

`peadr` additionally requires:

B A Philip Hall basis.

lbt A Lie bracket table defining the dependencies between elements of the PHB; see definition in the section for the function `reduceLBT` on page 81. It can be an empty list or set if no dependencies between the brackets in the PHB exist.

Examples In this first example the computation of $\prod_{i=1}^{2} e^{ad_{x_i} x_3}$ is considered. Here x_i are Lie algebra indeterminates, and the Taylor series expansion of the exponential *ad* operator is computed up to brackets of order two.

```
> pead(2,2,x);

  x[3] + g2 x[2] &* x[3] + 1/2 (g2 x[2] &* g2 x[2] &* x[3])

  + (g1 x[1] &* x[3]) + (g1 x[1] &* g2 x[2] &* x[3])

  + 1/2 (g1 x[1] &* (g2 x[2] &* g2 x[2] &* x[3]))

  + 1/2 (g1 x[1] &* (g1 x[1] &* x[3]))

  + 1/2 (g1 x[1] &* (g1 x[1] &* g2 x[2] &* x[3]))

  + 1/4 (g1 x[1] &* (g1 x[1] &* (g2 x[2] &* g2 x[2] &* x[3])))
```

For the next example consider the P. Hall basis `B` of the example for the function `phb` on page 40. The computation of $\prod_{i=1}^{2} e^{ad_{B_i} B_3}$ yields,

```
> pead(2,2,B);

  f2~ + (g2 f1~ &* f2~) + 1/2 (g2 f1~ &* (g2 f1~ &* f2~))

  + (g1 f0~ &* f2~) + (g1 f0~ &* (g2 f1~ &* f2~))

  + 1/2 (g1 f0~ &* (g2 f1~ &* (g2 f1~ &* f2~)))

  + 1/2 (g1 f0~ &* (g1 f0~ &* f2~))

  + 1/2 (g1 f0~ &* (g1 f0~ &* (g2 f1~ &* f2~)))

  + 1/4 (g1 f0~ &* (g1 f0~ &* (g2 f1~ &* (g2 f1~ &* f2~))))
```

Note that the above computation does not yields an expression whose elements are in the P. Hall, if this additional simplification is required then using `peadr` we obtain:

```
> peadr(2,2,B,B,{});

                                     2
  f2~ + g2 (f1~ &* f2~) + 1/2 g2  (f1~ &* (f1~ &* f2~))

                                        2
          + g1 (f0~ &* f2~) + 1/2 g1  (f0~ &* (f0~ &* f2~))
```

Note that in the above example the degree of the highest order brackets is three, unlike when `pead` was used, where the highest order brackets are of degree five. This is due to the fact that `peadr` calls `reduceLBT` to make simplifications, thus brackets in the series expansion whose order is higher than *max_bracket_order* are eliminated.

See Also `ad`, `ead`, `eadr`.

6.17 wne, wner

Purpose Compute the right-hand side of the Wei-Norman formula: $\sum_{k=1}^{r} X_k u_k = \dot{S}(t)S^{-1}(t)$, where $S(t) = \prod_{k=1}^{r} e^{g_k(t)X_k}$ and $\dot{S}(t) = \frac{dS(t)}{dt}$.

Syntax e:=wne(r,max_bracket_order,X);
e:=wner(r,max_bracket_order,X,B,lbt);

Description Computes the right-hand side of the Wei-Norman formula:

$$\sum_{k=1}^{r} X_k u_k = \dot{S}(t)S^{-1}(t) \tag{40}$$

where u_k are scalar time-dependent functions, i.e. $u_k : t \to \mathbb{R}$, X_k are indeterminate elements independent of time that define a basis for an arbitrary Lie algebra, and with

$$S(t) = \prod_{k=1}^{r} e^{g_k(t)X_k} \tag{41}$$

and

$$\dot{S}(t) = \frac{dS(t)}{dt} = \sum_{i=1}^{r} \dot{g}_i(t) \prod_{j=1}^{i-1} e^{g_j X_j} X_i \prod_{j=i}^{r} e^{g_j X_j} \tag{42}$$

The Lie algebra generated by the X_k is required to be finite dimensional.

The above expressions for $S(t)$ and $\dot{S}(t)$, together with the exponential formula given in equation (9) allow to express the right-hand side of the Wei-Norman equation as:

$$\dot{S}(t)S^{-1}(t) = \sum_{i=1}^{r} \dot{g}_i(t) \prod_{j=1}^{i-1} e^{g_j ad_{X_j}} X_i \tag{43}$$

Hence, this procedure can be implemented in a simple way in terms of the functions `ead`, `eadr` and `pead`, `peadr`.

`wner`, reduces the Lie brackets in the expression to elements in the PHB, and further simplifies the expression according to the supplied Lie bracket table. If the Lie bracket table is an empty list or set, no additional simplifications are carried out.

Arguments

r — Dimension of the Lie algebra basis. In particular, if a PHB is considered as a general basis, then r corresponds to the number of Lie brackets in the PHB which are independent.

max_bracket_order — The maximum bracket order in the exponential series expansion of e^{ad_x}; see `ead`, `eadr` on page 84.

X — Label or name for the (indeterminate) elements of the Lie algebra, or name of the list containing the names of each element of the Lie algebra.
In the case the Lie algebra basis has been constructed in terms of a P. Hall basis, X must contain the list of elements in the PHB that are linearly independent. If all the elements in the PHB are regarded as linearly independent, then X must be the name of the list that contains the PHB, and this this procedure would be invoked as `wne(r,max_bracket_order,B)` or `wner(r,max_bracket_order,B,B,lbt)`, where `B` contains the PHB.

`wner` additionally requires:

B — A Philip Hall basis.

lbt — A Lie bracket table defining the dependencies between elements of the PHB; see definition in the section for the function `reduceLBT` on page 81. It can be an empty list or set if no dependencies between the brackets in the PHB exist.

Examples The examples below consider the P. Hall basis B for a nilpotent Lie algebra of degree four generated by three indeterminates f0, f1 and f2, obtained in the example for the function phb on page 40. The maximum bracket order in the series expansion of the exponential that will be considered in the next examples is equal to three (i.e. the degree of nilpotency minus one, so that the resulting brackets are always contained in the PHB).
In the following two examples assume that the actual basis for the Lie algebra is of dimension 4. Notice that in the first example has an empty Lie bracket table, i.e. all the elements in B are independent.

```
> w2r:=wner(r,max_bracket_order,B,B,{});

  w2r := dg1~ f0~ + dg2~ f1~ + (dg2~ g1~ + dg4~) (f0~ &* f1~)

                       2
       + (1/2 dg2~ g1~  + dg4~ g1~) (f0~ &* (f0~ &* f1~)) +

                    3                2
      (1/6 dg2~ g1~  + 1/2 dg4~ g1~ )

      (f0~ &* (f0~ &* (f0~ &* f1~))) + dg3~ f2~

       + dg3~ g2~ (f1~ &* f2~)

                      2
       + 1/2 dg3~ g2~  (f1~ &* (f1~ &* f2~))

                      3
       + 1/6 dg3~ g2~  (f1~ &* (f1~ &* (f1~ &* f2~)))

       + dg3~ g1~ (f0~ &* f2~)

                      2
       + 1/2 dg3~ g1~  (f0~ &* (f0~ &* f2~))

                      3
       + 1/6 dg3~ g1~  (f0~ &* (f0~ &* (f0~ &* f2~)))

       + dg4~ g3~ (f2~ &* (f0~ &* f1~))

                      2
       + 1/2 dg4~ g3~  (f2~ &* (f2~ &* (f0~ &* f1~)))

       + dg4~ g2~ (f1~ &* (f0~ &* f1~))

                      2
       + 1/2 dg4~ g2~  (f1~ &* (f1~ &* (f0~ &* f1~)))

       + dg4~ g1~ g2~ (f1~ &* (f0~ &* (f0~ &* f1~)))
```

A similar calculation to the previous one, but with the assumption that the bracket $B_6 = [f_1, f_2]$ is zero, yields:

```
> w5r:=wner(r,max_bracket_order,B,B,{B[6]=0});

  w5r := dg1~ f0~ + dg2~ f1~ + (dg2~ g1~ + dg4~) (f0~ &* f1~)

                               2
         + (1/2 dg2~ g1~  + dg4~ g1~) (f0~ &* (f0~ &* f1~)) +

                          3                 2
        (1/6 dg2~ g1~  + 1/2 dg4~ g1~ )

        (f0~ &* (f0~ &* (f0~ &* f1~))) + dg3~ f2~

         + dg3~ g1~ (f0~ &* f2~)

                             2
         + 1/2 dg3~ g1~  (f0~ &* (f0~ &* f2~))

                             3
         + 1/6 dg3~ g1~  (f0~ &* (f0~ &* (f0~ &* f2~)))

         + dg4~ g3~ (f2~ &* (f0~ &* f1~))

                             2
         + 1/2 dg4~ g3~  (f2~ &* (f2~ &* (f0~ &* f1~)))

         + dg4~ g2~ (f1~ &* (f0~ &* f1~))

                             2
         + 1/2 dg4~ g2~  (f1~ &* (f1~ &* (f0~ &* f1~)))

         + dg4~ g1~ g2~ (f1~ &* (f0~ &* (f0~ &* f1~)))
```

As expected, the difference `w2r-w5r` shown below contains only terms of the bracket $B_6 = [f_1, f_2]$ which was assumed to be zero in the simplification of `w5r`.

```
                             3
         + 1/6 dg3~ g2~  (f1~ &* (f1~ &* (f1~ &* f2~)))

                             2
         + 1/2 dg3~ g2~  (f1~ &* (f1~ &* f2~))

         + dg3~ g2~ (f1~ &* f2~)
```

Note that an actual computation in Maple of 'w2r-w5r;' returns an expression that contains terms, such as:

```
(-dg1~ + dg1~) f0 +
+ (-dg2~ g1~ - dg4~ + dg2~ g1~ + dg4~) (f0~ &* f1~) + ...
```

that are not simplified to zero because each time wner is invoked, different variables dg1, dg2, etc., are created at each time (i.e. the variables have the same name, but they do not correspond to the same instance of a unique variable in the Maple space, in fact they are instances of different variables). Although this does not seem so far to cause significant problems, its is worth to mention that if one wishes to make comparisons between expressions it would probably be convenient to modify the routines wne and wner so that they declare variables g and dg in the global space as unique instances.

See Also ead, eadr, pead, peadr.

References See [32] and references therein for an explanation on the derivation of the Wei-Norman equations.

6.18 wnde

Purpose Construct the differential equation for the logarithmic coordinates g_k of the Wei-Norman formula, see `wne`, `wner`.

Syntax e:=wnde(x,r,max_bracket_order,B,lbt);

Description Constructs the differential equation for the logarithmic coordinates g_k of the Wei-Norman formula. This function equates the coefficients of the X_k in the left-hand side of the Wei-Norman equation 40 to the corresponding coefficients of X_k in the right-hand side 43, yielding a set of equations of the form:

$$\begin{aligned} u_1 &= F_1(g_1, \ldots, g_r)\dot{g}_1 \\ u_2 &= F_2(g_1, \ldots, g_r)\dot{g}_2 \\ &\vdots \\ u_r &= F_r(g_1, \ldots, g_r)\dot{g}_r \end{aligned}$$

or equivalently in matrix form as

$$u = F(g)\dot{g} \tag{44}$$

where $u, g, \dot{g} \in \mathbb{R}^r$, and $F(g) : \mathbb{R}^r \rightarrow \mathbb{R}^{r\times r}$. This procedure returns in its first argument the matrix $F(g)$ and the set of equations of the form $u_k = F_k(g_1, \ldots, g_r)\dot{g}_k$, $k = 1, \ldots, r$ for as a second output element. In order to equate the coefficients on each side of the Wei-Norman equation, this function requires the right-hand side as calculated with the procedure `wne`.

Arguments

x The right-hand side of the Wei-Norman equations as computed with `wne`, `wner` (described on page 89.

r Dimension of the Lie algebra basis. In particular, if a PHB is considered as a general basis, then r corresponds to the number of Lie brackets in the PHB which are independent.

max_bracket_order
The maximum bracket order in the exponential series expansion of e^{ad_x}; see `ead`, `eadr` on page 84. *NOTE: max_bracket_order is not currently used. It would be necessary if `wnde` would directly invoke `wner`, instead of passing the result of `wner` as argument x.*

B A Philip Hall basis.

lbt A Lie bracket table defining the dependencies between elements of the PHB; see definition in the section for the function `reduceLBT` on page 81. It can be an empty list or set if no dependencies between the brackets in the PHB exist.

Examples Consider the P. Hall basis, B, given in the example for the function phb on page 40. Additionally, suppose that $B_6 = [f_1, f_2] = 0$, then the Lie algebra can be expressed in terms of the following 14-dimensional basis of independent Lie brackets, in terms of which w5r is expressed (see example for the function wne, wner on page 89):

```
BB := [f0~, f1~, f0~ &* f1~, f0~ &* (f0~ &* f1~),

        f0~ &* (f0~ &* (f0~ &* f1~)), f2~, f0~ &* f2~,

        f0~ &* (f0~ &* f2~), f0~ &* (f0~ &* (f0~ &* f2~)),

        f2~ &* (f0~ &* f1~), f2~ &* (f2~ &* (f0~ &* f1~)),

        f1~ &* (f0~ &* f1~), f1~ &* (f1~ &* (f0~ &* f1~)),

        f1~ &* (f0~ &* (f0~ &* f1~))]
```

The Wei-Norman equations can now be calculated using wnde, note that *max_bracket_order* is an empty list, since the current version of wnde does not of this argument. Notice that *lbt* is also an empty list, since in this case we are passing directly the basis of the Lie algebra BB in the argument for the PHB.

```
> lc:=wnde(w5r,14,{},BB,{});

                                       3
  lq := Fg, {u[9] = 1/6 dg3~ g1~ , u[10] = dg4~ g3~, u[1] = dg1~,

                                                                         2
        u[2] = dg2~, u[3] = dg2~ g1~ + dg4~, u[11] = 1/2 dg4~ g3~ ,

                                                 2
        u[12] = dg4~ g2~, u[4] = 1/2 dg2~ g1~  + dg4~ g1~,

                                 2
        u[13] = 1/2 dg4~ g2~ , u[6] = dg3~,

                               3                2
        u[5] = 1/6 dg2~ g1~  + 1/2 dg4~ g1~ , u[7] = dg3~ g1~,

                                                       2
        u[14] = dg4~ g1~ g2~, u[8] = 1/2 dg3~ g1~ }
```

The matrix $F(g)$ of logarithmic coordinates g can be obtained from the first element returned by `wnde`.

```
> eval(lc[1]);
    [1      0        0        0     0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [0      1        0        0     0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [0     g1~       0        1     0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [           2                                               ]
    [0  1/2 g1~      0       g1~    0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [           3                2                              ]
    [0  1/6 g1~      0   1/2 g1~    0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [0      0        1        0     0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [0      0       g1~       0     0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [                    2                                      ]
    [0      0    1/2 g1~      0     0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [                    3                                      ]
    [0      0    1/6 g1~      0     0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [0      0        0       g3~    0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [                            2                              ]
    [0      0        0   1/2 g3~    0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [                                                           ]
    [0      0        0       g2~    0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [                            2                              ]
    [0      0        0   1/2 g2~    0  0  0  0  0  0  0  0  0  0]
    [                                                           ]
    [0      0        0   g1~ g2~    0  0  0  0  0  0  0  0  0  0]
```

Practical application examples involving the derivation of the logarithmic coordinates for an underactuated rigid body in space can be found in the directory `../ltp/dev`, in the files `weinorman_rb1.ws` and `weinorman_rbfull.mws`.

Notes Care should be put when passing the P. Hall basis, the Lie bracket table to this function, and the right-hand side of the Wei-Norman equation. It could be the case that the number of brackets in each argument are not consistent. This function attempts to construct the list containing the actual basis for the Lie algebra from the P. Hall basis and the Lie bracket table. It also tries to detect situation of argument inconsistency, however the checking procedure is rather simple still and it may not correctly identify all possible errors.
To avoid problems perhaps the best way is to provide the function with the Lie algebra basis and pass it in the argument for the P. Hall basis and pass an empty list or set as argument for the Lie bracket table. The disadvantage of this approach is the possible calculation involved in the determination of a suitable Lie algebra basis prior to invoking `wnde`.

See Also `wne`, `wner`.

References See [32] and references therein for an explanation on the derivation of the Wei-Norman equations.

7 Topics for Further Improvement

7.1 Generation of a k-th order CBH formula (cbhexpr)

Implementing a procedure that could calculate the terms in the in the exponent of the CBH formula up to an order k specified by the user. At this point it is not clear whether this routine would be strictly necessary, but it can be expected it would for high dimensional systems. A possible approach to solve this problem would be to use the Dynkin-Campbell-Baker-Hausdorff formula presented in [32].

7.2 Functions for setting-up and solving logarithmic equations automatically

In the case when the Lie algebra is non-nilpotent an approximation, such as truncation of the series expansion of the exponential operator will be necessary in any attempt to determine an explicit expression for the Wei-Norman equations. The automatic identification of the convergence of a series to a given function is in general a very difficult problem and one of the main obstacles to the implementation of an exact explicit solution.

If the system is non-nilpotent but has a finite-dimensional Lie algebra and allows a representation as a system on a matrix Lie group, then there exists the possibility to obtain explicit expressions of the Wei-Norman equations using the approach proposed in [2].

It would also be convenient to develop concurrently some routines for the nilpotentization of the system by state feedback and state-space transformations, since working with the nilpotent version of the system would allow to obtain an explicit Wei-Norman formula.

7.3 Automatic controller design/synthesis functions

Unless a better (simpler and smoother) nonlinear controller can be devised, one of the ultimate goals of the package should be to compute the equations for a suitable implementation of a Lie theory based controller for general nonlinear control systems described in terms of ordinary differential equations.

A Implementation Notes

LTP was implemented in Maple 6. Several Computer Algebra Systems (CAS) were evaluated, each showing drawbacks and strengths over Maple. Comparisons and additional references to CAS can be found in [46]. Unlike some of the other available CAS, Maple allows the definition of new operators (through the **define** command), and to make some assumptions about the properties of the operator.

Unlike other symbolic packages that are Lisp based and have inherited some of Lisp's disadvantages, Maple has a syntax similar to that of the C language. Maple also supports both the functional (or applicative) and the imperative (or procedural) programming styles, thus the routines can be implemented and tested faster than with a purely imperative language like C, which furthermore lacks of the appropriate data abstractions.

Functional programming languages have some clear advantages over purely procedural languages. This advantages can be summarized in:

- Problems can be modeled in a simpler and more natural way since functional programming facilitates the expression of concepts and structures at a higher level of abstraction. This is particularly true for problems involving symbolic manipulation.
- Recursions (a function calling itself) can be handled in a relatively simple way.
- The ability to express the problem in terms of recursions makes the code easier to understand.
- A functional programming language that avoids side effects is more efficient, and easier to optimize.

Imperative programming languages have some strengths too, however it is not the scope of this document to compare programming styles. It suffices to say that, ideally a good programming language should not make it difficult for a programmer to write computationally efficient code while maintaining an adequate level of abstraction that makes the problem easily expressible in terms of the language abstractions. In this sense, Maple collects appealing aspects of each programming style.

It is unlikely that traditional imperative languages such as C and Fortran would completely loose the popularity they enjoy, even if the well designed imperative language Java gained acceptance very fast. On the other hand, Lisp and other modern functional languages like ML or Haskell also have their supporters. Whether to choose a purely imperative or a functional language

depends mostly on the type of problem at hand and how well a particular language allows a rapid implementation of the solution. Plenty of references and information on computer languages can be found at [51]. A place to start for references on functional programming is [52].

A.0.1 Highlights of some Implementation Issues

- Whenever possible, operations have been implemented using a functional style that avoids side effects, rather than a traditional imperative style.
- Several routines of the package are recursive. Their flow chart description indicates recursion points (points at which the routine calls itself) by dashed paths joining the recursion point to the routine's entry point.

A.0.2 Recommendations for Improvement

Even if Maple is a good package there are several reasons for concern (see also Maple's Users Group on the Internet):

- Bugs were found in Maple's routines for defining new operators and for making assumptions about them.
- Support from the Maple developers is very poor and leaves a lot to be desired. This impression is shared by many users and can be verified by taking a look at the Maple Users Group.
- Maple bugs are poorly documented, and developers seem not to have any systematic approach to fixing them.
- Bugs seem to be history-dependent and what works on one release, might not on the next.

New implementations of LTP could rely on GiNaC [53], which is an open framework for symbolic computations recently developed in the C++ programming language. GiNaC extends the well established and standardized C++ language by some fundamental symbolic capabilities, and thus constitutes a convenient alternative for large-scale projects in which both numerical and symbolic calculations are required.

Another option would be to employ Reduce [54], which is a CAS with more than 30 years of development. Reduce also has a basic function called `operator` function that allows the declaration of new user-defined operators (this function is equivalent to `define` in Maple). Parts in an expression can be accessed in Reduce by means of the `parts` function (`op` function in Maple). This features

would allow to easily implement LTP using Reduce. Like GiNaC, Reduce is open source, however it is based on Lisp, which might not be so appealing as C++.

There are other CAS, however they do not provide adequate facilities to declare new operators, which is a key element in the creation of a software package like this. For this and other reasons that would be too extense to discuss here, other CAS are not advisable alternatives. The reader is referred to [55] and [56] for further information about CAS and related information.

References

[1] A. Agrachev and R. Gamkrelidze, *Chronological algebras and nonstationary vector fields*, Journal Soviet Math., 17 (1979), pp. 1650–1675.

[2] C. Altafini, *Explicit Wei-Norman formulae for matrix Lie groups*, submitted to Systems Control Lett., Elsevier Science, Feb. 2002, available at `http://www.sissa.it/~altafini`.

[3] J. G. F. Belinfante and B. Kolman, *A Survey of Lie Groups and Lie Algebras with Applications and Computational Methods.* Society for Industrial and Applied Mathematics, 1972.

[4] N. Bourbaki, *Lie Groups and Lie Algebras, Part I: Chapters 1–3*, Hermann, Great Britain, 1975.

[5] R. W. Brockett and J. M. C. Clark, *The geometry for the conditional density functions*, in Analysis and Optimization of Stochastic Systems, O. Jacobs et al., eds., Academic Press, New York, 1980, pp. 299–310.

[6] B. Champagne, W. Hereman and P. Winternitz, *The computer calculation of Lie point symmetries of large systems of differential equations*, Comput. Phys. Comm., 66 (1991), pp. 319–340.

[7] M. Cohen de Lara, *Finite-dimensional filters. Part I: The Wei-Norman technique*, SIAM J. Control Optim., 35 (1997), pp. 980–1001.

[8] V.T. Coppola and N.H. McClamroch, *Spacecraft attitude control*, in Control System Applications, W. S. Levine, ed., CRC Press, 2000.

[9] M. Dahleh, A. Peirce, H.A. Rabitz and V. Ramakrishna, *Control of molecular motion*, Proc. of the IEEE, 84 (1996), pp. 7–15.

[10] M. H. A. Davis and S. I. Marcus, *An introduction to nonlinear filtering*, in Stochastic Systems: The Mathematics of Filtering and Identification and Applications, M. Hazewinkel and J. Willems, eds., D. Reidel, Dordrecht, Netherlands, 1981, pp. 55–75.

[11] P. Feinsilver and R. Schott, *Algebraic Structures and Operator Calculus*, Vol. III. Mathematics and Its Applications, Kluwer Academic Publishers, 1996.

[12] M. Fliess, *Réalisation locale des systèmes non linéaires, algèbres de Lie filtrées transitives et séries génératrices non commutatives*, Inventiones Mathematicae, 71 (1983), pp. 521–537.

[13] R. Gilmore, *Lie Groups, Lie Algebras, and Some of Their Applications*, John Wiley & Sons, Inc., 1974.

[14] M. Hall, *The Theory of Groups*, Macmillan, 1959.

[15] H. Hermes, *Nilpotent and high-order approximations of vector field systems*, SIAM Rev., 33 (1991), pp. 238–264.

[16] A. Isidori, *Nonlinear Control Systems — An Introduction.* 2nd ed., Springer-Verlag Berlin, 1989.

[17] M. Kawski and H. J.Sussmann, *Noncommutative power series and formal Lie-algebraic techniques in nonlinear control theory*, in Operators, Systems, and Linear Algebra, U. Helmke, D. Prätzel-Wolters and E. Zerz, eds., Teubner, (1997), pp.111–128.

[18] M. Kawski, *The combinatorics of nonlinear controllability and noncommuting flows*, Abdus Salam ICTP Lecture Notes series, 8 (2002), pp. 223–312.

[19] P. S. Krishnaprasad, S. I. Marcus, M. Hazewinkel. Current algebras and the identification problem. *Stochastics*, v. 11, 1983, pp. 65-101.

[20] G. Lafferriere and H. J. Sussmann, *A differential geometric approach to motion planning*, in Nonholonomic Motion Planning, Z. Li, and J. F. Canny, eds., Kluwer Academic Publishers, 1993, pp. 235–270.

[21] M.A.A. van Leeuwen. LiE, A software package for Lie group computations. *Euromath Bulletin 1*, n. 2, 1994.

[22] W. Magnus, *On the exponential solution of differential equations for a linear operator*, Commun. Pure Appl. Math., VII (1954), pp. 649–673.

[23] S. Marcus, *Algebraic and geometric methods in nonlinear filtering*, SIAM J. Control Optim., 22 (1984), pp. 817–844.

[24] G. Melançon and C. Reutenauer, *Lyndon words, free algebras and shuffles*, Canadian J. Math., XLI (1989), pp. 577–591.

[25] G. Melançon and C. Reutenauer, *Combinatorics of Hall trees and Hall words*, J. Comb. Th., Ser. A 59 (1992), pp. 285–299.

[26] H. Michalska and M. Torres-Torriti, *A geometric approach to feedback stabilization of nonlinear systems with drift*, Systems Control Lett., 50 (2003), pp. 303-318.

[27] R.M. Murray, Z. Li and S.S. Sastry, *A Mathematical Introduction to Robotic Manipulation*, CRC Press, 1994.

[28] D. Ocone, *Finite dimensional estimation algebras and nonlinear filtering*, in *Stochastic Systems: The Mathematics of Filtering and Identification and Applications*, M. Hazewinkel and J.C. Willems, eds., Reidel, Dordrecht, 1981.

[29] R. Palais, *Global Formulation of the Lie Theory of Transformation Groups*, v. 22, Mem. Amer. Math. Soc., AMS, 1957.

[30] C. Reutenauer, *Free Lie algebras*, Clarendon Press, 1993.

[31] E. Rocha, *On computation of the logarithm of the Chen-Fliess series for nonlinear systems*, Lecture Notes in Control and Information Sciences – Vol. 281: Nonlinear and Adaptive Control: NCN4 2001, Springer-Verlag Heidelberg, (2003), pp. 317–326.

[32] S.S. Sastry, *Nonlinear Systems: Analysis, Stability and Control*, Springer-Verlag New York, Inc., 1999.

[33] D. H. Sattinger and O. L. Weaver, *Lie Groups and Algebras with Applications to Physics, Geometry and Mechanics*, Springer-Verlag New York, Inc., 1986.

[34] M.-P. Schützenberger, *Sur une propriété combinatoire des algèbres de Lie libres pouvant être utilisée dans un problème de mathématiques appliquées*, Séminaire P. Dubreil, Faculté des Sciences, Paris, 1958.

[35] J. M. Selig, *Geometrical Methods in Robotics*, Springer-Verlag New York, Inc., 1996.

[36] J.-P. Serre, *Lie Algebras and Lie groups*, W. A. Benjamin, New York, 1965.

[37] A. I. Shirshov, *Bases of free Lie algebras*, Algebra i Logika Sém., 1 (1962), pp. 14-19.

[38] W.-H. Steeb, *Continuous Symmetries, Lie Algebras, Differential Equations and Computer Algebra*, World Scientific Co., 1996.

[39] R.S. Strichartz, *The Campbell-Baker-Hausdorff-Dynkin formula and solutions of differential equations*, J. Funct. Anal., 72 (1987), pp. 320–345.

[40] H. J. Sussmann, *Lie brackets and local controllability: a sufficient condition for scalar-input systems*, SIAM J. Control Optim., 21 (1983), pp. 686–713.

[41] H. J. Sussmann, *A product expansion for the Chen series*, in Theory and Applications of Nonlinear Control Systems, C. Byrnes and A. Lindquist, eds., Elsevier Science Publishers B. V. (North Holland), (1986), pp.323-335.

[42] H. J. Sussmann, *A general theorem on local controllability*, SIAM J. Control Optim., 25 (1987), pp. 158–194.

[43] V. S. Varadarajan, *Lie Groups, Lie Algebras, and their Representations*, Springer-Verlag New York, Inc., 1984.

[44] X. G. Viennot, *Algébres de Lie Libres et Monoïdes Libres*, Lecture Notes in Mathematics, Vol. 691, Springer, Berlin, 1978.

[45] J. Wei and E. Norman, *On global representations of the solutions of linear differential equations as products of exponentials*, Proc. Amer. Math. Soc., 15 (1964), pp. 327–334.

[46] *Computer Algebra Systems - A Practical Guide*. M. J. Wester Editor, John Wiley & Sons Ltd., 1999.

Books on Maple Programming

[47] A. Heck, *Introduction to Maple*, Second Edition. Springer-Verlag New York, Inc., 1996.

[48] B. W. Char, *et al.*, *Maple V Library Reference Manual*. Springer-Verlag, 1991

[49] B. W. Char, *et al.*, *Maple V Language Reference Manual*. Springer-Verlag, 1991

[50] R. A. Nicolaides and N. Walkington, *Maple a Comprehensive Introduction*. Cambridge University Press, 1996.

Resources on Internet

[51] Computer Languages: `http://dmoz.org/Computers/Programming/Languages/`

[52] Functional Programming FAQ: `http://www.cs.nott.ac.uk/~gmh/faq.html`

[53] GiNaC (GNU is Not a CAS): `http://www.ginac.de/`

[54] Reduce CAS: `http://www.uni-koeln.de/REDUCE/`

[55] Computer Algebra Information Network (Europe): `http://www.mupad.de/CAIN/`

[56] Symbolic Mathematical Computation Information Center (US): `http://www.SymbolicNet.org/`