Symbolic Software Tools for Flatness of Linear Systems with Delays and Nonlinear Systems

Gregor Günther Verhoeven

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Universität der Bundeswehr München zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften
(Dr.-Ing.)

genehmigten Dissertation.

Gutachter/Gutachterin:

1. Prof. Dr. rer. nat. habil. Claus Hillermeier
2. Prof. Dr. Jean Lévine

Die Dissertation wurde am 18.09.2015 bei der Universität der Bundeswehr München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 19.04.2016 angenommen. Die mündliche Prüfung fand am 03.06.2016 statt.

Acknowledgment

This doctoral thesis was written during my work as an external doctoral student for the Department for Measurement and Automation of the University of the German Armed Forces in Munich.

First of all, I would like to thank Prof. Dr. rer. nat. habil. Claus Hillermeier for making this possible and providing my posting. He introduced me to control theory during my studies and inspired me to lay the focus of my scientific work on this field.

Furthermore, I would like to thank Dr.-Ing. habil. Felix Antritter for the long lasting scientific collaboration on the topic of flatness of linear and nonlinear systems. He initiated this thesis, supervised it during his time as professor at the University of the German Armed Forces and supported me greatly in order to finish this thesis, even after he had left the University of the German Armed Forces.

Moreover, I would like to thank Prof. Dr. Jean Lévine for supporting me and agreeing to be the second doctoral supervisor of my thesis. This thesis would not have been possible without his scientific research in flatness determination.

Further thanks go to the staff of the Chair for Automation and Control for the great scientific cooperation.

I would also like to thank Katharina Verhoeven and Dipl.-Ing. Thomas Krönert for proofreading this thesis.

Finally, I would like to thank my family for their support and encouragement.

Netphen, June 2016 Gregor Günther Verhoeven

Kurzfassung / Abstract

Flachheit als Eigenschaft linearer und nichtlinearer Systeme ermöglicht eine vergleichsweise einfache Konstruktion von Steuerungen und Reglern zur Lösung des Trajektorienfolgeproblems. Dieser Ansatz erfordert allerdings die Berechnung von flachen Ausgängen für das jeweilige System. Je nach Komplexität des Systems kann diese Berechnung sehr aufwändig sein. Im Rahmen dieser Arbeit wurden zwei Toolboxen für das Computer-Algebra-System Maple entwickelt, die die Berechnung von flachen Ausgängen für lineare Systeme (mit und ohne Totzeitglieder) und nichtlineare Systeme ermöglichen.
Im Falle linearer Systeme basieren die implementierten Datentypen und Methoden auf dem von F. Antritter, F. Cazaurang, J. Lévine und J. Middeke entwickelten Algorithmus, der einen Ansatz mit Schiefpolynommatrizen verwendet. Im Falle nichtlinearer Systeme basiert die implementierte Toolbox auf dem von J. Lévine entwickelten Algorithmus, der einen differentialgeometrischen Ansatz verfolgt.

Flatness as a system property of linear and nonlinear systems allows a relatively simple construction of feed-forward controllers and feedback controllers in order to solve the tracking problem. However, this approach makes the computation of flat outputs of the respective system necessary. Depending on the complexity of the system these computations can be very elaborate. In the context of this thesis, two toolboxes for the computer algebra system Maple were developed which allow the computation of flat outputs for linear systems (with and without delays) and for nonlinear systems.
In case of linear systems, the implemented data types and methods are based on the algorithm developed by F. Antritter, F. Cazaurang, J. Lévine and J. Middeke, which uses an approach with skew polynomial matrices. In case of nonlinear systems, the implemented toolbox is based on the algorithm developed by J. Lévine, which makes use of a differential geometric approach.

Contents

Chapter 1

Introduction

The concept of differential flatness as a system property for linear and nonlinear systems was introduced in [13] and [27]. This system property was extended to describe linear systems with delays as well, for instance in [2]. Roughly speaking, if a control system is flat, it is possible to describe its inputs and states by a (fictional) flat output (and its derivatives) of the system, while this flat output can be expressed by the states and inputs (and their derivatives). In case of linear systems, this property is equivalent to controllability.
If found, such a flat output will offer a relatively simple way to construct a flatness-based controller for the system. For linear systems with and without delays, there exist many approaches for constructing a flat output (for instance in [2, 4, 10, 12, 23, 31]). In case of nonlinear multi-input systems, there exist necessary and sufficient conditions (see [21, 22]) which allow an almost automated flatness determination of nonlinear systems. Nevertheless, constructing a flat output is very difficult, due to the computational complexity of this problem.

Since the computations which have to be made in order to determine whether a control system is flat or not are very elaborate, a toolbox for automated evaluation seems inevitable. At the moment, there exist a few toolboxes which are partially able to handle the computations for flatness determination.
In [4] a *Maple*-toolbox for the flatness determination of linear systems with delays was presented. It was based on the OreTools-package of *Maple*.
In [12] a *Maple*-toolbox was presented which offers features to determine controllability and parametrizability of linear control systems, as well as evaluate whether a linear system is π-flat.
In [33] a toolbox for the flatness determination of nonlinear systems was developed. The purpose of that toolbox was to show the feasibility of developing a toolbox in *Maple* which is capable of computing the flat output of nonlinear systems. But that toolbox was neither user-friendly nor laid its focus on a high computational performance.

Therefore, no existing toolbox is able to handle the thorough flatness determination in case of linear systems with delays and nonlinear systems while providing a high computational performance. In this thesis, two toolboxes will be developed which are capable of computing the flat output (in case of nonlinear systems) resp. the defining operators and the polynomial π (in case of linear systems with delays). Therefore, they can be used to explore the current approaches for flatness determination (for instance of [22]) more deeply.

The development of the two toolboxes follows three main issues: a high computational performance, a high maintainability and the ability to be reused as a framework for future algorithms and toolboxes.

At first, the used computer algebra system *Maple* will briefly be introduced in chapter 2.
Afterwards, in chapter 3, the mathematical basics of flatness determination will be explained. We will recall the required mathematical framework and the specific conditions for differential and π-flatness. We will also define algorithms which can be used in order to evaluate whether these conditions are met for a certain system.
This will allow us to set up the requirements which have to be met by the toolboxes. For this purpose, we will analyze the desired mathematical structures and how they can be represented by using suitable data types in *Maple* in chapters 4 and 5. At the end of these chapters, we will define data types to execute the flatness determination in case of linear systems (with and without delays) and nonlinear systems.
Furthermore, in chapter 6, we are going to look at some specific technical issues of the implementation of the two toolboxes.
In chapter 7, we take a look at the two *Maple*-toolboxes which were developed in the context of this thesis. We will describe all features in detail and illustrate some of them by using small examples.
Afterwards, in chapter 8, we will demonstrate the power of the toolboxes by analyzing two linear and two nonlinear systems and computing the defining matrices resp. the flat outputs of these systems.
Finally, we will take a look at possible future works and what might be the next developments in future versions of the toolboxes.

Chapter 2

About the Used Computer Algebra System *Maple*

In this chapter the features and functionality of the used computer algebra system (CAS) will be described briefly. Furthermore, additional software of the used development environment will be listed. At the end of the chapter, a short overview of the implemented toolboxes will be given.

2.1 Features of *Maple*

The computer algebra system *Maple* of the *Waterloo Maple Inc.* company [34] was used to develop the toolboxes which are introduced in this thesis. The used version was version 15.01[1].
To implement the algebraic transformations and computations of sections 3.2 and 3.3, the computer algebra system has to meet several requirements:

i) ability to compute numerical and symbolic[2]

ii) ability to solve partial differential equations

iii) providing a programming language to implement and develop custom algorithms and toolboxes

Maple meets all of these requirements. Most notably, *Maple* offers a very capable solver for symbolic partial differential equations. In addition, it comes with several useful libraries which extend the functionality of *Maple* massively. In this thesis, the following libraries (in *Maple* also called *packages*) were used:

[1] *Maple* undergoes continuous development, thus several methods of the *Maple* framework (like the including libraries) can be enhanced or completely changed over the versions. We experienced this in case of the integrated solver for partial differential equations. Because of this, it may be possible that the toolboxes do not work properly or work only with limited functionality under other *Maple*-versions.

[2] *Symbolic computation* means that the software is able to perform calculations with not specified variables and functions (such as: $f(x_1(t), x_2(t))$. This is also called *algebraic computation*.

Toolbox	Description
LinearAlgebra	This library offers fundamental functionalities for calculations with matrices and vectors, like computing the kernel of a matrix.
ListTools	A collection of methods for manipulating the *Maple* data type list, like searching in lists for a certain element.
Maplets	The maplets represent the components to build a GUI in *Maple*. Similar to the JComponents of *Java*, the maplets match a container pattern which allows to form the user interface as needed.
PDETools	The library PDETools contains methods for handling symbolic partial differential equations.
RandomTools	This library allows to create generators for random numbers.
StringTools	This library offers methods for manipulating strings.

2.2 Used Development Environment

The computer algebra system *Maple* is delivered with its own development environment, the *Maple User Interface* [26]. This environment can be used to develop custom *Maple* programs, which can be interpreted using the *Maple Computation Engine*. However, this development environment has several disadvantages and restrictions compared to other environments[3], e.g. *Eclipse* (an integrated development environment for several programming languages from the *Eclipse Foundation*). For instance, we have no automatic indenting for if-statements, loops and other structures which contain code-blocks. Also the environment will slow down significantly (for no reason) if we have a large code basis. Even basic behavior like syntax highlighting works very limited. Because of these disadvantages, we strongly recommend not to use the *Maple User Interface* for software solutions with a bigger code basis, but to use an alternative editor for implementing and to use the *Maple User Interface* only for converting the code into *Maple* packages afterwards.
In this thesis, the editor *Notepad++* [35] was used to develop the toolboxes. The editor *Notepad++* can be appropriately enhanced to support *Maple* code. Also the plugin *Compare*, which is able to compare different versions of the toolboxes, was used.
If we use an external editor, we can use the following approach: With the editor (in this case *Notepad++*), the *Maple* source code is saved into ANSI-encoded text files (in this thesis the complete source code was saved into *map*-files). Afterwards we can import it in the *Maple User Interface* and transform it into a *mla*-library. Here an example of this approach using the source code of the toolbox DifferentialDelays:

[3] **Remark:** The disadvantages we sum up here are based on the version 15.01 of *Maple*. They may be solved in later versions of *Maple*.

Listing 2.1: Transformation of the source code

```
> restart;
  #Maple searches there for packages
  libname := libname, "DifferentialDelays";
  #Import the source code
  read("DifferentialDelays.map");
  #Create a directory for the toolbox
  mkdir("DifferentialDelays");
  #Create the mla-file
  march('create',"DifferentialDelays\\DifferentialDelays.mla");
  #Define where the library shall be saved
  savelibname := "DifferentialDelays\\DifferentialDelays.mla";
  #The actual transformation into a mla-file
  savelib('DifferentialDelays');
```

2.3 Overview of the Developed Toolboxes

In this section, the functionalities of the developed toolboxes shall be described briefly. First of all a short remark: Libraries are implemented in *Maple* in form of modules. To stay general, we will call associated modules which serve a certain common purpose **toolbox**, in this thesis.
The implemented toolboxes are strictly separated according to the mathematical use they fulfill. Both toolboxes have so called submodules, i.e. inner libraries, which are able to compute a very certain kind of mathematical tasks. They can be accessed using the [..] command in *Maple*.

Toolbox	**Description**
DifferentialDelays	The main part of the toolbox offers all methods for handling skew polynomials and matrices over skew polynomials in order to find a flat output of a given linear system, except for the functionality in the subordinate packages.
↪[Decompose]	This submodule can be used to decompose matrices using minimal basis decomposition in case of flatness determination of linear systems with and without delays.
↪[LeftFractionUtils]	This submodule contains all methods for handling matrices over left fractions.
↪[PiFlatUtils]	This submodule offers methods to compute and verify the operator π, which is used in case of linear systems with delays.
DifferentialForms	The main part of the toolbox contains all methods to compute the flat output of a nonlinear system, except for the minimal basis decomposition.
↪[MinimalbasisDecomp]	This submodule can be used to decompose matrices using minimal basis decomposition in case of flatness determination of nonlinear systems.

Chapter 3

Mathematical Basics

In this chapter we are going to recall *differential flatness* as a system property of linear and nonlinear systems and π-*flatness* as a system property of linear systems with delays. Therefore, we will introduce the mathematical framework which we need in order to express conditions for flatness at the beginning of this chapter.
Then, we are going to introduce *differential* and π-*flatness* of linear systems and present a procedure to compute the defining operators referring to [4].
Furthermore, necessary and sufficient conditions for differential flatness of nonlinear systems will be recalled from [22], leading to the construction of a flat output.

3.1 Introduction of the Mathematical Framework

In this section, we will briefly explain the theoretical basics and concepts which are fundamental for the following considerations.

3.1.1 The Skew Polynomial Rings $\mathcal{K}\left[\frac{d}{dt}\right]$, $\mathcal{K}[\delta]$

For the determination of flatness according to [1, 4], we need detailed information about skew polynomials and how to handle them. Generally speaking, we handle polynomials over a non commutative ring whose multiplication has specific properties, introduced by Ø. Ore in [30].
First, we have to introduce the σ-*derivation* in order to describe skew polynomials:

Definition 1 ([9, 29, 30])**.** *Let K be a ring without zero divisors and $\sigma : K \to K$ an injective endomorphism. Any map $\vartheta : K \to K$ is called σ-derivation if and only if it fulfills the conditions*

i) $\vartheta(a+b) = \vartheta(a) + \vartheta(b)$

ii) $\vartheta(ab) = \sigma(a)\vartheta(b) + \vartheta(a)b$ *(σ-Leibniz rule)*

$\forall a, b \in K$.

The elements of the skew polynomial ring over K are given by polynomials in Z of the structure

$$p = p_0 Z^0 + p_1 Z^1 + \ldots + p_n Z^n, \ \deg(p) = n \tag{3.1}$$

with $p_i \in K$. The multiplication of Z with elements of field K is given by ([4, 9, 30])

$$Za = \sigma(a)Z + \vartheta(a), \quad \forall a \in K. \tag{3.2}$$

Therefore, the degree of the product of two skew polynomials p and q is given by

$$\deg(pq) = \deg(p) + \deg(q). \tag{3.3}$$

We denote the skew polynomial ring over K with the two maps σ and ϑ by $K[Z; \sigma, \vartheta]$, referring to the mathematical notation from [10].
Let $\sigma = id$, $\vartheta = \frac{d}{dt}$ and let us roughly choose $\frac{d}{dt}$ as symbol. If we choose the field of meromorphic functions $\mathcal{K}$ as ring K, we obtain the skew polynomial ring of *differential operators* $\mathcal{K}\left[\frac{d}{dt}; id, \frac{d}{dt}\right] = \mathcal{K}\left[\frac{d}{dt}\right]$ (see [4]) with

$$\frac{d}{dt}(a_k) = a_k \frac{d}{dt} + \dot{a}_k, \quad \forall a_k \in \mathcal{K} \tag{3.4}$$

and

$$\frac{d}{dt}\left(a_k \frac{d^k}{dt^k}\right) = a_k \frac{d}{dt}\left(\frac{d^k}{dt^k}\right) + \frac{d}{dt}(a_k)\frac{d^k}{dt^k} \tag{3.5}$$

with $a_k \frac{d^k}{dt^k}$ an arbitrary monomial $\in \mathcal{K}\left[\frac{d}{dt}\right]$. Obviously, the given ring $\mathcal{K}\left[\frac{d}{dt}\right]$ is generally not commutative because we have

$$\frac{d}{dt}(a_k) \neq a_k \frac{d}{dt}. \tag{3.6}$$

If $\mathcal{K} = \mathbb{R}$, we obtain the skew polynomial ring over the field of real numbers. In this special case, the ring is commutative according to multiplication since $\frac{d}{dt} a_i \equiv 0$, $\forall a_i \in \mathbb{R}$. Because of that the equation (3.5) leads to

$$\frac{d}{dt}\left(a_k \frac{d^k}{dt^k}\right) = a_k \frac{d^{k+1}}{dt^{k+1}}. \tag{3.7}$$

The sum of two skew polynomials $a, b \in \mathcal{K}\left[\frac{d}{dt}\right]$ with $\deg(a) = n, \deg(b) = m, n \geq m$ also results in a skew polynomial:

$$\begin{aligned} c &= a + b \\ &= \sum_{i=0}^{n} a_i \frac{d^i}{dt^i} + \sum_{j=0}^{m} b_j \frac{d^j}{dt^j} \\ &= \sum_{i=0}^{m} (a_i + b_i) \frac{d^i}{dt^i} + \sum_{j=m+1}^{n} a_i \frac{d^j}{dt^j}. \end{aligned} \tag{3.8}$$

In fact, we have $\deg(c) = n$.
The multiplication of two skew polynomials $a, b \in \mathcal{K}\left[\frac{d}{dt}\right]$ with $\deg(a) = n, \deg(b) = m$ is given by:

$$\begin{aligned} c &= a \cdot b \\ &= \sum_{i=0}^{n} a_i \frac{d^i}{dt^i} \cdot \sum_{j=0}^{m} b_j \frac{d^j}{dt^j} \\ &= \sum_{i=0}^{n} \left(a_i \frac{d^i}{dt^i} \left(\sum_{j=0}^{m} b_j \frac{d^j}{dt^j} \right) \right) \\ &= \sum_{i=0}^{n} \left(a_i \sum_{j=0}^{m} \frac{d^i}{dt^i} \left(b_j \frac{d^j}{dt^j} \right) \right). \end{aligned} \tag{3.9}$$

In fact, we have $\deg(c) = \deg(a) + \deg(b)$ ([29]).

If we set $K = \mathcal{K}$, $\vartheta = 0$ and $\sigma = \delta$, we obtain the skew polynomial ring of *delay operators* $\mathcal{K}[\delta; \delta, 0] = \mathcal{K}[\delta]$ ([4]). In this case, δ is defined as a delay operator with the following properties:

i) $\delta(f(t)) = f(t - \tau)$ (delay)

ii) $\delta^{-1}(f(t)) = f(t + \tau)$ (prediction)

iii) $\delta \cdot f(t) = f(t - \tau)\delta$

with a fixed $\tau \in \mathbb{R}$ in each property. Because of property iii) $\mathcal{K}[\delta]$ is not commutative.
Similar to (3.8), the sum of two skew polynomials $a, b \in \mathcal{K}[\delta]$ with $\deg(a) = n, \deg(b) = m, n \geq m$ is given by:

$$\begin{aligned} c &= a + b \\ &= \sum_{i=0}^{n} a_i \delta^i + \sum_{j=0}^{m} b_j \delta^j \\ &= \sum_{i=0}^{m} (a_i + b_i) \delta^i + \sum_{j=m+1}^{n} a_i \delta^j. \end{aligned} \tag{3.10}$$

I.e. we have $\deg(c) = n$.
Similar to (3.9), the multiplication of two skew polynomials $a, b \in \mathcal{K}[\delta]$ with

$\deg(a) = n, \deg(b) = m$ is given by:

$$\begin{aligned} c &= a \cdot b \\ &= \sum_{i=0}^{n} a_i \delta^i \cdot \sum_{j=0}^{m} b_j \delta^j \\ &= \sum_{i=0}^{n} \left(a_i \delta^i \left(\sum_{j=0}^{m} b_j \delta^j \right) \right) \\ &= \sum_{i=0}^{n} \left(a_i \sum_{j=0}^{m} \delta^i(b_j) \delta^{j+i} \right). \end{aligned} \tag{3.11}$$

In fact, we have $\deg(c) = \deg(a) + \deg(b)$.

3.1.2 The Skew Polynomial Ring $\mathcal{K}\left[\delta, \frac{d}{dt}\right]$

As written in [2, 4], we are able to enhance the skew polynomial ring $\mathcal{K}\left[\frac{d}{dt}\right]$ to have elements $\in \mathcal{K}[\delta]$ as coefficients. We denote this extended skew polynomial ring by $K[Z; \sigma_1, \vartheta_1][Y; \sigma_2, \vartheta_2]$ and set $\vartheta_1 = 0$, $\sigma_1 = \delta$, $\sigma_2 = id$ and $\vartheta_2 = \frac{d}{dt}$ and choose the field of meromorphic functions $\mathcal{K}$ as ring K. Thus, we obtain the skew polynomial ring $\mathcal{K}[\delta; \delta, 0][\frac{d}{dt}; id, \frac{d}{dt}] = \mathcal{K}\left[\delta, \frac{d}{dt}\right]$. This is possible since both maps δ and $\frac{d}{dt}$ commute by the chain rule (see [2, 4] for more details). Therefore, we have the computation rules

$$\frac{d}{dt}(\delta) = 0 \tag{3.12}$$

and

$$\delta \frac{d}{dt} = \frac{d}{dt}\delta. \tag{3.13}$$

3.1.3 The Field of Left Fractions, *lclm* and *gcrd*

For the analysis of linear systems with delays, it is also important to consider the field of left fractions, denoted by $\mathcal{K}(\delta)$. Elements in $\mathcal{K}(\delta)$ are of the form $b^{-1}a$ with $a, b \in \mathcal{K}[\delta]$. In this section, we will discuss some of the more important properties of this field. Detailed information about this field and its construction can be found e.g in [18].
To define addition and multiplication for the field of left fractions, it is required to compute the *least common left multiple* of two skew polynomials in $\mathcal{K}[\delta]$ and the corresponding cofactors. These are important to multiply the denominators and numerators of left fractions:

Definition 2 ([4, 9]). *Let a, b be two skew polynomials $\in \mathcal{K}[\delta] \setminus \{0\}$. $ra \in \mathcal{K}[\delta]$ and $sb \in \mathcal{K}[\delta]$ are called **common left multiple** of a and b if and only if*

$$ra = sb, \quad r, s \in \mathcal{K}[\delta] \setminus \{0\}. \tag{3.14}$$

In addition, $ra \in \mathcal{K}[\delta]$ and $sb \in \mathcal{K}[\delta]$ are called ***least common left multiple*** *of a and b (denoted by $\operatorname{lclm}(a,b)$) if and only if $\forall r', s' \in \mathcal{K}[\delta] \setminus \{0\}$ which satisfy $r'a = s'b$ exists $t \in \mathcal{K}[\delta] \setminus \{0\}$ such that $tr = r' \wedge ts = s'$.*
The two skew polynomials r and s are called ***cofactors*** *of a and b.*

In [9] an algorithm for the computation of the least common left multiple of two skew polynomials was developed. This algorithm was used and improved for automated computation of the cofactors of two skew polynomials in $\mathcal{K}[\delta]$ as part of this thesis:

Algorithm 1: Computation of Cofactors
Input: Two arbitrary skew polynomials $A, B \in \mathcal{K}[\delta] \setminus \{0\}$ with $\deg(A) \geq \deg(B)$[1]
Output: A_i (cofactor of A), $-B_i$ (cofactor of B)

$A_0 = 1$
$A_1 = 0$
$B_0 = 0$
$B_1 = 1$
$R_0 = A$
$R_1 = B$
$i = 2$
while true do
  $Q_0 = \frac{\operatorname{lc}(R_{i-2})}{\delta^{\Delta}(\operatorname{lc}(R_{i-1}))} \cdot \delta^{\Delta}$, $\Delta = \deg(R_{i-2}) - \deg(R_{i-1})$
  $C_i = R_{i-2} - Q_0 \cdot R_{i-1}$
  if $\deg(C_i) < \deg(R_{i-1})$ **then**
    $Q_{i-1} = Q_0$
    $R_i = C_i$
  else
    $Q_1 = \frac{\operatorname{lc}(C_i)}{\delta^{\Delta}(\operatorname{lc}(R_{i-1}))} \cdot \delta^{\Delta}$, $\Delta = \deg(C_i) - \deg(R_{i-1})$
    $Q_{i-1} = Q_0 + Q_1$
    $R_i = C_i - Q_1 \cdot R_{i-1}$
  end if
  $A_i = A_{i-2} - Q_{i-1} \cdot A_{i-1}$
  $B_i = B_{i-2} - Q_{i-1} \cdot B_{i-1}$
  if $\deg(R_i) = -\infty$ **then**
    break
  end if
  $i = i + 1$
end do
return A_i , $-B_i$

The two return values A_i and $-B_i$ represent the cofactors of A and B, i.e.:

$$A_i \cdot A = -B_i \cdot B = \operatorname{lclm}(A, B) \tag{3.15}$$

Now, we are able to describe the addition and multiplication of left fractions over $\mathcal{K}[\delta]$:

[1] If $\deg(A) < \deg(B)$, we have to switch A and B.

Theorem 1 ([4]). *The sum of two left fractions $b^{-1}a$ and $d^{-1}c$ with $a, b, c, d \in \mathcal{K}[\delta]$ and $rb = sd = \text{lclm}(b, d)$ is given by*

$$\begin{aligned} b^{-1}a + d^{-1}c &= (rb)^{-1}(ra) + (sd)^{-1}(sc) \\ &= (rb)^{-1}(ra + sc). \end{aligned} \tag{3.16}$$

Theorem 2 ([4]). *The product of two left fractions $b^{-1}a$ and $d^{-1}c$ with $a, b, c, d \in \mathcal{K}[\delta]$ and $ra = sd = \text{lclm}(a, d)$ is given by*

$$\begin{aligned} b^{-1}a \cdot d^{-1}c &= (rb)^{-1}(ra) \cdot (sd)^{-1}(sc) \\ &= (rb)^{-1}(sc). \end{aligned} \tag{3.17}$$

In the field of left fractions, we also need to define the notion *greatest common right divisor* of two skew polynomials in $\mathcal{K}[\delta]$.

Definition 3 ([9, 19, 28]). *Let a, b be two skew polynomials $\in \mathcal{K}[\delta] \setminus \{0\}$. A polynomial $f \in \mathcal{K}[\delta]$ is called **common right divisor** of a and b if and only if*

$$rf = a \text{ and } sf = b, \quad r, s \in \mathcal{K}[\delta] \setminus \{0\}. \tag{3.18}$$

*In addition, $f \in \mathcal{K}[\delta]$ is called **greatest common right divisor** of a and b (denoted by $\text{gcrd}(a, b)$) if and only if $\forall g \in \mathcal{K}[\delta] \setminus \{0\}$ which satisfy $r'g = a$ and $s'g = b$ with $r', s' \in \mathcal{K}[\delta] \setminus \{0\}$ exists $t \in \mathcal{K}[\delta] \setminus \{0\}$ such that $tg = f$.*

3.1.4 The Skew Polynomial Ring $\mathcal{K}(\delta)\left[\frac{d}{dt}\right]$

Since we also need to be able to express predictions, we have to extend the skew polynomial ring $\mathcal{K}\left[\delta, \frac{d}{dt}\right]$ and introduce the skew polynomial ring $\mathcal{K}(\delta)\left[\frac{d}{dt}\right]$. Similar to the skew polynomial ring $\mathcal{K}\left[\frac{d}{dt}\right]$, we choose the maps $\sigma = id$ and $\vartheta = \frac{d}{dt}$, but decide to use the field of left fractions over $\mathcal{K}[\delta]$ as ring K. So we obtain the skew polynomial ring $\mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ [4].
Because of the properties of ϑ, it is now possible to get derivatives of left fractions. Thus, we have to introduce the following computation rule:

Theorem 3 ([4, 18]). *The derivative of a left fraction $b^{-1}a$ with $a, b \in \mathcal{K}[\delta]$ and $\frac{d}{dt}(b) \neq 0$ with respect to t is given by*

$$\frac{d}{dt}\left(b^{-1}a\right) = (sb)^{-1}(s\frac{d}{dt}(a) - ra) \tag{3.19}$$

with $rb = s\frac{d}{dt}(b) = \text{lclm}(b, \frac{d}{dt}(b))$. If $\frac{d}{dt}(b) = 0$, the derivative is given by

$$\frac{d}{dt}\left(b^{-1}a\right) = b^{-1}\frac{d}{dt}(a). \tag{3.20}$$

3.1.5 Hyper-regularity as Property of Matrices over Skew Polynomials

In this section, we consider all matrices to be matrices over the ring of skew polynomials $K\left[\frac{d}{dt}\right]$. Thus, there is no commutativity in general.
Furthermore, the inverse M^{-1} of a skew polynomial matrix $M \in K\left[\frac{d}{dt}\right]$ is, in general, no skew polynomial matrix itself, because it may contain fractions with $\frac{d}{dt}$ in the denominator. In this thesis, we want to focus on matrices which have a skew polynomial matrix as inverse.

Definition 4. *A matrix $M \in K\left[\frac{d}{dt}\right]^{n\times n}$ is called **unimodular** if and only if its inverse M^{-1} is in $K\left[\frac{d}{dt}\right]^{n\times n}$. The set of such unimodular matrices is also possibly referred to as the linear group $Gl_n\left(K\left[\frac{d}{dt}\right]\right)$.*

A very important tool for the determination of a flat output of linear and nonlinear systems is the decomposition of skew polynomial matrices. To serve this purpose, we introduce the **Smith-Jacobson** decomposition:

Theorem 4 ([10])**.** *Every matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ can be transformed into the **Smith-Jacobson form**[2] by using two unimodular matrices $V \in K\left[\frac{d}{dt}\right]^{r\times r}$, $U \in K\left[\frac{d}{dt}\right]^{s\times s}$:*

$$VMU = \begin{cases} \begin{pmatrix} \Delta_r & 0_{r\times(s-r)} \end{pmatrix} & \text{if } r \leq s \\ \begin{pmatrix} \Delta_s \\ 0_{(r-s)\times s} \end{pmatrix} & \text{if } r \geq s \end{cases} \tag{3.21}$$

Δ_r and Δ_s must satisfy[3] *$\Delta_r = diag(\lambda_1, \ldots, \lambda_r)$ resp. $\Delta_s = diag(\lambda_1, \ldots, \lambda_s)$ with $\lambda_i \| \lambda_j \ \forall i < j$*[4]*.*

Definition 5 ([23])**.** *A matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ is called **hyper-regular** if and only if it satisfies Theorem 4 and in (3.21) we have $\Delta_r = I_r$ resp. $\Delta_s = I_s$.*

Let us recall, that the inverse M^{-1} of an unimodular matrix M is always unimodular itself and both left and right inverse of M[5].

Theorem 5. *Every square hyper-regular matrix $M \in K\left[\frac{d}{dt}\right]^{r\times r}$ is also unimodular*[6]*.*

[2] The *Smith-Jacobson* form is also called *Smith* form or *Jacobson* form.

[3] The order of elements on the diagonal can be changed using simple row or column operations. That means we can always transform Δ_r and Δ_s into the needed form by switching the rows or columns of V and U. The matrices V and U will stay unimodular.

[4] $\lambda_i \| \lambda_j$ means λ_i totally divides λ_j, i.e. $\exists \alpha \in K\left[\frac{d}{dt}\right] : \lambda_i \| \alpha \| \lambda_j \wedge k\alpha = \alpha k, \forall k \in K\left[\frac{d}{dt}\right]$.

[5] This can easily be shown by computing the left and right inverse of M from $VMU = I$.

[6] This can also easily be shown by an algebraic transformation of $VMU = I$.

3.1.6 Minimal Bases for the Decomposition of Skew Polynomial Matrices

The decomposition of skew polynomial matrices via minimal bases is a special case of the *Smith-Jacobson* decomposition and will be described in detail in this section. All matrix decompositions in the developed toolboxes will be done via minimal bases since the minimal basis decomposition is very convenient for the automated evaluation of linear and nonlinear systems[7]. In a first step the terms **module** and **submodule** will be defined.

Definition 6 ([19]). *An abelian group $\mathcal{M}$ together with a bilinear map $\mathcal{R} \times \mathcal{M} \to \mathcal{M}$ is called **left $\mathcal{R}$-module** over the ring $\mathcal{R}$ if and only if the map satisfies*

i) $1_{\mathcal{R}} m = m \quad \forall m \in \mathcal{M}$ *(identity element)*

ii) $r(m+n) = rm + rn \quad \forall r \in \mathcal{R}$ *and* $m, n \in \mathcal{M}$

iii) $(r+s)m = rm + sm \quad \forall r, s \in \mathcal{R}$ *and* $m \in \mathcal{M}$

iv) $(rs)m = r(sm) \quad \forall r, s \in \mathcal{R}$ *and* $m \in \mathcal{M}$

The **right $\mathcal{R}$-module** is similarly defined with a bilinear map $\mathcal{M} \times \mathcal{R} \to \mathcal{M}$. If $\mathcal{R}$ is a commutative ring, the left $\mathcal{R}$-module and the right $\mathcal{R}$-module are identical. In this case it is simply called **$\mathcal{R}$-module**.

Definition 7 ([19]). *A subgroup $\mathcal{N} \subseteq \mathcal{M}$ is called **submodule** of the module $\mathcal{M}$ over the ring $\mathcal{R}$ if and only if $rn \in \mathcal{N} \ \forall r \in \mathcal{R}$ and $n \in \mathcal{N}$.*

Definition 8 ([10, 19]). *Let $\mathcal{M}$ be a module over the ring $\mathcal{R}$. $\mathcal{M}$ is called **free** if and only if $\forall m \in \mathcal{M}$ we have*

$$\exists\, m_1, \ldots, m_n \in \mathcal{M} : \ m = r_1 m_1 + \cdots + r_n m_n \tag{3.22}$$

with $r_i \in \mathcal{R}$.

Definition 9 ([19]). *Let $\mathcal{M}$ be a free module over the ring $\mathcal{R}$. The distinct set of elements $\{m_1, \ldots, m_n\}$ which satisfy (3.22) with $m_1, \ldots, m_n$ linearly independent is called **basis** of $\mathcal{M}$ over $\mathcal{R}$.*

The $\mathcal{K}\left[\frac{d}{dt}\right]$-module $\mathcal{M}$ over the ring of skew polynomials $\mathcal{K}\left[\frac{d}{dt}\right]$, which is spanned by the basis $\{b_1, \ldots, b_n\}, b_i \in K\left[\frac{d}{dt}\right]^{1\times n}$, admits the representation

$$\mathcal{M} = \{r_1 b_1 + \ldots + r_n b_n \mid r_i \in K\left[\tfrac{d}{dt}\right], b_i \in K\left[\tfrac{d}{dt}\right]^{1\times n}\}. \tag{3.23}$$

Furthermore, the terms *row degree*, *column degree*, *row order* and *column order* will be recalled:

[7] It should be mentioned at this point that the decomposition via minimal bases is only able to decompose hyper-regular matrices. In general, minimal basis decomposition cannot create a *Smith-Jacobson* form for non-hyper-regular matrices.

Definition 10 ([28, 33]). *The highest degree in $\frac{d}{dt}$ among the elements of the i'th row M_i of a skew polynomial matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ is called* ***row degree*** *of the i'th row:*

$$\text{degree}_{row}(M_i) = \max\{\deg(M_{i,j})\}, \quad j = 1..s. \tag{3.24}$$

The ***column degree*** *of the j'th column M_j is defined similar as the highest occurring degree in $\frac{d}{dt}$ of the j'th column:*

$$\text{degree}_{column}(M_j) = \max\{\deg(M_{i,j})\}, \quad i = 1..r. \tag{3.25}$$

Definition 11 ([28]). *Let $M \in K\left[\frac{d}{dt}\right]^{r\times s}$. The* ***row order*** *of M is given by*

$$\text{order}_{row}(M) = \sum_{i=1}^{r} \text{degree}_{row}(M_i). \tag{3.26}$$

The ***column order*** *of M is given by*

$$\text{order}_{column}(M) = \sum_{j=1}^{s} \text{degree}_{column}(M_j). \tag{3.27}$$

Now we are able to recall the definition for *minimal basis*:

Definition 12 ([17, 28]). *Let the rows of a skew polynomial matrix $G \in K\left[\frac{d}{dt}\right]^{r\times s}$ be a basis for a submodule $U \in K\left[\frac{d}{dt}\right]^{1\times s}$. The rows of G are called* ***minimal basis*** *if G has the lowest row order among all bases for the submodule U.*
Similarly, the columns of a skew polynomial matrix $G \in K\left[\frac{d}{dt}\right]^{r\times s}$ are called ***minimal basis*** *if the columns of G are a basis for a submodule $U \in K\left[\frac{d}{dt}\right]^{r\times 1}$ and if G has the lowest column order among all bases for the submodule U.*

In conjunction with this definition, we are able to find another way to check a skew polynomial matrix for hyper-regularity:

Theorem 6 ([4, 29, 33]). *The skew polynomial matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ with $r \geq s$ is* hyper-regular *if and only if the rows of M are a basis of $\mathbb{R}^s$.*
If $r \leq s$, the columns of M have to be a basis of $\mathbb{R}^r$.

Remark about Theorem 6: The two assertions

- the rows resp. columns of M are a basis of $\mathbb{R}^s$ resp. $\mathbb{R}^r$
- the minimal basis of M has degree 0 in $\frac{d}{dt}$

are equivalent.
Based on Theorem 6, we are now able to construct an algorithm to check skew polynomial matrices for hyper-regularity. The algorithm described in this section is adapted from [28] to run in *Maple*. But first, we have to introduce the *leading coefficient matrix*:

Definition 13 ([28]). *Let $C(i)$ be for each row M_i a set of column indexes such that*

$$C(i) = \{j | \deg(M_{ij}) = \text{degree}_{row}(M_i)\}, \quad i = 1..r, \ j = 1..s. \tag{3.28}$$

*Additionally, let $\text{lc}(M_{ij})$ be the leading coefficient of the polynomial M_{ij}. Then we obtain the **row leading coefficient matrix** $\text{LC}_{row}(M) \in K^{r\times s}$, whose elements are defined by*

$$\text{LC}_{row}(M)_{ij} = \begin{cases} 0 & \text{if } j \notin C(i) \\ \text{lc}(M_{ij}) & \text{if } j \in C(i) \end{cases}, \quad i = 1..r, \ j = 1..s. \tag{3.29}$$

Similar to that, we are able to define the leading coefficient matrix of M column-wise: Let $R(j)$ be for each column M_j a set of row indexes such that

$$R(j) = \{i | \deg(M_{ij}) = \text{degree}_{column}(M_j)\}, \quad i = 1..r, \ j = 1..s. \tag{3.30}$$

*Then we obtain the **column leading coefficient matrix** $\text{LC}_{column}(M) \in K^{r\times s}$:*

$$\text{LC}_{column}(M)_{ij} = \begin{cases} 0 & \text{if } i \notin R(j) \\ \text{lc}(M_{ij}) & \text{if } i \in R(j) \end{cases}, \quad i = 1..r, \ j = 1..s. \tag{3.31}$$

Now it is possible to determine whether the corresponding matrix M is a minimal basis by evaluating the rank of the leading coefficient matrix:

Definition 14 ([28]). *The matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ is called **row-reduced** if and only if its row leading coefficient matrix has full rank.*
*Similar to that, the matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ is called **column-reduced** if and only if its column leading coefficient matrix has full rank.*

Theorem 7 ([28]). *Let $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ be a skew polynomial matrix whose rows (in case of $r \geq s$) resp. columns (in case of $r \leq s$) span the submodule $U \subseteq K\left[\frac{d}{dt}\right]^{\max\{r,s\}}$. In this case both assertions are equivalent:*

- *M is row- resp. column-reduced*
- *the rows resp. the columns of M are a minimal basis of U*

Theorem 7 is a simplified and abridged version of the *main theorem of minimal bases*, which is described in greater detail in [28].
From Theorems 6 and 7 results that a matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ is hyper-regular if and only if its leading coefficient matrix has full rank and the rows (in case of $r \geq s$) resp. columns (in case of $r \leq s$) of M span the module $\mathbb{R}^{min\{r,s\}}$[8].
To construct the minimal basis of a skew polynomial matrix M, it is necessary to reduce the row degrees resp. column degrees step by step to zero using row resp. column operations. In case of $r > s$ the reduction has to be done by row

[8] This is equivalent to the assertion that the minimal basis of M has degree 0 in $\frac{d}{dt}$.

operations and in case of $r < s$ by column operations. In the special case of a square matrix (i.e. $r = s$) both ways are possible ([33]).
The leading coefficient matrix of M has full rank if and only if M is a minimal basis. That means, as long as M is no minimal basis, there exists a linear combination of rows resp. columns of M to reduce the degree of a row resp. a column at least by 1.
First, we want to describe the algorithm for row-wise reduction:

Algorithm 2: Row-wise Decomposition
Input: An arbitrary skew polynomial matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ with $r \geq s$.
Output: If M is hyper-regular, an unimodular skew polynomial matrix $V \in K\left[\frac{d}{dt}\right]^{r\times r}$ which transforms M into a *Smith-Jacobson* form. Else *NULL*.

$V = I_r$
while $\exists \alpha \neq 0 : \alpha^T \operatorname{LC}_{row}(M) = 0$ **do**
  $n =$ index of the uppermost row of M with highest row degree and $\alpha_n \neq 0$
  for $i = 1..r$
    $\widetilde{\alpha}_i = \alpha_i \frac{d^\Delta}{dt^\Delta}$, $\Delta = \operatorname{degree}_{row}(M_n) - \operatorname{degree}_{row}(M_i)$
  end for
  for $i = 1..r$
    if $i = n$ **then**
      $V'_{i,1..r} = \widetilde{\alpha}^T$
    else
      $V'_{i,1..r} = e_i$ (unit vector)
    end if
  end for
  $M = V' \cdot M$
  $V = V' \cdot V$
end do
if $\max\{\operatorname{degree}_{row}(M_i) | i = 1..r\} = 0$ **then**
  for $i = 1..r$
    if $\operatorname{degree}_{row}(M_i) = -\infty$
      $j =$ index of the uppermost row with $\operatorname{degree}_{row}(M_j) = 0$
      Switch i'th row and j'th row of M
      Switch i'th row and j'th row of V
    end if
  end for
  if $\exists (r - s)$ zero rows in M **then**
    $M^{-1} =$ left inverse of M
    $V = M^{-1} \cdot V$
    return V
  else
    return NULL
  end if
else
  return NULL
end if

The types of the used variables are: $V \in K\left[\frac{d}{dt}\right]^{r\times r}, \alpha \in K^{r\times 1}, n \in \mathbb{N}, \widetilde{\alpha} \in K\left[\frac{d}{dt}\right]^{r\times 1}, V' \in K\left[\frac{d}{dt}\right]^{r\times r}$ und $M^{-1} \in K\left[\frac{d}{dt}\right]^{r\times r}$.

Remark: In this algorithm the essential condition $\alpha^T \operatorname{LC}_{row}(M) = 0$ ignores all α which represent only linear combination of all zero rows of $\operatorname{LC}_{row}(M)$. The matrix V' which contains the particular row operation is composed row-wise. The n'th row represents the modified vector α. All other rows are those of an identity matrix.

Similar to the algorithm above, we are able to describe the algorithm for column-wise reduction (in case of $r \leq s$):

Algorithm 3: Column-wise Decomposition
Input: An arbitrary skew polynomial matrix $M \in K\left[\frac{d}{dt}\right]^{r\times s}$ with $r \leq s$.
Output: If M is hyper-regular, an unimodular skew polynomial matrix $U \in K\left[\frac{d}{dt}\right]^{s\times s}$ which transforms M into a *Smith-Jacobson* form. Else *NULL*.

$U = I_s$
while $\exists \alpha \neq 0 : \operatorname{LC}_{column}(M)\,\alpha = 0$ **do**
  $n =$ index of the first column of M with highest column degree and $\alpha_n \neq 0$
  for $i = 1..s$
    $\widetilde{\alpha}_i = \alpha_i \frac{d^\Delta}{dt^\Delta}$, $\Delta = \operatorname{degree}_{column}(M_n) - \operatorname{degree}_{column}(M_i)$
  end for
  for $i = 1..s$
    if $i = n$ **then**
      $U'_{1..s,i} = \widetilde{\alpha}$
    else
      $U'_{1..s,i} = e_i$ (unit vector)
    end if
  end for
  $M = M \cdot U'$
  $U = U \cdot U'$
end do
if $\max\{\operatorname{degree}_{column}(M_i) | i = 1..s\} = 0$ **then**
  for $i = 1..s$
    if $\operatorname{degree}_{column}(M_i) = -\infty$
      $j =$ index of the first column with $\operatorname{degree}_{column}(M_j) = 0$
      Switch i'th column and j'th column of M
      Switch i'th column and j'th column of U
    end if
  end for
  if $\exists(s-r)$ zero columns in M **then**
    $M^{-1} =$ right inverse of M
    $U = U \cdot M^{-1}$
    return U
  else
    return NULL
  end if
else
  return NULL
end if

The types of the used variables are: $U \in K\left[\frac{d}{dt}\right]^{s\times s}, \alpha \in K^{s\times 1}, n \in \mathbb{N}$,

$\widetilde{\alpha} \in K\left[\frac{d}{dt}\right]^{s\times 1}, U' \in K\left[\frac{d}{dt}\right]^{s\times s}$. The matrix U' which contains the particular column operation is composed column-wise. The n'th column represents the modified vector α. All other columns are those of an identity matrix.

3.2 Differential and π-Flatness of Linear Systems

The main goal of this thesis is the symbolic computation of flat outputs of finite dimensional control systems. At first, we deal with linear systems. They can be written in the form

$$Ax = Bu \tag{3.32}$$

with $A \in K\left[\frac{d}{dt}\right]^{n\times n}, B \in K\left[\frac{d}{dt}\right]^{n\times m}, x = (x_1, \ldots, x_n)^T$ and $u = (u_1, \ldots, u_m)^T$. Let us give an informal definition that will be made precise in sections 3.2.1 – 3.2.3: If the system (3.32) is **flat**, there exists a **flat output** y of the system. One of the properties of a flat output is that all x_i and u_i can be described as functions of the entries y_i of the flat output and a finite number of their time derivatives $y_i^{(j)}$. Based on such a flat output we are able to construct a flatness-based controller or feedback controller for this system (for more information see e.g. [14, 15, 21, 27]). This section is about the determination of a flat output for the linear system (3.32).

3.2.1 Flatness Analysis for Linear Time-Invariant Systems

First, we take a look at *linear time-invariant systems* (also known as *LTI systems*) of the form (3.32) with $A \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times n}$ and $B \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times m}$. Now *differential flatness* as a system property can be defined in the following:
We assume that B is hyper-regular. This is equivalent to the assertion that the inputs of the system are independent. In addition, we assume that the rows of (A, B) are linearly independent. This ensures that there are no redundant equations in (3.32) (see [1]). The informal definition from the introduction, specialized to this case, is the following:

Definition 15 ([23]). *The linear time-invariant system in the form of (3.32) with $A \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times n}$ and $B \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times m}$ is called **differentially flat** if and only if there exist matrices[9] $P \in \mathbb{R}\left[\frac{d}{dt}\right]^{m\times n}$, $Q \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times m}$ and $R \in \mathbb{R}\left[\frac{d}{dt}\right]^{m\times m}$ such that:*

$$y = Px, \quad x = Qy, \quad u = Ry, \quad PQ = I_m. \tag{3.33}$$

*The vector y is called **flat output** of the system.*
***Remark:** The flat output y is not unique.*

[9] The three matrices P, Q and R are also called **defining operators**.

Theorem 8. *In case of linear systems, both system properties* differential flatness *and* controllability *are equivalent.*

Theorem 8 results from [14], see also [23, 21, 1, 2].
From now on, we will always assume that the number of independent inputs u_i is less or equal to the number of partial states x_i, i.e. $n \geq m$[10].
Since B is hyper-regular, according to the assumption above, there exists an unimodular matrix $\widetilde{M} \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times n}$ (referring to the results given in section 3.1.6) such that

$$\widetilde{M}B = \begin{pmatrix} I_m \\ 0_{n-m\times m} \end{pmatrix}. \tag{3.34}$$

Applying $\widetilde{M}$ to the system (3.32), we get

$$\widetilde{M}Ax \quad = \quad \widetilde{M}Bu = \begin{pmatrix} I_m \\ 0_{n-m\times m} \end{pmatrix} u = \begin{pmatrix} u_1 \\ \vdots \\ u_m \\ 0_{n-m\times 1} \end{pmatrix}. \tag{3.35}$$

Since we can set the inputs u arbitrarily, they can always be set to

$$u = \begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{M}Ax \tag{3.36}$$

to satisfy the first m equations of (3.35). Thus, only the last $n-m$ rows remain in (3.35). They represent the implicit equations of the linear system. By introducing the matrix $F \in \mathbb{R}\left[\frac{d}{dt}\right]^{n-m\times n}$ with

$$F = \begin{pmatrix} 0_{n-m\times m} & I_{n-m} \end{pmatrix} \widetilde{M}A \tag{3.37}$$

we are able to introduce the implicit system representation of the system (3.32) resp. (3.35):

$$Fx = 0_{n-m\times 1}. \tag{3.38}$$

A condition for F is brought by the following theorem:

Theorem 9 ([21, 22])**.** *The system (3.32) is flat if and only if the matrix F in (3.37) is hyper-regular, i.e.:*

$$\exists \widetilde{Q} \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times n} : \; F\widetilde{Q} = \begin{pmatrix} I_{n-m} & 0_{n-m\times m} \end{pmatrix}. \tag{3.39}$$

[10] In the special case $n = m$ the system is always flat. In this case a solution for (3.33) is given by $P = I_n$, $Q = I_n$ and $R = B^{-1}A$ with $y = x$ as flat output. At this point we want to recall that the flat output is not unique.

Now, we want to demonstrate that hyper-regularity of F provides the differential flatness according to Definition 15. This shall motivate the algorithms later in this section.
By decomposing the last m columns of $\widetilde{Q}$ from (3.39) via minimal bases

$$Q = \widetilde{Q} \begin{pmatrix} 0_{n-m\times m} \\ I_m \end{pmatrix} \tag{3.40}$$

we can easily prove that there exists an unimodular matrix $\widetilde{P} \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times n}$ such that

$$\widetilde{P}Q = \begin{pmatrix} I_m \\ 0_{n-m\times m} \end{pmatrix}. \tag{3.41}$$

It can be shown that a flat output of the system (3.32) is given by

$$y = \underbrace{\begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{P}}_{P} x = Px \tag{3.42}$$

and the differential parameterization of x is given by

$$x = Qy. \tag{3.43}$$

Furthermore, we may set the parameterization of x from (3.43) into (3.38) and obtain:

$$\begin{aligned} Fx &= FQy \\ &= F\widetilde{Q} \begin{pmatrix} 0_{n-m\times m} \\ I_m \end{pmatrix} y \\ &= \underbrace{\begin{pmatrix} I_{n-m} & 0_{n-m\times m} \end{pmatrix} \begin{pmatrix} 0_{n-m\times m} \\ I_m \end{pmatrix}}_{0_{n-m\times m}} y = 0_{n-m\times 1} \end{aligned} \tag{3.44}$$

Thus, the parameterization $x = Qy$ satisfies $\forall y$ the implicit system representation (3.38).
Due to the fact that $\widetilde{Q}$ is unimodular, it is also possible to compute the matrix $\widetilde{Q}^{-1}$ simultaneously with $\widetilde{Q}$ according to the minimal basis decomposition (3.39). Using that, the second minimal basis decomposition (3.41) can be avoided. A matrix $\widetilde{P}$ satisfying (3.41) is given by

$$\widetilde{P} = \begin{pmatrix} 0_{m\times n-m} & I_m \\ I_{n-m} & 0_{n-m,m} \end{pmatrix} \widetilde{Q}^{-1}. \tag{3.45}$$

With (3.42) P results in

$$P = \begin{pmatrix} 0_{m\times n-m} & I_m \end{pmatrix} \widetilde{Q}^{-1}. \tag{3.46}$$

And indeed, with (3.40) – (3.42) we get:

$$\begin{aligned} Px &= PQy = \begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{P}Qy \\ &= \begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \begin{pmatrix} I_m \\ 0_{n-m\times m} \end{pmatrix} y \\ &= I_m y = y. \end{aligned} \tag{3.47}$$

The still missing parameterization of u results from (3.36):

$$\begin{aligned} u &= \begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{M}Ax \\ &= \underbrace{\begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{M}AQ}_{R} y. \end{aligned} \tag{3.48}$$

This leads to the following theorem:

Theorem 10 ([22, 2])**.** *If the matrix F in (3.37) is hyper-regular, a possible solution for (3.33) is given by the three matrices*

$$P = \begin{pmatrix} 0_{m\times n-m} & I_m \end{pmatrix} \widetilde{Q}^{-1} \tag{3.49}$$

$$Q = \widetilde{Q} \begin{pmatrix} 0_{n-m\times m} \\ I_m \end{pmatrix} \tag{3.50}$$

$$R = \begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{M}AQ. \tag{3.51}$$

This leads to an algorithm for the flatness analysis of linear systems in order to compute a flat output (if possible) to:

Algorithm 4: Computing the Defining Operators for Linear Time-Invariant Systems

Input: The two matrices $A \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times n}$, $B \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times m}$, which characterize the linear system (3.32). It must be $n \geq m$.

Output: If the system is flat, the defining operators $P \in \mathbb{R}\left[\frac{d}{dt}\right]^{m\times n}$, $Q \in \mathbb{R}\left[\frac{d}{dt}\right]^{n\times m}$ and $R \in \mathbb{R}\left[\frac{d}{dt}\right]^{m\times m}$. Else *NULL*.

```
if not ∃M̃ : M̃B = ( I_m ; 0_{n−m×m} ) then
    return NULL
end if
if ∃α ≠ 0 : α^T ( A  B ) = 0 then
    return NULL
end if
M̃B = ( I_m ; 0_{n−m×m} ) ⇒ M̃
F = ( 0_{n−m×m}  I_{n−m} ) M̃A
if not ∃Q̃ : FQ̃ = ( I_{n−m}  0_{n−m×m} ) then
    return NULL
end if
```

$F\widetilde{Q} = \begin{pmatrix} I_{n-m} & 0_{n-m\times m} \end{pmatrix} \Rightarrow \widetilde{Q}$
$\widetilde{Q}^{-1}\widetilde{Q} = I_n \Rightarrow \widetilde{Q}^{-1}$
$P = \begin{pmatrix} 0_{m\times n-m} & I_m \end{pmatrix} \widetilde{Q}^{-1}$
$Q = \widetilde{Q} \begin{pmatrix} 0_{n-m\times m} \\ I_m \end{pmatrix}$
$R = \begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{M}AQ$
return P, Q, R

3.2.2 Flatness Analysis for Linear Time-Varying Systems

In case of linear time-invariant systems, the system matrices A and B in (3.32) are always defined over the ring $\mathbb{R}\left[\frac{d}{dt}\right]$. In case of linear time-varying systems, the system matrices are based on the ring of meromorphic functions $\mathcal{K}\left[\frac{d}{dt}\right]$. That means the coefficients of the occurring skew polynomials are now meromorphic functions of t. The field of meromorphic functions has no zero divisors, that is if

$$k(t) \cdot l(t) = 0 \Rightarrow k(t) = 0 \ \vee \ l(t) = 0 \tag{3.52}$$

with $k(t), l(t) \in \mathcal{K}$. Furthermore time derivatives of meromorphic functions are also meromorphic[11].
Because of the generalization of the used skew polynomial ring, we do not have a commutative multiplication of skew polynomials.
The flatness analysis of the linear time-varying system (3.32) with $A \in \mathcal{K}\left[\frac{d}{dt}\right]^{n\times n}$, $B \in \mathcal{K}\left[\frac{d}{dt}\right]^{n\times m}$, $x \in \mathcal{K}^n$ and $u \in \mathcal{K}^m$ works similar to the analysis of the linear time-invariant system, which is described in section 3.2.1 and especially in algorithm 4. We only have to change the skew polynomial ring of the used matrices to $\mathcal{K}\left[\frac{d}{dt}\right]$.
Remark: We have to keep in mind that there is no commutativity in $\mathcal{K}\left[\frac{d}{dt}\right]$. This has to be considered in algorithm 4.

3.2.3 Flatness Analysis for Linear Time-Varying Systems with Delays

In case of linear systems with delays, we consider the ring $\mathcal{K}\left[\delta, \frac{d}{dt}\right]$. The reader will find a thorough discussion on this choice in [2]. In order to use flatness as a system property for a system (3.32) which has been extended with delays we have to specialize the informal definition from the introduction:

Definition 16 ([2, 4])**.** *The linear system of the form (3.32) with* $A \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{n\times n}$ *and* $B \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{n\times m}$ *is called* ***π-flat*** *if and only if there exist an operator*

[11] A detailed analysis of meromorphic functions can be found in [11].

$\pi \in \mathcal{K}[\delta]$ and matrices $P \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{m\times n}$, $Q \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{n\times m}$ and $R \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{m\times m}$ such that:

$$y = \underbrace{\pi^{-1}P}_{\overline{P}}x, \quad x = \underbrace{\pi^{-1}Q}_{\overline{Q}}y, \quad u = \underbrace{\pi^{-1}R}_{\overline{R}}y, \quad \overline{P}\,\overline{Q} = I_m. \tag{3.53}$$

*The vector y is called **π-flat output** of the system.*

Remark: Since $\overline{P}$, $\overline{Q}$ and $\overline{R}$ contain fractions in δ, the further computations must be done in $\mathcal{K}(\delta)\left[\frac{d}{dt}\right]$.

In order to compute a suitable operator π which satisfies (3.53) we can use the approach according to [4]: First of all, we construct operators $\pi_{\overline{P}}, \pi_{\overline{Q}}$ and $\pi_{\overline{R}}$ which eliminate all predictions in the corresponding matrices.[12] Then π is given by

$$\pi = \mathrm{lclm}(\pi_{\overline{P}}, \pi_{\overline{Q}}, \pi_{\overline{R}}). \tag{3.54}$$

The algorithm 4 can now be extended to:

Algorithm 5: Computing the Defining Operators for Linear Time-Varying Systems with Delays

Input: The two matrices $A \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{n\times n}, B \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{n\times m}$ which characterize the linear system (3.32). It must be $n \geq m$.

Output: If the system is $\pi-$flat, the matrices $P \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{m\times n}$, $Q \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times m}$ and $R \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{m\times m}$ and the delay operator π. Else *NULL*.

1 **if not** $\exists \widetilde{M} : \widetilde{M}B = \begin{pmatrix} I_m \\ 0_{n-m\times m} \end{pmatrix}$ **then**
2 **return NULL**
3 **end if**
4 **if** $\exists \alpha \neq 0 : \alpha^T \begin{pmatrix} A & B \end{pmatrix} = 0$ **then**
5 **return NULL**
6 **end if**
7 $\widetilde{M}B = \begin{pmatrix} I_m \\ 0_{n-m\times m} \end{pmatrix} \Rightarrow \widetilde{M}$
8 $F = \begin{pmatrix} 0_{n-m\times m} & I_{n-m} \end{pmatrix}\widetilde{M}A$
9 **if not** $\exists \widetilde{Q} : F\widetilde{Q} = \begin{pmatrix} I_{n-m} & 0_{n-m\times m} \end{pmatrix}$ **then**
10 **return NULL**
11 **end if**
12 $F\widetilde{Q} = \begin{pmatrix} I_{n-m} & 0_{n-m\times m} \end{pmatrix} \Rightarrow \widetilde{Q}$
13 $\widetilde{Q}^{-1}\widetilde{Q} = I_n \Rightarrow \widetilde{Q}^{-1}$
14 $\overline{P} = \begin{pmatrix} 0_{m\times n-m} & I_m \end{pmatrix}\widetilde{Q}^{-1}$
15 $\overline{Q} = \widetilde{Q}\begin{pmatrix} 0_{n-m\times m} \\ I_m \end{pmatrix}$

[12] A possible option is to take all occurring denominators of left fractions of the corresponding matrix and compute the least common left multiple of them.

$\overline{R} = \begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{M} A \overline{Q}$
Compute $\pi_{\overline{P}}$: $\pi_{\overline{P}}\overline{P} \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{m\times n}$
Compute $\pi_{\overline{Q}}$: $\pi_{\overline{Q}}\overline{Q} \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{n\times m}$
Compute $\pi_{\overline{R}}$: $\pi_{\overline{R}}\overline{R} \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{m\times m}$
$\pi = \mathrm{lclm}(\pi_{\overline{P}}, \pi_{\overline{Q}}, \pi_{\overline{R}})$
$P = \pi\overline{P}$
$Q = \pi\overline{Q}$
$R = \pi\overline{R}$
return P, Q, R, π

It has to be mentioned that certain restrictions for the trajectories of y exist (for more details see [2]).

3.3 Differential Flatness of Nonlinear Systems

In section 3.2 *flatness* for linear systems was introduced. Now this system property will also be defined for nonlinear systems. The mathematical structures, which will be introduced in this chapter, are reduced to their properties which are relevant to the determination of flatness. The mathematical background of the differential geometry of manifolds of jets of infinite order, which is applied here, may be found in [21].

3.3.1 Flatness of Explicit Nonlinear Systems

A definition for flatness of explicit nonlinear systems is given by:

Definition 17 ([14, 15, 20, 21, 22]). *The explicit nonlinear system*

$$\dot{x} = f(x,u), \quad \mathrm{rank}\left(\frac{\partial f}{\partial u}\right) = m \tag{3.55}$$

*with $x \in X \subset \mathbb{R}^n, u \in U \subset \mathbb{R}^m$ and $f \in \mathcal{K}$ (field of meromorphic functions over $X \times U$) is called **differentially flat** if and only if there exists an output $y = \begin{pmatrix} y_i & \dots & y_m \end{pmatrix}^T$ such that*

$$y = \psi_0(x, u, \dot{u}, \dots). \tag{3.56}$$

In addition, there must exist expressions of x and u such that

$$x = \varphi_x(y, \dot{y}, \dots, y^{(\kappa-1)}) \tag{3.57}$$
$$u = \varphi_u(y, \dot{y}, \dots, y^{(\kappa)}). \tag{3.58}$$

*The output y is called **flat output** of the system (3.55). It is not unique.*

3.3.2 Flatness of Implicit Nonlinear Systems

If an explicit system (3.55) satisfies rank $\left(\frac{\partial f}{\partial u}\right) = m$, $x \in X \subset \mathbb{R}^n$, $u \in U \subset \mathbb{R}^m$, we can assume without loss of generality that

$$\operatorname{rank}\left(\frac{\partial(f_{n-m+1}, \ldots, f_n)}{\partial u}\right) = m. \tag{3.59}$$

This means that we can solve the last m equations of the explicit system for u. This yields

$$u = \mu(x, \dot{x}_{n-m+1}, \ldots, \dot{x}_n). \tag{3.60}$$

By substituting (3.60) into the first $n - m$ equations of (3.55), we gain the implicit system representation

$$F(x, \dot{x}) = 0, \quad \operatorname{rank}\left(\frac{\partial F}{\partial \dot{x}}\right) = n - m. \tag{3.61}$$

This corresponds to equation (3.38) in case of linear systems.
Every implicit system can also be transformed into an equivalent explicit system (see [5]).
In order to go more into detail referring to the system class of nonlinear systems, we have to introduce several differential geometric constructs. First of all, we introduce the infinitely differentiable manifold

$$\mathfrak{X} = X \times \mathbb{R}^n_\infty \tag{3.62}$$

as *prolongation* of the state space X with $\dim X = n$. The coordinates of the prolongation $\mathfrak{X}$ are given by

$$\overline{x} = (x, \dot{x}, \ldots). \tag{3.63}$$

In a second step, the *tangent space* $T_{\overline{x}}\mathfrak{X}$ of $\mathfrak{X}$ shall be introduced. This is the tangent space of the prolongation $\mathfrak{X}$ at the arbitrary point $\overline{x} \in \mathfrak{X}$. The tangent space is spanned by the basis vectors

$$\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \ldots, \frac{\partial}{\partial x_j^{(k)}}, \ldots \tag{3.64}$$

Now we introduce the *trivial Cartan field* of $\mathfrak{X}$

$$\tau_{\mathfrak{X}} = \sum_{i=1}^{n} \sum_{j \geq 0} x_i^{(j+1)} \frac{\partial}{\partial x_i^{(j)}} \tag{3.65}$$

which allows for any arbitrary meromorphic function $\phi(\overline{x})$ the identification

$$L_{\tau_{\mathfrak{X}}} \phi(\overline{x}) = \sum_{i=1}^{n} \sum_{j \geq 0} x_i^{(j+1)} \frac{\partial \phi(\overline{x})}{\partial x_i^{(j)}} = \frac{d}{dt} \phi(\overline{x}). \tag{3.66}$$

$L_{\tau_{\mathfrak{X}}}\phi(\overline{x})$ is also called *Lie derivative* of $\phi(\overline{x})$ along the trivial Cartan field $\tau_{\mathfrak{X}}$[13]. Now we are able to define the system class of regular implicit systems:

Definition 18 ([20]). *The triple $(\mathfrak{X}, \tau_{\mathfrak{X}}, F)$ consisting of a prolongation $\mathfrak{X} = X \times \mathbb{R}^n_\infty$, the associated trivial Cartan field $\tau_{\mathfrak{X}}$ and a meromorphic function F which maps TX to $\mathbb{R}^{n-m}$ with $\operatorname{rank}\left(\frac{\partial F}{\partial \dot{x}}\right) = n - m$ is called **regular implicit system**.*

If the set of solutions of the system (3.61) is denoted by the manifold

$$\mathfrak{X}_0 = \{\overline{x} | L^k_{\tau_{\mathfrak{X}}} F = 0 \ \ \forall\, k \geq 0\}, \tag{3.67}$$

Lie-Bäcklund equivalence can be defined by:

Definition 19 ([22]). *Two regular implicit systems $(\mathfrak{X}, \tau_{\mathfrak{X}}, F)$ and $(\mathfrak{Y}, \tau_{\mathfrak{Y}}, G)$ are called **Lie-Bäcklund equivalent** at the pair of points $(\overline{x}_0, \overline{y}_0) \in \mathfrak{X}_0 \times \mathfrak{Y}_0$ if and only if the assertions*

i) there exist a neighborhood $\mathcal{X}_0$ of $\overline{x}_0$ in $\mathfrak{X}_0$ and a neighborhood $\mathcal{Y}_0$ of $\overline{y}_0$ in $\mathfrak{Y}_0$ and a meromorphic map Φ such that

$$\Phi = (\varphi_0, \varphi_1, \ldots) : \mathcal{Y}_0 \to \mathcal{X}_0 \ \text{with}\ \Phi(\overline{y}_0) = \overline{x}_0 \tag{3.68}$$

ii) the associated trivial Cartan fields are Φ-related, i.e. $\Phi_* \tau_{\mathfrak{Y}} = \tau_{\mathfrak{X}}$

iii) there exists a one-to-one meromorphic map Ψ such that

$$\Psi = (\psi_0, \psi_1, \ldots) : \mathcal{X}_0 \to \mathcal{Y}_0 \ \text{with}\ \Psi(\overline{x}_0) = \overline{y}_0 \tag{3.69}$$

iv) the associated trivial Cartan fields are Ψ-related, i.e. $\Psi_* \tau_{\mathfrak{X}} = \tau_{\mathfrak{Y}}$

*are satisfied. In this case the meromorphic map Φ called **Lie-Bäcklund isomorphism** with the inverse Ψ.*
*The two systems $(\mathfrak{X}, \tau_{\mathfrak{X}}, F)$ and $(\mathfrak{Y}, \tau_{\mathfrak{Y}}, G)$ are called **locally Lie-Bäcklund equivalent** if they are Lie-Bäcklund equivalent at every pair of points $(\overline{x}, \Psi(\overline{x})) = (\Phi(\overline{y}), \overline{y})$ of an open dense subset $\mathcal{Z}$ of $\mathfrak{X}_0 \times \mathfrak{Y}_0$, with Φ and Ψ mutually inverse Lie-Bäcklund isomorphisms on $\mathcal{Z}$.*

Now we are able to introduce differential flatness:

Definition 20 ([15]). *The regular implicit system $(\mathfrak{X}, \tau_{\mathfrak{X}}, F)$ is called **differentially flat** around $(\overline{x}_0, \overline{y}_0) \in \mathfrak{X}_0 \times \mathbb{R}^m_\infty$ if and only if the system is Lie-Bäcklund equivalent to the trivial implicit system $(\mathbb{R}^m_\infty, \tau_{\mathbb{R}^m_\infty}, 0)$ in a neighborhood of $(\overline{x}_0, \overline{y}_0)$.*
*It is called **differentially flat** if it is differentially flat around every pair of points $(\overline{x}_0, \overline{y}_0)$ of an open dense subset of $\mathfrak{X}_0 \times \mathbb{R}^m_\infty$.*

[13] The k'th *Lie derivative* is denoted by $L^k_{\tau_{\mathfrak{X}}}\phi(\overline{x}) = \frac{d^k}{dt^k}\phi(\overline{x})$

3.3.3 Introduction of Differential Forms and Operators

Based on the tangent space $T_{\overline{x}}\mathfrak{X}$ the cotangent space $T^*_{\overline{x}}\mathfrak{X}$ can be introduced. It is spanned by the basis vectors

$$dx_1, dx_2, \ldots, dx_n, d\dot{x}_1, d\dot{x}_2, \ldots, d\dot{x}_n, \ldots, dx_i^{(j)}, \ldots \tag{3.70}$$

which satisfy

$$< dx_i^{(j)}, \frac{\partial}{\partial x_k^{(l)}} >= \delta_{ik}\delta_{jl} \text{ with } \delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{if } i \neq k \end{cases} \tag{3.71}$$

Because of that $T^*_{\overline{x}}\mathfrak{X}$ is also called *dual space* of $T_{\overline{x}}\mathfrak{X}$. A finite linear combination of basis vectors of the cotangent space is called *1-form*:

$$\omega = \sum_{j=0}^{\infty} \sum_{i=1}^{n} \omega_{i,j}(\overline{x}) dx_i^{(j)} \tag{3.72}$$

with the coefficients $\omega_{i,j}$ as meromorphic functions of the state and its derivatives with respect to time. The space of 1-forms is also denoted by $\Lambda^1(\mathfrak{X})$. By using the so called *wedge product* it is possible to introduce the space of p-forms $\Lambda^p(\mathfrak{X})$ which is spanned by the basis vectors

$$dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)}. \tag{3.73}$$

The properties of the wedge product $\wedge$ applied to differentials are

$$\omega \wedge \theta = (-1)^{p \cdot q} \theta \wedge \omega, \;\; \omega \in \Lambda^p(\mathfrak{X}), \; \theta \in \Lambda^q(\mathfrak{X}) \tag{3.74}$$

and

$$\omega \wedge \omega \equiv 0 \;\; \forall \omega \in \Lambda^1(\mathfrak{X}). \tag{3.75}$$

Therefore, a p-form $\omega \in \Lambda^p(\mathfrak{X})$ is given by

$$\omega = \sum_{i_1, j_1, \ldots, i_p, j_p} \omega_{i_1, j_1, \ldots, i_p, j_p}(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)}. \tag{3.76}$$

The derivative with respect to time of a differential form $\omega \in \Lambda^p(\mathfrak{X})$ is given by the Leibniz-rule:

$$\begin{aligned} \frac{d}{dt}\omega &= \frac{d}{dt} \sum_{i_1, j_1, \ldots, i_p, j_p} \omega_{i_1, j_1, \ldots, i_p, j_p}(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \\ &= \sum_{i_1, j_1, \ldots, i_p, j_p} \Big(\dot{\omega}_{i_1, j_1, \ldots, i_p, j_p}(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \\ &\quad + \omega_{i_1, j_1, \ldots, i_p, j_p}(\overline{x}) \frac{d}{dt} \left(dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \right) \Big) \end{aligned} \tag{3.77}$$

with

$$\begin{aligned} & \frac{d}{dt}\left(dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)}\right) \\ = \; & d\dot{x}_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} + dx_{i_1}^{(j_1)} \wedge d\dot{x}_{i_2}^{(j_2)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} + \ldots \\ & + dx_{i_1}^{(j_1)} \wedge \ldots \wedge d\dot{x}_{i_p}^{(j_p)} \end{aligned} \tag{3.78}$$

(see also [22]).
Furthermore, the operator d shall be introduced whose main ability is to transform p-forms into (p+1)-forms. The operator d is also called *exterior derivative*. It has the property[14]:

$$d(\omega \wedge \theta) = d\omega \wedge \theta + (-1)^p \omega \wedge d\theta \tag{3.79}$$

with $\omega \in \Lambda^p(\mathfrak{X})$ and $\theta \in \Lambda^q(\mathfrak{X})$. We also have $\forall \omega \in \Lambda^p(\mathfrak{X})$

$$d(d(\omega)) \equiv 0. \tag{3.80}$$

Computing the exterior derivative of an arbitrary 1-form $\omega \in \Lambda^1(\mathfrak{X})$ yields

$$\begin{aligned} d(\omega) &= d\left(\sum_{j=0}^{\infty}\sum_{i=1}^{n} \omega_{i,j}(\overline{x}) dx_i^{(j)}\right) \\ &= \sum_{j=0}^{\infty}\sum_{i=1}^{n} d(\omega_{i,j}(\overline{x})) \wedge dx_i^{(j)} \end{aligned} \tag{3.81}$$

with

$$d(\omega_{i,j}(\overline{x})) = \sum_{l=0}^{\infty}\sum_{k=1}^{n} \frac{\partial \omega_{i,j}(\overline{x})}{\partial x_k^{(l)}} dx_k^{(l)} \tag{3.82}$$

as the exterior derivative of a meromorphic function $\omega_{i,j}(\overline{x})$. Together with the *anti-derivation* property (3.79), the concrete exterior derivative of p-forms can be computed.
The operator d can now be used to define the operator $\mu \in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$ which transforms p-Forms into (p+q)-forms:

$$\mu = \mu_0 \wedge + \mu_1 \wedge \frac{d}{dt} + \ldots + \mu_n \wedge \frac{d^n}{dt^n} \tag{3.83}$$

with $\mu_i \in \Lambda^q(\mathfrak{X}), i = 0..n$. If we apply the operator $\mu \in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$ to a differential form $\omega \in \Lambda^p(\mathfrak{X})$, we will get

$$\mu\omega = \mu_0 \wedge \omega + \mu_1 \wedge \dot{\omega} + \ldots + \mu_n \wedge \omega^{(n)} \in \Lambda^{p+q}(\mathfrak{X}). \tag{3.84}$$

[14] This property is also known as *anti-derivation*-property.

The exterior derivative mapping d can be extended to a mapping $\mathfrak{d}$ to the class of previous operators $\mu \in \mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^{p+q}\left(\mathfrak{X}\right)\right)$ as follows:

$$\mathfrak{d}(\mu) = d\mu_0 \wedge + d\mu_1 \wedge \frac{d}{dt} + \ldots + d\mu_n \wedge \frac{d^n}{dt^n} \tag{3.85}$$

with $\mu_i \in \Lambda^q\left(\mathfrak{X}\right), i = 0..n$. Similar to operator d, we also have

$$\mathfrak{d}(\mathfrak{d}(\mu)) \equiv 0, \ \forall \mu \in \mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^{p+q}\left(\mathfrak{X}\right)\right). \tag{3.86}$$

The time derivative of an operator $\mu \in \mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^{p+q}\left(\mathfrak{X}\right)\right)$ results from the Leibniz-rule:

$$\frac{d}{dt}\mu = \sum_{i=0}^{n} \left(\frac{d}{dt}(\mu_i) \wedge \frac{d^i}{dt^i} + \mu_i \wedge \frac{d^{i+1}}{dt^{i+1}} \right). \tag{3.87}$$

3.3.4 A First Characterization of Flatness

For further evaluation of necessary and sufficient conditions for differential flatness of nonlinear systems, we use the following theorem:

Theorem 11 ([22]). *The regular implicit system $(\mathfrak{X}, \tau_{\mathfrak{X}}, F)$ is locally flat in a neighborhood of the pair of points $(\overline{x}_0, \overline{y}_0) \in \mathfrak{X}_0 \times \mathbb{R}^m_\infty$ if and only if there exists a local invertible meromorphic function $\Phi : \mathbb{R}^m_\infty \to \mathfrak{X}_0$ with $\Phi(\overline{y}_0) = \overline{x}_0$, whose **pull-back** $\Phi^* : T^*_{\overline{x}}\mathfrak{X} \to T^*_{\overline{y}}\mathbb{R}^m_\infty$ satisfies the equation*

$$\Phi^*\left(dF\right) = 0. \tag{3.88}$$

The differential $dF \in \Lambda^1\left(\mathfrak{X}\right)$ in (3.88) is, referring to [20], given by

$$dF = \frac{\partial F}{\partial x} dx + \frac{\partial F}{\partial \dot{x}} d\dot{x} \tag{3.89}$$

with $dx = \begin{pmatrix} dx_1 & \ldots & dx_n \end{pmatrix}^T$ and $d\dot{x} = \begin{pmatrix} d\dot{x}_1 & \ldots & d\dot{x}_n \end{pmatrix}^T$.
By introducing the skew polynomial matrix $P_F \in \mathcal{K}\left[\frac{d}{dt}\right]^{n-m \times n}$ such that

$$P_F = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial \dot{x}} \frac{d}{dt} \tag{3.90}$$

with $\mathcal{K}$ the field of meromorphic functions over $\mathfrak{X}$ (see (3.63)), we are able to rewrite (3.89):

$$dF = P_F dx. \tag{3.91}$$

By setting (3.91) $= 0$, we obtain the *variational system* with the system dynamics P_F:

$$P_F dx = 0. \tag{3.92}$$

We deduce from equation (3.68) that for the Lie-Bäcklund isomorphism Φ in Theorem 11 we have

$$\begin{aligned}\Phi(\overline{y}) &= (\varphi_0(\overline{y}), \varphi_1(\overline{y}), \varphi_2(\overline{y}), \ldots) \\ &= (\varphi_0(\overline{y}), \dot{\varphi}_0(\overline{y}), \ddot{\varphi}_0(\overline{y}), \ldots) = (x, \dot{x}, \ddot{x}, \ldots) = \overline{x}.\end{aligned} \tag{3.93}$$

We can show that we have ([22])

$$\Phi^* (dF) = \Phi^* (P_F dx) = P_F P_{\varphi_0} dy \tag{3.94}$$

with the skew polynomial matrix

$$P_{\varphi_0} = \sum_{j \geq 0} \frac{\partial \varphi_0}{\partial y^{(j)}} \frac{d^j}{dt^j}. \tag{3.95}$$

Let us assume that y is a flat output. Then, $\Phi^* (dF) = 0$ is equivalent to

$$P_F P_{\varphi_0} = 0. \tag{3.96}$$

To determine a function φ_0, whose pull-back satisfies (3.88), we first determine all unimodular (i.e. invertible) skew polynomial matrices $Q \in \mathcal{K}\left[\frac{d}{dt}\right]^{n \times n}$ which satisfy

$$P_F Q = 0. \tag{3.97}$$

Afterwards we evaluate if there exists a solution for Q which also satisfies the equation

$$Q dy = d\varphi_0. \tag{3.98}$$

I.e. it has to be analyzed if Qdy is the differential of a function φ_0. If that is true, φ_0 can be computed by the integration of Qdy. Also in this case, the equations (3.88) and (3.94) are satisfied with $P_{\varphi_0} = Q$.

3.3.5 Construction of a Flat Output of the Variational System

Remark: The necessary condition for differential flatness of the nonlinear system (3.61) is the differential flatness of the variational system (3.92) (see [22]).
This leads to the following approach: First we construct an integrable flat output ω of the variational system (3.92). Then we get the flat output of the nonlinear system (3.61) by integrating ω.

Theorem 12 ([22]). *The variational system $P_F dx = 0$ with $P_F \in \mathcal{K}\left[\frac{d}{dt}\right]^{n-m \times n}$ is called differentially flat if and only if P_F is hyper-regular. I.e. there has to exist an unimodular skew polynomial matrix $\widetilde{Q} \in \mathcal{K}\left[\frac{d}{dt}\right]^{n \times n}$ such that*

$$P_F \widetilde{Q} = \begin{pmatrix} I_{n-m} & 0_{n-m \times m} \end{pmatrix}. \tag{3.99}$$

Both the construction of a flat output ω of the variational system and the expression of ω using dx are executed similar to the linear case in section 3.2.1: So, in this case we have to compute $P_{\varphi_0} = Q$ such that (3.96) is satisfied. This is equivalent to (3.99) with

$$Q = \widetilde{Q} \begin{pmatrix} 0_{n-m\times m} \\ I_m \end{pmatrix}. \tag{3.100}$$

Then, a minimal basis decomposition of Q yields the unimodular skew polynomial matrix $\widetilde{P} \in \mathcal{K}\left[\frac{d}{dt}\right]^{n\times n}$ such that

$$\widetilde{P}Q = \begin{pmatrix} I_m \\ 0_{n-m\times m} \end{pmatrix}. \tag{3.101}$$

It can be shown (see [6]) that a flat output of the variational system is always given by

$$\omega = \underbrace{\begin{pmatrix} I_m & 0_{m\times n-m} \end{pmatrix} \widetilde{P}}_{P} dx. \tag{3.102}$$

To express the differential dx in coordinates of ω we can use (see [22, 6])

$$dx = Q\omega. \tag{3.103}$$

Like in the linear case, we can avoid the second minimal basis decomposition by using (3.45) – (3.46). This yields the following theorem:

Theorem 13. *If the matrix $P_F \in \mathcal{K}\left[\frac{d}{dt}\right]^{n-m\times n}$ of the variational system $P_F dx = 0$ is hyper-regular, $P \in \mathcal{K}\left[\frac{d}{dt}\right]^{m\times n}$ can be computed by the equation*

$$P = \begin{pmatrix} 0_{m\times n-m} & I_m \end{pmatrix} \widetilde{Q}^{-1}. \tag{3.104}$$

Remark: ω is interpreted as a flat output of the variational system. P and Q are the corresponding defining operators. Therefore, the algorithms in the time-varying linear case apply.

3.3.6 Integrability of the Variational Flat Output

Referring to [22], we have a link between the flat output ω of the variational system (3.92) and the flat output y of the nonlinear system (3.61):
If ω is a closed form, i.e.

$$d\omega = 0, \tag{3.105}$$

a flat output of the nonlinear system will be given by the integration of

$$\omega = Pdx. \tag{3.106}$$

If the condition (3.105) is not fulfilled, we will have to find an unimodular skew polynomial matrix $M \in \mathcal{K}\left[\frac{d}{dt}\right]^{m\times m}$ such that

$$d\left(M\omega\right) = 0. \tag{3.107}$$

It can be shown that $M\omega$ is also a flat output of the variational system (3.92) (see [22, 6]). Therefore, we can use the following theorem:

Theorem 14 ([21]). *The system $(\mathfrak{X}, \tau_{\mathfrak{X}}, F)$ is flat, if and only if there exist an operator $\mu \in \mathcal{L}\left(\Lambda^1\left(\mathfrak{X}\right), \Lambda^2\left(\mathfrak{X}\right)\right)^{m\times m}$ and an unimodular matrix $M \in \mathcal{K}\left[\frac{d}{dt}\right]^{m\times m}$ such that:*

$$d\omega = \mu\omega, \qquad \mathfrak{d}\left(\mu\right) = \mu\mu, \qquad \mathfrak{d}\left(M\right) = -M\mu. \tag{3.108}$$

If μ and M exist, a flat output y can be obtained by integrating $dy = M\omega$.

In order to find suitable matrices $\mu \in \mathcal{L}\left(\Lambda^1\left(\mathfrak{X}\right), \Lambda^2\left(\mathfrak{X}\right)\right)^{m\times m}$ and $M \in \mathcal{K}\left[\frac{d}{dt}\right]^{m\times m}$ which satisfies (3.108), the following approach from [21] can be used: Assuming the entries of $M \in \mathcal{K}\left[\frac{d}{dt}\right]^{m\times m}$ are elements in $\mathcal{L}\left(\Lambda^1\left(\mathfrak{X}\right), \Lambda^1\left(\mathfrak{X}\right)\right)$, the exterior derivative of (3.107) combined with the (for the operator $\mathfrak{d}$ extended) anti-derivation-property (3.79) is given by

$$\begin{aligned} & \mathfrak{d}\left(M\right)\omega + Md\omega = 0 \\ \Leftrightarrow \quad & d\omega = -M^{-1}\mathfrak{d}\left(M\right)\omega. \end{aligned} \tag{3.109}$$

Now, if we construct an operator matrix $\mu \in \mathcal{L}\left(\Lambda^1\left(\mathfrak{X}\right), \Lambda^2\left(\mathfrak{X}\right)\right)^{m\times m}$ such that

$$\mu = -M^{-1}\mathfrak{d}\left(M\right), \tag{3.110}$$

μ and M satisfy the conditions $d\omega = \mu\omega$ and $\mathfrak{d}\left(M\right) = -M\mu$ because of their structure. In addition, we have to have $\mathfrak{d}\left(\mathfrak{d}\left(M\right)\right) = 0$, i.e.

$$\begin{aligned} \mathfrak{d}\left(\mathfrak{d}\left(M\right)\right) &= \mathfrak{d}\left(-M\mu\right) \\ 0 &= -\mathfrak{d}\left(M\right)\mu - M\mathfrak{d}\left(\mu\right) \\ M\mathfrak{d}\left(\mu\right) &= -\mathfrak{d}\left(M\right)\mu \\ \Rightarrow \mathfrak{d}\left(\mu\right) &= -M^{-1}\mathfrak{d}\left(M\right)\mu = \mu\mu. \end{aligned} \tag{3.111}$$

Thus also the second condition, which has to be satisfied by the operators μ and M, can be derived from (3.111).

3.3.7 A Sequential Procedure

This yields to the following algorithm for the determination of a flat output of the implicit nonlinear system (3.61), according to [6]:

1. Compute $P_F = \frac{\partial F}{\partial x} + \frac{\partial F}{\partial \dot{x}}\frac{d}{dt}$.

2. Algebraic flatness analysis of P_F: Compute $\omega = Pdx$ according to the equations (3.99) – (3.102).
3. Compute μ such that $\mu\omega = d\omega$.
4. Determine all μ which also satisfy $\mathfrak{d}(\mu) = \mu^2$.
5. Compute M such that $\mathfrak{d}(M) = -M\mu$.
6. Discard all non-unimodular matrices M. A flat output can be obtained by integrating $dy = M\omega$.
7. If no suitable unimodular matrix M exists, the system is not flat.

This algorithm has to be executed in several iteration steps. In each step the degree in $\frac{d}{dt}$ of the operators μ and M (and therefore the number of degrees of freedom) will be increased by one until a flat output was found. It is still uncertain whether an upper limit for the degree in $\frac{d}{dt}$ of the operators μ and M exists or not (and therefore it remains unclear whether the algorithm terminates automatically after a certain number of steps). This question is currently a subject of research ([22]).

Chapter 4

Developing a Data Structure for Linear Systems

In section 3.2 we described the mathematical operations we need for the flatness determination in case of linear systems (with and without delays). At the beginning of this chapter, we will analyze the mathematical structures which are needed for the flatness determination. In a second step, we are going to define criteria for the decision on the implementation approach. Furthermore, we will discuss practical possibilities of the implementation. Finally, we are going to choose an appropriate data structure based on the previously defined criteria.

4.1 Analysis of the Mathematical Structures

For the linear flatness determination the decision was made to implement only a single toolbox which is able to handle the determination of differential and π-flatness of linear systems. This requires common data structures in order to be compatible between the different features. Since linear time-varying systems with delays include linear time-varying systems and linear time-invariant systems we can use the same data structure to describe them.
Let us start with data structures that suit linear time-invariant systems and then extend them for the other system classes: Considering the computations made in section 3.2.1, it becomes evident that we need a technical representation of operator matrices $M \in \mathbb{R}\left[\frac{d}{dt}\right]^{r\times s}$ and therefore of operators $p \in \mathbb{R}\left[\frac{d}{dt}\right]$ which are of the form:

$$p = \sum_{i=0}^{N} p_i \frac{d^i}{dt^i},\ p_i \in \mathbb{R},\ \deg(p) = N \in \mathbb{N}_0 \tag{4.1}$$

with $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$.
In order to consider linear time-varying systems from section 3.2.2 as well, we

simply enhance the representation of the operators (4.1) to

$$p(t) = \sum_{i=0}^{N} p_i(t) \frac{d^i}{dt^i},\ p_i(t) \in \mathcal{K},\ \deg(p) = N \in \mathbb{N}_0. \tag{4.2}$$

We consider $\mathcal{K}$ to be the field of meromorphic functions. At this point, we decide to implement only a representation of ordinary functions in t and not of *Laurent* series.
To enhance the data structure in order to support also delays and predictions, we have to extend the polynomials to the ring $\mathcal{K}(\delta)\left[\frac{d}{dt}\right]$. I.e. we have polynomials of the form:

$$p(t) = \sum_{i=0}^{N} b_i(t)^{-1} a_i(t) \frac{d^i}{dt^i},\ \deg(p) = N \in \mathbb{N}_0, \tag{4.3}$$

while the polynomials $a_i(t) \in \mathcal{K}[\delta]$, $b_i(t) \in \mathcal{K}[\delta] \setminus \{0\}$ are of the form:

$$a_i(t) = \sum_{j=0}^{R_i} a_{i,j}(t)\delta^j,\ a_{i,j}(t) \in \mathcal{K},\ \deg(a_i) = R_i \in \mathbb{N}_0, \tag{4.4}$$

$$b_i(t) = \sum_{j=0}^{S_i} b_{i,j}(t)\delta^j,\ b_{i,j}(t) \in \mathcal{K},\ \deg(b_i) = S_i \in \mathbb{N}_0. \tag{4.5}$$

Additionally, to compute the minimal basis decomposition mentioned in section 3.1.6, we need a representation of matrices of left fractions, too.
In order to separate the data structure into smaller parts, a possible solution is to distinguish:

- polynomial $a_i(t)$ resp. $b_i(t) \in \mathcal{K}[\delta]$
- left fraction $b_i(t)^{-1}a_i(t) \in \mathcal{K}(\delta)$
- matrix $\in \mathcal{K}(\delta)^{n\times m}$
- polynomial $p(t) \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$
- matrix $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times m}$

Therefore, a target data structure for the representation of polynomials $\in \mathcal{K}[\delta]$ needs to provide the information:

i) the degree of the polynomial in δ

ii) a mapping from a degree to the corresponding coefficient of the polynomial

To describe left fractions $\in \mathcal{K}(\delta)$ we simply need the information:

i) denominator

ii) numerator

Polynomials $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ can be expressed by the information:

i) the degree of the polynomial in $\frac{d}{dt}$

ii) a mapping from a degree to the corresponding coefficient of the polynomial

For matrices $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times m}$ and $\in \mathcal{K}(\delta)^{n\times m}$ we need:

i) the number of rows

ii) the number of columns

iii) a mapping from the pair (row, column) to the corresponding entry

4.2 Criteria for the Implementation Approach

The most important property which shall be fulfilled by the toolbox is to have good performance characteristics. In order to achieve this, the data structures have to provide a low requirement for both computation time and memory.
Furthermore, we want to rely on *Maple*-own data structures if sustainable referring to its performance. Thereby, we are able to use *Maple*-own methods for handling the data structures, so we do not need to implement everything new.

4.3 Practical Possibilities in Terms of Implementation

4.3.1 Available Basic Data Structures in *Maple*

Since we want to rely on existing basic data structures in *Maple*, let us take a look at those at the beginning of this section:
Generally, we have two groups of data structures in *Maple* (see [26]): mutable and immutable data structures. In case of mutable data structures, we are able to change the attributes of an instance after its instantiation. In case of immutable data structures, this is not possible. Instead, a copy of the original instance, but with changed attribute values, will be created automatically.

Set

Sets belong to the immutable data structures in *Maple*. The elements in a set

are unique. Doubled entries will be deleted[1]. Furthermore, the elements in sets have no specified order and thus they cannot be distinctly called by their index[2]. Because of that they only come into consideration for very specific data structures[3].

Listing 4.1: Creating a set

```
> example_set_1 := {a, b, c};
          example_set_1 := {a, b, c}
> example_set_2 := {c, b, b, a, c};
          example_set_2 := {a, b, c}
```

Since doubled entries are going to be deleted, the two sets above are completely identical.

List

Lists also belong to the immutable data structures of *Maple*. But unlike sets, their elements are not unique and have a specific order. Thus, they have an unique index and can be accessed by it.

Listing 4.2: Creating a list

```
> example_list := [a, b, c, c];
          example_list := [a, b, c, c]
#Get the number of elements
> nops(example_list);
          4
```

Maple offers a lot of methods for the manipulation of lists and sets. So working with them can be easily handled without implementing additional methods.

Table

Tables belong to the mutable data structures. That means that if we change the attribute values of a table, *Maple* will not create a whole new table, but instead the attributes of this very instance will be changed.
Tables consist of key-value pairs, i.e. there exists at most one value to each key. The data type of the key and the value may be different. Thus, a table correlates with a mapping from keys to values.

Listing 4.3: Creating a table

```
> example_table := table([1 = a, 2 = b, 3 = c, 4 = c]);
#Accessing the value to the key "2"
> example_table[2];
          b
```

[1] Since there exists no automatic simplification of mathematical terms in *Maple*, it may be possible that there occur doubled entries in sets, so we have to simplify those by our own, to make sure that all elements in a set are unique.

[2] In fact, we can use the index to get elements of the set, but since *Maple* does the internal sorting of the elements we cannot be sure which element we get.

[3] Note that even sums cannot be represented by a *Maple*-own set, because an unsimplified sum may contain the very same element several times. Automatically deleting those would corrupt the sum!

Array

Arrays are multidimensional data fields which can be filled with arbitrary data types. They are implemented as rtables (a more general data structure in *Maple*), just like the *Maple* data structures Matrix and Vector. They belong to the mutable data structures, i.e. changing the entries does not force *Maple* to create a new instance.

Listing 4.4: Creating an Array

```
#One-dimensional Array
> example_array := Array([a, b, c, c]);
#Accessing the second value of the array
> example_array[2];
            b
#Multidimensional Array with two rows and four columns
> example_array := Array([[a, b, c, c], [d, e, f, f]]);
#Accessing the value at (2,3) of the array
> example_array[2,3];
            f
```

Record

Records are data structures which consist of a specific number of named attributes. Those attributes have to be defined by the programmer and may not be changed after its instantiation. Records are mutable data structures. Their attribute values can be read, copied and changed using specific *Maple* commands.

Listing 4.5: Creating a Record

```
#Create a record with unassigned attributes
> example_record := Record('foo','bar');
#Set the attributes
> example_record:-foo := a;
> example_record:-bar := b;
#Get the value of attribute 'foo'
> example_record:-foo;
            a
```

Matrix and Vector

Although Matrix and Vector are not exactly basic data structures of *Maple*, we still want to describe them briefly at this point since we consider using them in our data structure. Both are mutable data structures and represent matrices and vectors in *Maple*. In *Maple* there are a lot of predefined methods for manipulating both Vectors and Matrices.

Remark: Matrix, Vector and Array share the same internal data structure rtable (see [26] for more details).

Listing 4.6: Creating a Matrix

```
> example_matrix := Matrix(3, 2, {
  (1,1) = a,
  (1,2) = b,
  (2,1) = c,
```

```
  (2,2) = d,
  (3,1) = e,
  (3,2) = f
  });
#Get the element at position (2,1)
> example_matrix[2,1];
            c
#Change the element at position (2,1)
> example_matrix[2,1] := x;
```

The Vector works similar to that. In addition, you can add an orientation of the Vector to specify it to be a row or column vector.

4.3.2 Possible Approaches for the Implementation

Looking at the data, which has to be stored in the new data types, and the possible basic data structures *Maple* already provides, many possible solutions come into consideration. We will now discuss a few different approaches for an implementation of the data structures mentioned in section 4.1.
Remark: Besides their internal implementation, both data structures list and Array provide similar features, therefore we will treat them interchangeable as long as we do not consider performance issues.

Polynomials $\in \mathcal{K}[\delta]$ and $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$

Since both data structures contain the same sort of information (the degree of the polynomial and a mapping from a degree to the corresponding coefficient), they can be expressed the same way in *Maple*:
1.) Using a table: Since a table is a mapping, we can simply map the degree to the corresponding coefficient, sparing all degrees whose coefficient is zero. The degree of such a polynomial representation can then be obtained by the highest occurring degree. Therefore, there is no need to store the degree separately. E.g. the polynomial $x\delta + y\delta^3$ could be stored as:

Listing 4.7: Polynomial as table

```
> poly := table([1 = x, 3 = y]);
> max(indices(poly, 'nolist'));
            3
```

2.) Using a list or Array: We can store a polynomial into a list resp. Array by simply storing the coefficient to the n'th degree at position $n+1$ and then filling the empty positions with zeros. The degree of the polynomial is then given by the number of entries minus 1. E.g. the polynomial $x\delta + y\delta^3$ could be stored as[4]:

Listing 4.8: Polynomial as list

```
> poly := [0, x, 0, y];
> nops(poly) - 1;
            3
```

[4]The representation of a polynomial as Array works similar.

Left fractions $\in \mathcal{K}(\delta)$

Since we only handle left fractions, we do not need to store the orientation (left or right) of the fraction, but only the denominator and numerator. A few possible approaches to store this information are:
1.) Using a list or Array: We can simply store denominator and numerator as two entries in a list resp. Array. The position of the numerator resp. denominator must be equal in all left fractions (e.g. denominator at position 1 and numerator at position 2):

Listing 4.9: Left fraction as list

```
> leftFrac := [denom, numer];
#Accessing the numerator
> leftFrac[2];
                numer
```

2.) Using a Record: Since every left fraction consists of a denominator and a numerator, it is possible to use a Record for this purpose:

Listing 4.10: Left fraction as Record

```
> leftFrac := Record('denominator','numerator');
> leftFrac:-denominator := denom;
> leftFrac:-numerator := numer;
#Accessing the numerator
> leftFrac:-numerator;
                numer
```

Matrices $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times m}$ and $\in \mathcal{K}(\delta)^{n\times m}$

Although a matrix can be seen as a mapping from the indexes to the entries, we decide at this point not to use a table, because handling this data structure would be impractical. This leaves over the following possibilities:
1.) Using a list or Array: We can use a multidimensional list resp. Array to store all elements of the matrix. The dimensions of the matrix can then be evaluated by the size of the list resp. Array.

Listing 4.11: Matrix as list

```
> matrix_32 := [[entry_11, entry_12], [entry_21, entry_22], [entry_31,
entry_32]];
#Number of rows:
> nops(matrix_32);
                3
#Number of columns:
> nops(matrix_32[1]);
                2
```

2.) Using the *Maple*-own Matrix: Since the Matrix allows us to insert every kind of data, we are able to use the *Maple*-own Matrix and all predefined methods for it, such as those from the LinearAlgebra package.
Remark: For vectors of polynomials or left fractions, we can use the *Maple*-own structures Matrix and Vector as well.

Listing 4.12: Matrix as Matrix

```
> matrix_32 := Matrix(3, 2, {
  (1,1) = entry_11,
  (1,2) = entry_12,
  (2,1) = entry_21,
  (2,2) = entry_22,
  (3,1) = entry_31,
  (3,2) = entry_32
  });
> LinearAlgebra[RowDimension](matrix_32);
            3
> LinearAlgebra[ColumnDimension](matrix_32);
            2
```

4.4 Choosing an Appropriate Data Structure in *Maple* Based on the Defined Criteria

First of all, we will determine how much time the different read and write accesses and the creation of the eligible data structures need in *Maple*, using a simple performance test. Based on the results, we are able to decide which data structures are suitable in order to achieve good performance characteristics. For this test, we use the following four terms as our test data:

$$a = \sin(x_1(t))x_2(t) \tag{4.6}$$
$$b = \tan(x_1(t)) \tag{4.7}$$
$$c = 2x_1(t)x_2(t) \tag{4.8}$$
$$d = \cos(x_2(t)) \tag{4.9}$$

During the test, we create 4 million instances of the eligible data structures, which are always filled with these four attributes of our test data but in different order. Afterwards, we select every attribute of one instance of the data structure 1 million times and overwrite every attribute of one instance 1 million times. This yields the result:

Data structure	Reading	Writing	Creating
Set	1,747 sec	-[5]	10,514 sec
List	1,810 sec	4,103 sec	9,797 sec
Table	2,231 sec	2,730 sec	43,727 sec
Array	0,546 sec	2,902 sec	16,846 sec
Record	2,121 sec	2,777 sec	17,488 sec

From this result it can be derived that tables, Arrays and Records are not applicable to use them in a basic data structure. The reason for this assertion is that we have to create a lot of instances of the data structures in order

[5] Since there is no actual way to override specific elements in sets, this is not comparable with the other structures.

to implement the required algorithms, thus causing an enormous amount of computation time.
Since sets do not have a specified order, manipulating a set while iterating over it could cause errors due to the internal sorting of *Maple*. So we determine at this point that we focus on lists.
Matrices and Vectors have not been evaluated in this performance test, due to the fact that they are not directly comparable to the other data structures (except for multidimensional Arrays). In fact, we will use many matrix and vector operations in the toolbox. Nevertheless, the number of actual matrices and vectors which will be created is far from being as high as the number of polynomials, etc. So, we decide at this point to use the *Maple*-own Matrix and Vector structures to be able to use the already defined methods for matrix and vector manipulations.
Now we are able to define the concrete implementation of the data structures which have been determined in section 4.1. To define new data types in *Maple*, we can use the type command[6]:

Listing 4.13: Usage of type for type definition

```
> `type/<NameOfDataType>` := <FormOfDataType>;
```

This leads to the following list of new data types:

Polynomial $\in \mathcal{K}[\delta]$

Data type name: DelayPolynomial

Listing 4.14: Definition of DelayPolynomial in *Maple*

```
> `type/DelayPolynomial` := list(Not(list));
```

Examples:

Math. expression	DelayPolynomial
$x_1(t)\delta^2 + x_2(t)$	[x2D0T0[7], 0, x1D0T0]
$\ddot{\phi}(t+\tau)$	[phiD2P1]
$\delta^3 + \delta$	[0, 1, 0, 1]

Left fraction $\in \mathcal{K}(\delta)$

Data type name: LeftFraction

Listing 4.15: Definition of LeftFraction in *Maple*

```
> `type/LeftFraction` := [DelayPolynomial, DelayPolynomial];
```

[6] By using specific commands, it is possible to implement those data types in a special way in order to define them automatically when the user accesses the toolbox by using the with command of *Maple*. Thus, the user does not have to worry about the internal type definitions and may use them instantly like standard *Maple* types.

[7] To increase the performance, we do not use the data type function to express functions of time. However, to describe derivatives and delays (resp. predictions), a special transformation is used. A detailed description can be found in section 6.1.

Note: We define that we always keep the order [denominator, numerator].

Examples:

Math. expression	LeftFraction
$(\delta)^{-1}(1)$	[[0, 1], [1]]
$(x_1(t))^{-1}(\dot{x}_1(t)\delta^2)$	[[x1D0T0], [0, 0, x1D1T0]]
$(\ddot{\phi}(t+\tau))^{-1}(\delta)$	[[phiD2P1], [0, 1]]

Matrix $\in \mathcal{K}\langle\delta\rangle^{n\times m}$

Data type name: LeftFractionMatrix

Listing 4.16: Definition of LeftFractionMatrix in *Maple*

```
> `type/LeftFractionMatrix` := 'Matrix(LeftFraction)';
```

In this case, we omit examples since a LeftFractionMatrix is simply a *Maple*-own matrix with LeftFractions as entries.

Vector $\in \mathcal{K}\langle\delta\rangle^{n}$

Data type name: LeftFractionVector

Listing 4.17: Definition of LeftFractionVector in *Maple*

```
> `type/LeftFractionVector` := 'Vector(LeftFraction)';
```

In this case, we omit examples since a LeftFractionVector is simply a *Maple*-own vector with LeftFractions as entries. For the type definition it is not necessary to have a row or column vector. Both kinds are valid.

Polynomial $\in \mathcal{K}\langle\delta\rangle\left[\frac{d}{dt}\right]$

Data type name: OrePolynomial

Listing 4.18: Definition of OrePolynomial in *Maple*

```
> `type/OrePolynomial` := list(LeftFraction);
```

Examples:

Math. expression	OrePolynomial
$(\delta)^{-1}s(t)\frac{d}{dt}$	[[[1], [0]], [[0, 1], [sD0T0]]]
$x_1(t)x_2(t)\frac{d^2}{dt^2}+x_1(t)\frac{d}{dt}+1$	[[[1], [1]], [[1], [x1D0T0]], [[1], [x1D0T0 * x2D0T0]]]
$x_1(t)\frac{d}{dt}+s(t)\delta^2-s(t)\delta$	[[[1], [0, -sD0T0, sD0T0]], [[1], [x1D0T0]]]

Matrix $\in \mathcal{K}\langle\delta\rangle\left[\frac{d}{dt}\right]^{n\times m}$

Data type name: OreMatrix

Listing 4.19: Definition of OreMatrix in *Maple*

```
> `type/OreMatrix` := 'Matrix(OrePolynomial)';
```

In this case, we omit examples since an OreMatrix is simply a *Maple*-own matrix with OrePolynomials as entries.

Vector $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^n$
Data type name: OreVector

Listing 4.20: Definition of OreVector in *Maple*

```
> 'type/OreVector' := 'Vector(OrePolynomial)';
```

In this case, we omit examples since an OreVector is simply a *Maple*-own vector with OrePolynomials as entries. For the type definition it is not necessary to have a row or column vector. Both kinds are valid.

Therefore, the final data structure of the linear toolbox is given by:

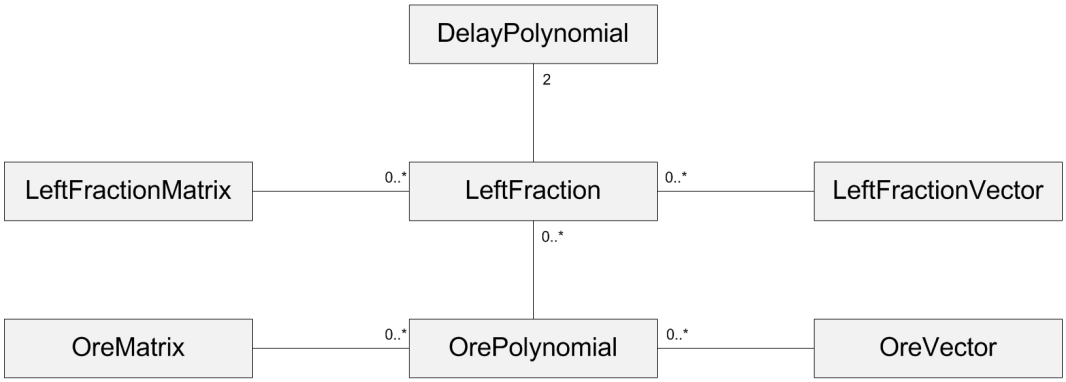

Figure 4.1: Abridged class diagram for DifferentialDelays

Chapter 5

Developing a Data Structure for Nonlinear Systems

We determined in chapter 4 which data structure we need in case of linear systems with and without delays. Now we want to evaluate in this chapter which data structure we require for nonlinear systems in order to implement methods for the algorithms from section 3.3.
Similar to chapter 4, we will analyze the mathematical structures we need to implement first. Furthermore, we are going to discuss practical approaches in terms of implementation. Then, we will choose an appropriate data structure for the toolbox. Finally, we are going to discuss the special case of the minimal basis decomposition.
Since we have already declared criteria for the evaluation of data structures in section 4.2, we will not repeat them in this chapter, but use the same criteria to choose among the different approaches.

5.1 Analysis of the Mathematical Structures

Like in the linear case, the decision was made to implement only a single toolbox with a common data structure which is able to handle the flatness determination for nonlinear systems. First, let us recall the data structure for nonlinear systems. The basic mathematical structures which we need are differential forms and operators. Furthermore, we need to handle matrices over these two structures.
A **general differential p-form** $\omega \in \Lambda^p(\mathfrak{X})$ is given by (3.76):

$$\omega = \sum_{i_1,j_1,\ldots,i_p,j_p} \omega_{i_1,j_1,\ldots,i_p,j_p}(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \tag{5.1}$$

with $\omega_{i_1,j_1,\ldots,i_p,j_p}$ be an arbitrary meromorphic function of the state and its derivatives with respect to time. An **operator** $\mu \in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$ is

given by extending a p-form with the differential operator $\frac{d}{dt}$ (see (3.83)):

$$\mu = \mu_0 \wedge + \mu_1 \wedge \frac{d}{dt} + \ldots + \mu_n \wedge \frac{d^n}{dt^n},\ \mu_i \in \Lambda^q(\mathfrak{X}). \tag{5.2}$$

In a first step, we split these two data structures into smaller parts in order to have a modular design of the data structures. This allows us to reuse several components and methods inside the toolbox. Both differential forms (6.12) and operators (5.2) are represented by sums of **monomial p-differential forms** (in case of differential forms) and **monomial operators** (in case of operators). A single monomial differential form is given by

$$\theta = \theta_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)}, \tag{5.3}$$

while a single monomial operator is given by

$$\nu \quad = \quad \nu_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_q}^{(j_q)} \wedge \frac{d^n}{dt^n} \tag{5.4}$$

with θ_c, ν_c be meromorphic functions of the state and its derivatives with respect to time.
Considering (5.3) and (5.4), it becomes evident that both have a specific number of **monomial differential forms** $dx_i^{(j)}$ in a fixed order[1]. This gives reason to treat these monomial differential forms as an own data structure. In fact, monomial differential forms may contain general p-forms (e.g. the monomial differential form $\omega \in \Lambda^3(\mathfrak{X})$) as well, so we need to create a very general data structure.
So, we distinguish the following data structures:

- monomial differential form
- monomial p-differential form $\in \Lambda^p(\mathfrak{X})$
- general p-form, i.e. $\in \Lambda^p(\mathfrak{X})$
- monomial operator $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$
- operator $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$
- matrix $\in \Lambda^p(\mathfrak{X})^{n \times m}$
- matrix $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))^{n \times m}$

The data structure for the representation of monomial differential forms needs to provide:

i) the information whether it is closed or not[2]

[1] This order has to be kept in the implementation since the wedge product is not commutative (see (3.74)).

[2] Let us recall that a differential ω with the property $d(\omega) = 0$ is called *closed*.

ii) the meromorphic function (resp. its time derivative)

iii) of how many 1-forms it consists

The first and third attributes are required due to the possibility of unknown, i.e. not completely substituted, differentials during the computation. A short example may illustrate this: The differential form $\omega \in \Lambda^3(\mathfrak{X})$ may be described in several ways using these unknown differentials:

$$\begin{aligned}
\omega &= dx_1 \wedge d\dot{x}_2 \wedge dx_2 \\
&= dx_1 \wedge \theta \text{ with } \theta = d\dot{x}_2 \wedge dx_2 \\
&= d\chi \wedge \theta, \text{ with } \chi = x_1,\ \theta = d\dot{x}_2 \wedge dx_2 \\
&= \chi \wedge \beta \wedge \gamma \text{ with } \chi = dx_1,\ \beta = d\dot{x}_2,\ \gamma = dx_2 \\
&= \vartheta \text{ with } \vartheta = dx_1 \wedge d\dot{x}_2 \wedge dx_2. \qquad (5.5)
\end{aligned}$$

The 'closed' attribute is primarily required to speed up the computations, because in case of an exterior derivative of a closed differential form (known or unknown) we can automatically consider the whole differential form as zero, because of (3.80).
However, the information about how many forms a differential consists of is very important and may not be omitted because of the permutation rule (3.74).

A representation of a monomial p-differential form must provide:

i) the coefficient of the monomial p-differential form (i.e. a meromorphic functions of the state and its derivatives with respect to time)

ii) an arbitrary number of monomial differential forms with a fixed order

Similarly, a representation of a monomial operator must provide:

i) the coefficient of the monomial operator (i.e. a meromorphic functions of the state and its derivatives with respect to time)

ii) an arbitrary number of monomial differential forms with a fixed order

iii) the degree of the operator in $\frac{d}{dt}$

Then, both general differential p-forms $\in \Lambda^p(\mathfrak{X})$ and operators $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$ can be expressed by:

i) all consisting summands (which are monomial p-differential forms resp. monomial operators)

For matrices $\in \Lambda^p(\mathfrak{X})^{n\times m}$ and $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))^{n\times m}$ we need:

i) the number of rows

ii) the number of columns

iii) a mapping from the pair (row, column) to the corresponding entry

5.2 Practical Possibilities in Terms of Implementation

Since we have already recalled suitable basic data structures of *Maple* in section 4.3.1, we will not repeat them in this section. Because of the thoughts given in section 4.4, we decide at this point to use lists instead of sets and Arrays. Since we will encounter sums of forms and operators, let us briefly discuss using actual sums (which is possible in *Maple*) in order to use the plus sign in *Maple* terms:

Listing 5.1: Using actual sums

```
> diffSum := diffForm_1 + diffForm_2;
```

In this case we simply overload the addition defined in *Maple* in order to extend it for the application on differential forms and operators. This can be done by using the overload-option:

Listing 5.2: Overloading '+'

```
arbitraryModule := module()
  option package;
  export `+`;
  `+` := proc(a::DiffForm, b::DiffForm) option overload;
    #Handle the addition
  end proc;
end module;
```

The method above allows us to simply write $\omega+\theta$ in our worksheets, but causes a tremendous decrease of computational performance. Obviously, we overload the plus-sign, causing *Maple* to check the parameters every time a plus-sign is parsed, even if none of the parameters is a DiffForm. Thus, we determine not to use overloading for sums. This yields the following data types in *Maple*:

Monomial differential form

In this case we just need to store three details. This can be done by the following approaches:
1.) Using a list: We can simply store the three details in a list by using fixed entries for each type of information:

Listing 5.3: Monomial differential form as list

```
> monoDiff := [isClosed, theFunction, numberOfForms];
#E.g. the monomial differential form dx_1 could be expressed by
> monoDiff := [true, x_1, 1];
#while an arbitrary unclosed 3-form would be expressed by
> monoDiff := [false, omega, 3];
```

2.) Using a Record: Since we have three separated attributes, we can store the information in a Record as well:

Listing 5.4: Monomial differential form as Record

```
> monoDiff := Record('isClosed', 'theFunction', 'numberOfForms');
> monoDiff:-isClosed := false;
```

```
> monoDiff:-theFunction := omega;
> monoDiff:-numberOfForms := 3;
```

Monomial p-differential form $\in \Lambda^p(\mathfrak{X})$

In this case, we have to keep in mind that we need a fixed order for the containing monomial differential forms. Therefore, a simple list may be the most appropriate representation for this. Nevertheless, we have these approaches for the monomial p-differential form:
1.) Using a list: If we consider fixed positions for the coefficient and the list of monomial differential forms, we may use a list:

Listing 5.5: Monomial p-differential form as list

```
> diffForm := [coefficient, [monomial differential forms...]];
```

2.) Using a Record: Another approach is to treat the coefficient and the list of monomial differential forms as attributes of a Record:

Listing 5.6: Monomial p-differential form as Record

```
> diffForm := Record('coefficient', 'monoDiffForms');
> diffForm:-coefficient := coeffz;
> diffForm:-monoDiffForms := [monomial differential forms...];
```

Monomial operator $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$

Assuming that a monomial operator is just a monomial p-differential form which has been extended with the operator $\frac{d}{dt}$, we can reuse the two approaches above and simply add the degree of the monomial operator in $\frac{d}{dt}$ to the existing data structure:
1.) Using a list:

Listing 5.7: Monomial operator as list

```
> operForm := [coefficient, [monomial differential forms...], degreeInDdt];
```

2.) Using a Record:

Listing 5.8: Monomial operator as Record

```
> operForm := Record('coefficient', 'monoDiffForms', 'degreeInDdt');
> operForm:-coefficient := coeffz;
> operForm:-monoDiffForms := [monomial differential forms...];
> operForm:-degreeInDdt := degree;
```

General p-forms $\in \Lambda^p(\mathfrak{X})$ and operators $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$

Since a general differential p-form (resp. an operator) is actually just a sum of monomial p-differential forms (resp. monomial operators), we can simply express these data structures by using a list[3]:

[3] Although a sum of monomial p-differential forms or monomial operators has no fixed order, we must not use the *Maple*-own type set since then doubled summands would be automatically deleted by *Maple*.

Listing 5.9: General p-form resp. operator as list

```
> diffSum := [diffForm_1, diffForm_2, ...];
> operSum := [operForm_1, operForm_2, ...];
```

Matrices $\in \Lambda^p(\mathfrak{X})^{n\times m}$ and $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))^{n\times m}$

Since we have already evaluated in chapter 4 that the *Maple*-own data types Matrix and Vector proved to be the most suitable representation for matrices and vectors, we decide at this point to rely on Matrix and Vector.

5.3 Choosing an Appropriate Data Structure in *Maple*

Based on the results from section 4.4, we maintain using lists as a basic data structure. This decision rewards us with both good performance characteristics and an easy handling of those structures using predefined methods of *Maple*. Furthermore, we obtain the new data types:

Monomial differential form

Data type name: MonoDiffForm

Listing 5.10: Definition of MonoDiffForm in *Maple*

```
> `type/MonoDiffForm` := [boolean, symbol, integer];
```

The first parameter describes whether the MonoDiffForm is closed. If the monomial differential form is not closed or if it is unknown whether the monomial differential form is closed, this parameter must be set to false.
The second parameter is the meromorphic function (resp. its time derivative) of the monomial differential form.
The third parameter describes how many 1-forms the monomial differential form consists of (e.g. 2 in case of a monomial differential form $\in \Lambda^2(\mathfrak{X})$).

Examples:

Math. expression	MonoDiffForm
$dx_1 \in \Lambda^1(\mathfrak{X})$	[true, x1D0[4], 1]
$\omega \in \Lambda^3(\mathfrak{X})$	[false, omegaD0, 3]
$d\dot{\theta} \in \Lambda^2(\mathfrak{X})$	[true, thetaD1, 2]

[4] To increase the performance, we do not use the data type function to express functions of time. However, a special transformation is used in order to describe derivatives. A detailed description can be found in section 6.1.

Monomial p-differential form

Data type name: DiffForm

Listing 5.11: Definition of DiffForm in *Maple*

```
> 'type/DiffForm' := [Not(list), list(MonoDiffForm)];
```

The first parameter is the coefficient of the monomial p-differential form. The second parameter is a list of the monomial differential forms of the monomial p-differential form.

Examples:

Math. expression	DiffForm
$\sin(x_1(t))dx_1 \wedge dx_2$	[sin(x1D0), [[true, x1D0, 1], [true, x2D0, 1]]]
$x_1(t)\omega$	[x1D0, [[false, omegaD0, 3]]]
$-dx_1 \wedge dx_2 \wedge d\theta$	[-1, [[true, x1D0, 1], [true, x2D0, 1], [true, thetaD0, 2]]]

with $\omega \in \Lambda^3(\mathfrak{X})$ and $\theta \in \Lambda^2(\mathfrak{X})$.

Monomial operator

Data type name: OperForm

Listing 5.12: Definition of OperForm in *Maple*

```
> 'type/OperForm' := [Not(list), list(MonoDiffForm), integer];
```

The first and second parameter are the same as in the DiffForm. The third parameter represents the degree of the operator $\frac{d}{dt}$ of the monomial operator.

Examples:

Math. expression	OperForm
$\sin(x_1(t))dx_1 \wedge dx_2 \wedge \frac{d}{dt}$	[sin(x1D0), [[true, x1D0, 1], [true, x2D0, 1]], 1]
$x_1(t)\omega\wedge$	[x1D0, [[false, omegaD0, 3]], 0]
$-dx_1 \wedge dx_2 \wedge d\theta \wedge \frac{d^3}{dt^3}$	[-1, [[true, x1D0, 1], [true, x2D0, 1], [true, thetaD0, 2]], 3]
$1 \wedge \frac{d}{dt}$	[1, [], 1]

with $\omega \in \Lambda^3(\mathfrak{X})$ and $\theta \in \Lambda^2(\mathfrak{X})$.

General p-form $\in \Lambda^p(\mathfrak{X})$

Data type name: DiffSum

Listing 5.13: Definition of DiffSum in *Maple*

```
> 'type/DiffSum' := And(list(DiffForm), Not([]));
```

In this case, we omit examples since a DiffSum is simply a *Maple*-own list of DiffForms. In case of an empty list, the toolbox will assert that it is no DiffSum. This is important, otherwise we would not be able to tell whether we have a

differential p-form or an operator.

Operator $\in \mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^{p+q}\left(\mathfrak{X}\right)\right)$

Data type name: OperSum

Listing 5.14: Definition of OperSum in *Maple*

```
> `type/OperSum` := And(list(OperForm), Not([])):
```

In this case we also omit examples since a OperSum is simply a *Maple*-own list of OperForms. Similar to the DiffSum, the toolbox will not accept empty lists as OperSums.

Matrix $\in \Lambda^p\left(\mathfrak{X}\right)^{n\times m}$

Data type name: DiffMatrix

Listing 5.15: Definition of DiffMatrix in *Maple*

```
> `type/DiffMatrix` := 'Matrix({DiffForm, DiffSum})':
```

In this case, we omit examples since a DiffMatrix is simply a *Maple*-own matrix with DiffForms and/or DiffSums as entries. So we do not need to transform single DiffForms into DiffSums.

Matrix $\in \mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^{p+q}\left(\mathfrak{X}\right)\right)^{n\times m}$

Data type name: OperMatrix

Listing 5.16: Definition of OperMatrix in *Maple*

```
> `type/OperMatrix` := 'Matrix({OperForm, OperSum})':
```

In this case, we omit examples since a OperMatrix is simply a *Maple*-own matrix with OperForms and/or OperSums as entries. So we do not need to transform single OperForms into OperSums.

Therefore, we obtain the data structure for handling differential forms and operators:

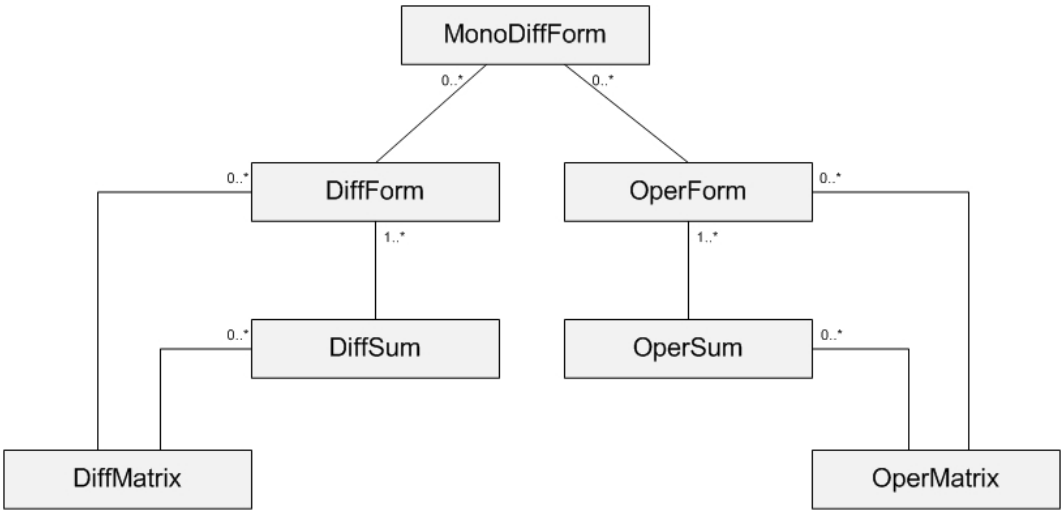

Figure 5.1: Abridged class diagram for DifferentialForms

5.4 Special Case: Minimal Basis Decomposition

The minimal basis decomposition from (3.99) and (3.101) is executed with matrices $\in \mathcal{K}\left[\frac{d}{dt}\right]^{n\times m}$. Using the data types defined in section 5.3 would entail the usage of OperMatrices without differentials to express matrices $\in \mathcal{K}\left[\frac{d}{dt}\right]^{n\times m}$. This data type however needs more memory than necessary. Moreover, the methods for handling operators take more computation time since they are implemented to handle the far more complex case in which we have $\mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^{p+q}\left(\mathfrak{X}\right)\right)$ with $q \geq 1$. Therefore, we decided to use a specialized data structure for this purpose.[5]
In case of the minimal basis decomposition we have matrices $\in \mathcal{K}\left[\frac{d}{dt}\right]^{n\times m}$, i.e. we have matrices over polynomials of the form:

$$p(t) = \sum_{i=0}^{N} c_i(t)\frac{d^i}{dt^i},\ c_i(t) \in \mathcal{K},\ \deg(p) = N \in \mathbb{N}_0. \tag{5.6}$$

I.e. we need to store the information:

i) the degree of the polynomial in $\frac{d}{dt}$

ii) a mapping from a degree to the corresponding coefficient of the polynomial

[5]This new data structure shall not directly be available for the user of the toolbox. Instead, the minimal basis decomposition shall convert between the 'regular' OperMatrices and our specialized data structure.

For matrices $\in \mathcal{K}\left[\frac{d}{dt}\right]^{n\times m}$ we need:

i) the number of rows

ii) the number of columns

iii) a mapping from the pair (row, column) to the corresponding polynomial

Like in case of polynomials $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ and matrices $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times m}$ (see chapter 4) we decide to implement the data types as lists, Matrices and Vectors:

Polynomial $\in \mathcal{K}\left[\frac{d}{dt}\right]$

Data type name: SkewPolynomial

Listing 5.17: Definition of SkewPolynomial in *Maple*

```
> `type/SkewPolynomial` := list(Not(list));
```

Examples:

Math. expression	SkewPolynomial
$\sin(x_1(t))+1$	[sin(x1D0) + 1]
$x_1(t)+x_2(t)\frac{d}{dt}$	[x1D0, x2D0]
$-\frac{d}{dt}+\dot{x_2}(t)\frac{d^3}{dt^3}$	[0, -1, 0, x2D1]

Matrix $\in \mathcal{K}\left[\frac{d}{dt}\right]^{n\times m}$

Data type name: SkewMatrix

Listing 5.18: Definition of SkewMatrix in *Maple*

```
> `type/SkewMatrix` := 'Matrix(SkewPolynomial)';
```

In this case, we omit examples since a SkewMatrix is simply a *Maple*-own matrix with SkewPolynomials as entries.

Vector $\in \mathcal{K}\left[\frac{d}{dt}\right]^{n}$

Data type name: SkewVector

Listing 5.19: Definition of SkewVector in *Maple*

```
> `type/SkewVector` := 'Vector(SkewPolynomial)';
```

In this case we also omit examples since a SkewVector is simply a *Maple*-own matrix with SkewPolynomials as entries.

Therefore, the final data structure of the minimal basis decomposition for nonlinear systems is given by:

Figure 5.2: Abridged class diagram for the submodule MinimalbasisDecomp

Chapter 6

Issues of the Implementation

This chapter discusses several certain issues concerning the implementation which allow a higher performance of the algorithms or improve the toolboxes concerning usability and quality.

6.1 Using functions versus symbols

The most important issue during the implementation was using symbols instead of functions to express function of time in both of the toolboxes in order to decrease the needs of computation time and memory.

6.1.1 Introducing Regular Expressions for Function Names

Usually, functions of time are represented in *Maple* by the common mathematical notation:

Listing 6.1: Function of time

```
> x(t);
                    x(t)
```

The derivative of a not specified function can be computed via the method diff or the operator D:

Listing 6.2: Differentiate a function

```
> diff(x(t),t);
                  d/dt x(t)
> D[1](x)(t)
                  D(x)(t)
  #The result of the D-operator can be converted to the diff-notation
> convert(D[1](x)(t), diff);
                  d/dt x(t)
  #And vice versa
> convert(diff(x(t),t), D);
                  D(x)(t)
```

This also works in case of delays or predictions:

Listing 6.3: Differentiate a function with a delay

```
> diff(f(t-tau), t);
                    D(f)(t - tau)
```

Thus, we could use the standard *Maple*-own functions to represent the meromorphic functions we encounter. To increase the performance of the implemented algorithms, the decision was made to use the *Maple* data type symbol instead of function[1]. In order to do this, we convert the function-based notation into a symbol-based one using a special pattern for the name.
In case of linear systems, we need three attributes within the name:

i) the actual function name

ii) the degree of the derivative of the function with respect to t

iii) the magnitude of the delay resp. prediction of the function

This yields the following regular expression for function names (for an explanation of the syntax of regular expressions see [32]):

$$\hat{}\underbrace{[\text{a-zA-CE-OQ-SU-Z0-9\_}]+}_{i)}\underbrace{\text{D}[0\text{-}9]+}_{ii)}\underbrace{(\text{T}[0\text{-}9]+|\text{P}[1\text{-}9][0\text{-}9]^*)}_{iii)}\$ \tag{6.1}$$

The capitalized letters **D**, **T** and **P** are keywords and must not be used inside the function name. All other letters (including the lowercased d, t and p), numbers and the underscore are allowed to be used in function names as well as in standard *Maple*. The name is followed by the keyword **D** and the degree of the derivative of the function. Finally, the expression waits for the keyword **T** (in case of delays) or **P** (in case of predictions) followed by the magnitude. We define a function with neither delays nor predictions as **T0**. A few examples shall demonstrate the regular expression:

Math. function	In DifferentialDelays
$x(t-\tau)$	xD0T1
$\ddot{x}(t)$	xD2T0
$\dot{\omega}_y(t+3\tau)$	omega_yD1P3
$\_Foo\_Bar\_1(t)$	_Foo_Bar_1D0T0
$y^{(12)}(t-10\tau)$	yD12T10

In a second step, we have to define constrains for possible names of constants, too. Otherwise we would not be able to differentiate between constants and functions. Thus, we consider constants to fit the regular expression

$$\hat{}[\text{a-zA-CE-OQ-SU-Z0-9\_}]+\$ \tag{6.2}$$

i.e. constants must not contain the keywords **D**, **T** and **P**. A few examples for constants:

[1] We will illustrate the improvement of the performance at a later point in this section.

Math. constant	In DifferentialDelays
τ	tau
_$C1$	_C1

In case of nonlinear systems, we omit the delays resp. predictions yielding the regular expression for functions:

$$\hat{}\underbrace{[\text{a-zA-CE-Z0-9\_}]+}_{i)}\underbrace{\text{D}[\text{0-9}]+}_{ii)}\$ \tag{6.3}$$

As we can see in (6.3), only the keyword **D** exists due to the fact that we do not have delays and predictions in the nonlinear case. Thus, we are allowed to use the formerly keywords **T** and **P** in function names:

Math. function	In DifferentialForms
$x(t)$	xD0
$\ddot{\delta}(t)$	deltaD2
$\dot{y}_{flat}(t)$	y_flatD1
Ψ $1(t)$	_Psi_1D0

Similar to the linear case, all constants have to fit the regular expression

$$\hat{}[\text{a-zA-CE-Z0-9\_}]+\$ \tag{6.4}$$

A few examples for constants:

Math. constant	In DifferentialForms
Θ	Theta
_$C1$	_C1

6.1.2 Spell Checking to Avoid Invalid Functions and Constants

To ensure that all functions and constants have valid names, there exist methods for spell checking in the toolboxes which validate all user-entered expressions whether they fit the corresponding regular expressions. These validations are only executed when the user creates new instances of the data types and not during the computation itself. Because of the internal use of functions and constants of the toolbox, it is not possible that invalid functions and/or constants may be created during the computations, thus additional spell checking would need more computation time without creating any benefits.
For instance, in case of the nonlinear toolbox, the internal spell checking method is:

Listing 6.4: spellingChecker

```
1 local spellingChecker := proc(term::Not(list),
{onlyFunctionsAllowed::boolean:=false})
```

```
2  description "This method checks the spelling of the functions and
   constants. If there are functions which do not fit the regular expression
   ^[a-zA-CE-Z0-9_]+D[0-9]+$ or constants which do not fit ^[a-zA-CE-Z0-9_]+$,
   the method will raise an error and explain why!";
3  local
4    #list(name)
5    varList,
6    #name
7    variable;
8
9  varList := indets(term, name);
10 for variable in varList do
11   if (not StringTools[RegMatch]("^[a-zA-CE-Z0-9_]+D[0-9]+$", variable)) then
12     if (onlyFunctionsAllowed) then
13       error("The expression %1 is no function. Please spell your functions
         compatible to [a-zA-CE-Z0-9_]+D[0-9]+", variable);
14     else
15       if (not StringTools[RegMatch]("^[a-zA-CE-Z0-9_]+$", variable)) then
16         error("The expression %1 is neither a function nor a constant.
           Please spell your functions compatible to [a-zA-CE-Z0-9_]+D[0-9]+
           and your constants compatible to [a-zA-CE-Z0-9_]+", variable);
17       end if;
18     end if;
19   end if;
20 end do;
21 return NULL;
22 end proc:
```

So the user will immediately get an error if he enters invalid spelled functions resp. constants. The option onlyFunctionsAllowed occurring in line 1 and 12 is used for the spell checking of certain terms which may only contain functions such as MonoDiffForms (see section 5.3). In those cases, even right spelled constants will be invalid. The spell checking method in the linear case works similar but checks for the regular expressions (6.1) and (6.2).

6.1.3 Implementing the Time Derivative for Arbitrary Terms

Due to the fact that we no longer have *Maple*-functions of t, we have to implement our own method to differentiate arbitrary terms with functions and constants in them with respect to t. In order to do this, we use the Lie derivative along the Cartan field to differentiate the term. This shall be illustrated by a simple example: Let $x(t) + y(t)\dot{z}(t)$ be the term we want to differentiate. Then, we are able to compute the differentiation by applying the operator

$$\dot{x}(t)\frac{d}{dx(t)} + \dot{y}(t)\frac{d}{dy(t)} + \ddot{z}(t)\frac{d}{d\dot{z}(t)} \tag{6.5}$$

to the term. Written in the syntax of the toolbox the operator would be

$$xD1\frac{d}{dxD0} + yD1\frac{d}{dyD0} + zD2\frac{d}{dzD1}. \tag{6.6}$$

In the nonlinear toolbox, this can be done with the local method timeDerivative:

Listing 6.5: timeDerivative

```
local timeDerivative := proc(term::Not(list))
  description "This method differentiates the given term with respect to
  time.";
  local
    #list(name)
    varList,
    #list(string)
    splitted,
    #name
    variable,
    #Not(list)
    derivTerm;

  derivTerm := 0;
  varList := indets(term, name);
  for variable in varList do
    if (SearchText("D", variable) = 0) then next; end if;
    splitted := StringTools[Split](variable, "D");
    derivTerm := derivTerm + diff(term, variable) * convert(cat(splitted[1],
    "D", parse(splitted[2], statement)+1), symbol);
  end do;
  return derivTerm;
end proc:
```

In the linear toolbox, the differentiation is done similarly, but there is one difference: To represent t in terms (which is important in case of linear systems), the decision was made to define t as **tD0T0**. Analog $t - x\tau$ and $t + x\tau$ are represented by **tD0Tx** resp. **tD0Px** with $x \in \mathbb{N}_0$. Derivations of $t \pm x\tau$ are represented by **tDyTx** resp. **tDyPx** with y as the degree of the differentiation. The internal simplification methods will automatically substitute occurring **tD1Tx** resp. **tD1Px** for 1 and **tDyTx** resp. **tDyPx** with $y > 1$ for 0.

6.1.4 Consequences of Using symbols instead of functions

First of all, we discover a small performance loss, using the time derivative via the Cartan field (6.5). But in return, we encounter the big benefit of not having functions of t in *Maple* anymore. This yields three advantages

1.) faster handling of terms with symbols instead of functions

2.) much faster evaluation of the included functions

3.) no need for conversion between D and diff-notation

1.) The data type symbol seems to be more lightweight since handling terms with symbols instead of functions needs less computation time than vice versa[2]. Since the algorithms in the toolboxes will produce mathematical terms with

[2] A simple performance test showed a reduction about 30-40%, depending on the size of the terms. The bigger the terms get, the higher the performance boost due to using symbols instead of functions is.

a high complexity, fast methods for simplification, canceling, multiplication, etc. are very important.

2.) In many methods of the toolbox, but especially in the implementation of the exterior derivative (3.81) in case of nonlinear systems, it is required to evaluate the concrete functions (including their current differentiation degree) which are contained in a specific term. A small example to illustrate this requirement: Let $F(\overline{x}), \overline{x} \in \mathfrak{X}$ be

$$F(\overline{x}) = x_1(t) - \ddot{x}_2(t) + sin(x_3(t)^2). \tag{6.7}$$

The exterior derivative $d(F(\overline{x}))$ according to (3.82) is given by

$$d(F(\overline{x})) = d(x_1(t)) + d(\ddot{x}_2(t)) + 2x_3(t)cos(x_3(t))d(x_3(t)). \tag{6.8}$$

I.e. there has to be a method which (in this example) returns the functions $x_1(t)$, $\ddot{x}_2(t)$ and $x_3(t)$ in order to work properly. For this purpose *Maple* offers the method indets, which evaluates the indeterminates of an expression. It can be specified by providing the data type:

Listing 6.6: Usage of indets

```
#Find all functions
> indets(x_1(t) - diff(x_2(t), t$2) + sin(x_3(t)^2), function);
          {diff(x_2(t), t), diff(x_2(t), t, t), sin(x_3(t)^2), x_1(t),
          x_2(t), x_3(t)}
#Find all functions of symbols
> indets(x_1(t) - diff(x_2(t), t$2) + sin(x_3(t)^2), function(symbol));
          {x_1(t), x_2(t), x_3(t)}
```

Even the approach of iterating over the more general indets result and differentiating with respect to each function to see whether the function as a whole exists in the term will not work in case of nested functions (like e.g. $\sin((x_3(t))^2)$). It may be possible to implement a method which returns the needed functions using a combination of the techniques mentioned above, but this method would take without much doubt a lot of computation time due to the variety of cases that has to be considered.
In the toolbox developed in [33], this problem was solved by storing the occurring functions in a special attribute of each coefficient. But this approach caused a high need of computation time since this function-list had to be maintained during the algorithms.
Without the time dependency the evaluation of the functions is quite simple. We just have to evaluate all symbols in the specific term and then exclude all symbols without the keyword **D** in it (to exclude all constants):

Listing 6.7: Get all functions of a term

```
1 #varList contains all symbols in the term (constants and functions)
2 varList := indets(term, symbol);
3 functions := [];
4 for variable in varList do
5   #Does the variable contain the keyword D?
```

```
6   if (SearchText("D", variable) = 0) then next; end if;
7   #Add a found function to the list of functions
8   functions := [op(functions), variable];
9 end do;
```

3.) There exist two notations for the differentiation of functions in *Maple*: the D and diff-notation. Actually D is handled as an operator while diff is a function. Both can be used for the differentiation of terms with respect to variables as seen in listing 6.2 and can be converted in the opposite notation via the convert method of *Maple*. But it cannot be determined without doubt when *Maple* uses the diff notation and when the D notation since both can be used in a similar way[3]. Also users may use both notations to represent differentiations in the system matrices. Thus, it would be necessary to convert every term that could contain differentiations into one of the two notations, which would cause an enormous performance loss.
Without the time dependency, we have an own representation of differentiations and therefore there exists no need for conversion between the two representations in *Maple*.

6.2 Discussion on the Need of a Unique Data Structure

As explained in chapter 4 and 5 the decision was made to develop two separated data structures[4] instead of a common data structure for all kinds of control systems. In this section, we will discuss the possibility of merging those data structures into a single data structure and the consequences in terms of performance and usability.
At first, let us recall the key data type OrePolynomial of the linear toolbox. It represents a polynomial $p(t) \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$:

$$p(t) = \sum_{i=0}^{N} b_i(t)^{-1} a_i(t) \frac{d^i}{dt^i},\ a_i(t), b_i(t) \in \mathcal{K}[\delta],\ \deg(p) = N \in \mathbb{N}_0 \tag{6.9}$$

with the polynomials $a_i(t), b_i(t) \in \mathcal{K}[\delta]$:

$$a_i(t) = \sum_{j=0}^{R_i} a_{i,j}(t)\delta^j,\ a_{i,j}(t) \in \mathcal{K},\ \deg(a_i) = R_i \in \mathbb{N}_0 \tag{6.10}$$

$$b_i(t) = \sum_{j=0}^{S_i} b_{i,j}(t)\delta^j,\ b_{i,j}(t) \in \mathcal{K},\ \deg(b_i) = S_i \in \mathbb{N}_0 \tag{6.11}$$

[3]For example, during the development of the toolboxes it was discovered that the result of the partial differential equation solver of *Maple* changed the kind of notation when a higher version of *Maple* was used.

[4]For the minimal basis decomposition in case of nonlinear systems, we used a specialized internal data structure in order to speed up the computations. This is mentioned in section 5.4.

Although this polynomial is clearly an operator, we will not find a matching data type among the operators of the nonlinear toolbox. A possible approach for merging the data structures would be to enhance the data structures for the monomial operators of the nonlinear toolbox. Since the monomial operator itself is an extension of the monomial p-differential form, we have to extend the monomial p-differential form $\omega \in \Lambda^p(\mathfrak{X})$ (see (5.3)) to

$$\begin{aligned} \theta &= \theta_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \\ &= \theta_{c_D}(\overline{x})^{-1} \theta_{c_N}(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \end{aligned} \tag{6.12}$$

with θ_{c_D} and $\theta_{c_N} \in \mathcal{K}[\delta]$. Moreover, the extension of the monomial operator (see 5.4) is given by

$$\begin{aligned} \nu &= \nu_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_q}^{(j_q)} \wedge \frac{d^n}{dt^n} \\ &= \nu_{c_D}(\overline{x})^{-1} \nu_{c_N}(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_q}^{(j_q)} \wedge \frac{d^n}{dt^n} \end{aligned} \tag{6.13}$$

with ν_{c_D} and $\nu_{c_N} \in \mathcal{K}[\delta]$. I.e. we use left fractions over polynomials in δ as coefficients for monomial p-differential forms (and therefore monomial operators) to enable a representation for delays and predictions. This way, we may represent a monomial of the operator $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ as a monomial operator (6.13) with no attached monomial differential forms. Since an operator $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^{p+q}(\mathfrak{X}))$ is defined as a sum of monomial operators, we may represent our polynomials $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ as operators $\in \mathcal{L}(\Lambda^p(\mathfrak{X}), \Lambda^p(\mathfrak{X}))$.
This yields the hypothetical data structure:

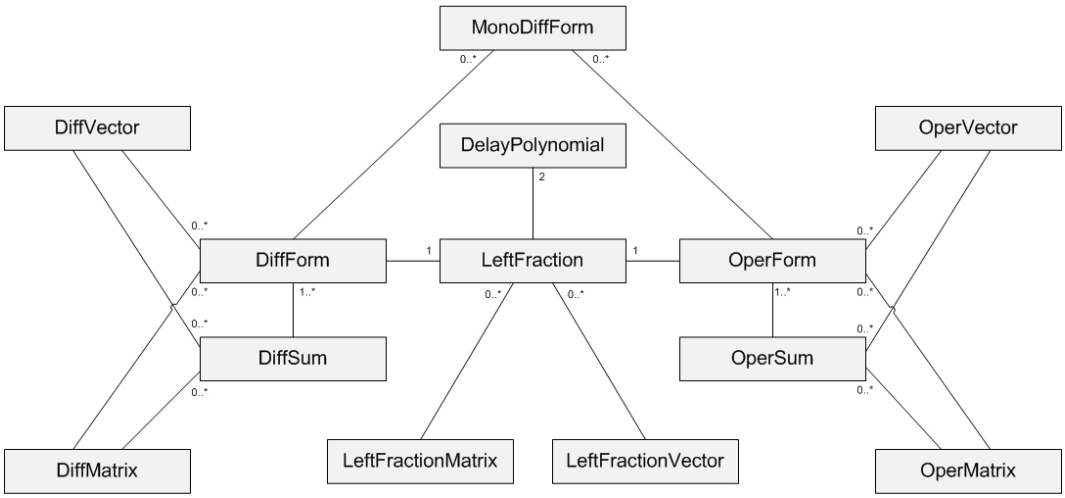

Figure 6.1: Abridged class diagram for the merged data structure

The biggest advantage of a merged data structure would be that we could handle linear and nonlinear systems with and without delays with a single toolbox[5], making it easier for the user to handle the implemented features.

[5] **Remark:** This does not necessarily mean that we have fewer lines of code.

On the other hand, we will definitely encounter a huge loss of performance since we would have to implement the most general case for every computation. E.g. in case of the differentiation of a DiffForm, we would have to consider delays resp. predictions in the coefficients and also contained monomial differential forms. In case of a specialized data structure for each system class, we simply do not implement such cases since they do not appear. In addition, we would have to store more information than needed inside the data types causing *Maple* to allocate more memory than required.
Since computational performance is the key issue of the toolboxes, we maintain a separated data structure.

6.3 On the Treatment of Local Variables to Improve the Computational Performance

Maple has by default a weak typing for local variables in methods. I.e. it does not check the data type of local variables before an assignment during the runtime:

Listing 6.8: Assign an invalid type to a variable

```
> assignmentTest := proc(arbitraryParameter::anything)
    #Set the type of 'result' to 'integer'
    local result::integer;
    result := arbitraryParameter;
    return result;
  end proc;
> assignmentTest(42);
          42
> assignmentTest("foobar");
          "foobar"
```

However, *Maple* also supports a strong typing by changing the assertLevel (see [26] for more details):

Listing 6.9: Enable strong typing

```
> kernelopts(assertlevel = 2);
```

This will cause the second method call above to raise an assertion error, but causes a noticeably performance loss[6]. So, the decision was made to omit typing local variables at all and just write the type of the variables as an inline comment in order to support the developers:

Listing 6.10: Example for type-comments

```
local timeDerivative := proc(term::Not(list))
  description "This method differentiates the given term with respect to
  time.";
  local
    #list(name)
    varList,
```

[6] A simple performance test showed a lack of performance of 15% and more, depending on the number of local variables and assignments.

```
#list(string)
splitted,
#name
variable,
#Not(list)
derivTerm;
...
```

Of course, this leaves us with the usual problems of weak typing, such as unrecognized assertion errors, etc. In this case we accept these in order to speed up the computation. Another approach would be to enable the strong typing for tests and disable it for the actual application. But since we do not know the setting of the assertion level in the environment which is used by the user of the toolboxes, we will not follow this approach.

Chapter 7

Introduction of the Developed Toolboxes

In this chapter, the internal structure of the toolboxes and their functionalities will be described in detail. We will also take a look at the methods the toolboxes provide and how to use[1] them.

7.1 General Remarks

Since the toolboxes introduce own data types for skew polynomials, left fractions and differential forms resp. operators, almost every computation rule and algorithm must explicitly be implemented in the toolboxes. The toolboxes do not use the *Maple* libraries OreTools, DifferentialGeometry and JetCalculus in order to focus on the very particular case of flatness determination. This will distinctly increase the computational performance since we do not consider more general mathematical cases (which the above mentioned libraries are capable of). Nevertheless, this decision leads to the implementation of a lot of methods. To organize the toolboxes, the methods are ordered[2] by their type resp. their visibility:

1.) **Initialization**
These methods are automatically executed when the respective toolbox gets loaded by the with command of *Maple*. They are not visible to the user.

2.) **Constructors**
Constructors are used for creating instances of the data types developed

[1] Some of the methods are not directly visible to the user (due to the *Maple* modifier local). To demonstrate the usage of those methods, the toolbox was altered in order to export these methods. If we want to access them directly, we will have to change the visibility modifier of the methods.

[2] This order corresponds to the order of the methods in the actual source code and in the following chapters.

in the chapters 4 and 5. Although these are also exported methods, they are treated as an own type of methods since they serve the same purpose.
The matrix- and vector-based data types (such as LeftFractionVector, OreMatrix, DiffMatrix, etc.) do not have constructors since they are simply *Maple*-own matrices resp. vectors with custom entries and may still be created by the *Maple* commands Matrix and Vector.

3.) **Local methods**
These methods can only be called from other methods of the toolbox but not from the user himself.

4.) **Exported methods**
These methods are visible to the user and offer the core features of the respective toolbox.

Besides the default *Maple* language core, the toolboxes only need the following libraries (resp. methods of them) in order to work[3]:

DifferentialDelays:

Library	Used methods
LinearAlgebra	ColumnDimension, Dimension, RowDimension
Maplets	Display, various methods from the package Elements
StringTools	RegMatch, RegSubs, Split, StringBuffer, SubstituteAll

DifferentialForms:

Library	Used methods
LinearAlgebra	ColumnDimension, Dimension, MatrixInverse, NullSpace, RowDimension
Maplets	Display, various methods from the package Elements
ListTools	Search
PDETools	difforder, dsubs
RandomTools	Generate
StringTools	RegMatch, RegSubs, Split, StringBuffer, SubstituteAll

Please note that the DifferentialForms toolbox uses the method pdsolve from the *Maple* language core to solve the occurring differential equations.

In addition, we may spend a few words referring to the naming of the methods and variables[4]: In both toolboxes, all constructor methods are written in

[3]This is not a fixed list and may be extended in future versions of the toolboxes. It shall just illustrate that the toolboxes are mainly based on the *Maple* language core and therefore mainly independent from other libraries.

[4]This notation is not specified by *Maple*. Nevertheless, it has been chosen to increase the readability of the source code.

upper camel case, while all variables, parameters and methods are written in lower camel case[5] similar to the common notation in *Java*.

7.2 DifferentialDelays

7.2.1 Internal Structure of the Toolbox

The toolbox for flatness determination in case of linear systems with and without delays consists of the main module and the submodules Decompose, LeftFractionUtils and PiFlatUtils. Each part offers a certain kind of functionality of the toolbox. By including the toolbox in a worksheet using the with command, we automatically gain access to all exported methods of the toolbox and its submodules.

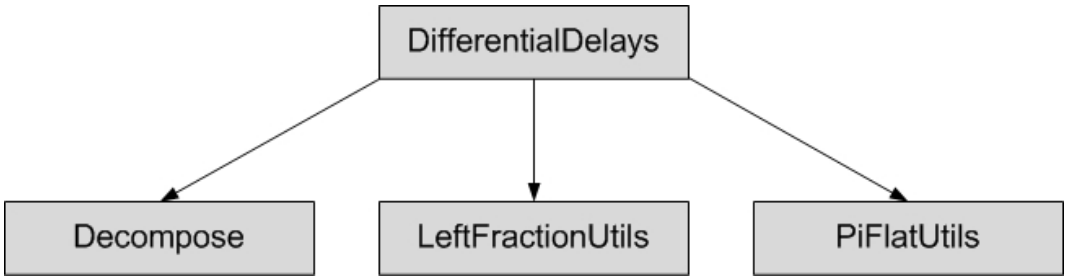

Figure 7.1: Structure of DifferentialDelays

In the following sections the main module and its submodules will be explained in detail.

7.2.2 Main Module

In this section, all methods of the main module of the toolbox DifferentialDelays are described in detail. The main module offers the core methods which are used in order to handle the most of the computation rules, which are defined in chapter 4.
In order to improve the differentiation of the methods, all exported methods which handle DelayPolynomials $\in \mathcal{K}[\delta]$ start with delay, while all methods for LeftFractions $\in \mathcal{K}(\delta)$ start with lFraction and finally methods for OrePolynomials $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ start with ore.

[5] *Upper camel case* means that the first letter of every word within a name is capitalized while all other letters are lowercased (e.g. DelayPolynomial). *Lower camel case* is like *upper camel case*, but the very first letter is also lowercased (e.g. delayShift).

7.2.2.1 Initialization

7.2.2.1.1 initializeMe

Visibility: local

Parameters: –

Return type: –

Description: This method will automatically be called when using the with-command of *Maple*. It initializes the different data types defined in section 4.4. This method also displays these data types and the current version of the toolbox. Roughly speaking, it is the maintenance method of the toolbox.

7.2.2.2 Constructors

7.2.2.2.1 DelayPolynomial

Visibility: export

Parameters:

coefficients ::seq(Not(list))

Return type: DelayPolynomial

Description: This method is one of the constructors of this toolbox. It can be used to create a single DelayPolynomial $\in \mathcal{K}[\delta]$. The parameter coefficients has to be a comma separated sequence of coefficients for the new skew polynomial $\in \mathcal{K}[\delta]$. The first entry represents the coefficient to degree 0 in δ. The coefficients must be of type Not(list) (i.e. any function $\in \mathcal{K}$). If the parameter coefficients is missing, the constructor will create a zero as DelayPolynomial. I.e. the polynomial

$$a_0 + a_1\delta + a_2\delta^2 + \ldots + a_n\delta^n \tag{7.1}$$

with $a_i \in \mathcal{K}$ is created by the command

Listing 7.1: Create a general DelayPolynomial

```
> DelayPolynomial(a_0, a_1, ..., a_n);
```

The constructor automatically checks the spelling of the variables and constants as defined in section 6.1.

Examples:

Listing 7.2: Create DelayPolynomials

```
> poly_1 := DelayPolynomial(0, 1);
> poly_2 := DelayPolynomial(x1D0T0, x2D1T0, 1);
```

These two commands create the polynomials

$$\begin{aligned} poly_1 &= \delta \\ poly_2 &= x_1(t) + \dot{x}_2(t)\delta + \delta^2 \end{aligned} \tag{7.2}$$

7.2.2.2.2 LeftFraction

Visibility: export

Parameters:
denominator::Or(DelayPolynomial, Not(list)),
numerator::Or(DelayPolynomial, Not(list))

Return type: LeftFraction

Description: This method is one of the constructors of this toolbox. It can be used to create a single LeftFraction $\in \mathcal{K}(\delta)$. The first parameter is the denominator and the second parameter the numerator of the left fraction. Numerator and denominator can be of type

- DelayPolynomial or
- Not(list) (any function $\in \mathcal{K}$)

The user may use different types in one LeftFraction. Entered Not(list) will automatically be converted to DelayPolynomials. If the denominator is missing (i.e. there is only one parameter), the denominator will be treated as 1. I.e. the general LeftFractions

$$\begin{aligned} f_1 &= b^{-1} \cdot a \\ f_2 &= 1^{-1} \cdot c \end{aligned} \tag{7.3}$$

with $a, b, c \in \mathcal{K}[\delta]$ are created by the commands

Listing 7.3: Create general LeftFractions

```
> f_1 := LeftFraction(b, a);
> f_2 := LeftFraction(c);
```

Without any parameters the LeftFraction will be treated as zero. The constructor also checks the spelling of the variables and constants as defined in section 6.1.

Examples:

Listing 7.4: Create LeftFractions

```
> frac_1 := LeftFraction(x1D0T0);
> frac_2 := LeftFraction(DelayPolynomial(x1D0T0, 0, 1));
> frac_3 := LeftFraction(x2D0T0, 1);
> frac_4 := LeftFraction(DelayPolynomial(0, 1), DelayPolynomial(x1D0T0,
x2D1T0, 1));
```

These fractions correspond to

$$\begin{aligned} frac_1 &= (1)^{-1} \cdot x_1(t) \\ frac_2 &= (1)^{-1} \cdot (x_1(t) + \delta^2) \\ frac_3 &= (x_2(t))^{-1} \cdot 1 \\ frac_4 &= (\delta)^{-1} \cdot (x_1(t) + \dot{x}_2(t)\delta + \delta^2) \end{aligned} \tag{7.4}$$

7.2.2.2.3 OrePolynomial

Visibility: export

Parameters:

coefficients ::seq(Or(DelayPolynomial, LeftFraction, Not(list)))

Return type: OrePolynomial

Description: This method is one of the constructors of this toolbox. It can be used to create a single OrePolynomial $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$. The parameter coefficients has to be a comma separated sequence of coefficients for the new skew polynomial $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$. The first entry represents the coefficient to degree 0 in $\frac{d}{dt}$. The coefficients can be of type

- LeftFraction
- DelayPolynomial
- Not(list) (any function $\in \mathcal{K}$)

The types of the coefficients may be mixed. Entered DelayPolynomials and Not(list) will automatically be converted to LeftFractions. Therefore the user is able to enter the simpler types without converting them by himself. If the parameter coefficients is missing, the constructor will create a zero as OrePolynomial. I.e. the polynomial

$$f_0 + f_1\frac{d}{dt} + f_2\frac{d^2}{dt^2} + \ldots + f_n\frac{d^n}{dt^n} \tag{7.5}$$

with $f_i \in \mathcal{K}(\delta)$ is created by the command

Listing 7.5: Create a general OrePolynomial

```
> OrePolynomial(f_0, f_1, ..., f_n);
```

The constructor checks the spelling of the variables and constants as defined in section 6.1.

Examples:

Listing 7.6: Create OrePolynomials

```
> poly_1 := OrePolynomial(0, 1);
> poly_2 := OrePolynomial(0, 0, DelayPolynomial(0, x1D0T0), 1);
> poly_3 := OrePolynomial(LeftFraction(DelayPolynomial(0, 1),
DelayPolynomial(x2D0T2, 1)), sin(x2D0T1) * x1D0T0);
```

These commands create the skew polynomials

$$\begin{aligned} poly_1 &= \frac{d}{dt} \\ poly_2 &= (x_1(t)\delta)\frac{d^2}{dt^2} + \frac{d^3}{dt^3} \\ poly_3 &= (\delta)^{-1}\cdot(x_2(t-2\tau)+\delta) + (\sin(x_2(t-\tau))x_1(t))\frac{d}{dt} \end{aligned} \tag{7.6}$$

7.2.2.3 Local methods

7.2.2.3.1 convertTermToLatex

Visibility: local

Parameters:
term::Not(list)
Option: substituteGreekLetters::boolean:=true

Return type: string

Description: This method is used by the exported latexPrinting-methods of this toolbox (see the methods delayLatexPrinting, lFractionLatexPrinting and oreLatexPrinting). It converts the given term into *LaTeX*-code. It contains several features, which shall be described in detail:

1.) substitute functions

2.) substitute certain characters

3.) substitute Greek letters (if enabled)

1.) As explained in section 6.1, all functions are represented by symbols which satisfy a certain regular expression which contains the differentiation degree and the delay resp. prediction. An exceptional case is t since t is itself a function of t, but shall not be converted to `t(t)`. In a first step, these symbols will be converted into *LaTeX*-code. A few examples shall illustrate this:

Function as symbol	Converted function
x1D0T1	`x1(t - \tau)`
yD2P2	`\ddot{y}(t + 2 \tau)`
yD4T0	`y^{(4)}(t)`
tD0P1	`t + \tau`
tD1T0	`\dot{t}`

2.) To increase the readability of the produced *LaTeX*-code we also substitute the characters `_` with `\_` and `*` with `\cdot`.

3.) This method uses the local method substituteGreekLettersInTerm (7.2.2.3.5) to substitute Greek letters. If the option substituteGreekLetters is enabled, all lowercased Greek letters except 'eta', 'delta', 'psi' and 'tau' will be substituted with the corresponding *LaTeX*-expressions. 'eta' and 'psi' may not be substituted to avoid conversion problems since they are literally contained in other Greek letters (e.g. in 'theta'). The letters 'delta' and 'tau' will not be substituted in order to avoid getting confused with the delays and predictions of the functions.
If a Greek letter is followed by another character, the toolbox will use the underscore to separate the Greek letter from the following characters. A few examples with Greek letters shall illustrate the substitution:

Function	Converted function
2*sin(thetaD1P2)	`2\cdot sin(\dot{\theta}(t + 2 \tau))`
_upsilon1	`\_\upsilon\_1`
omega1D0T0 + muxD0T1	`\omega\_1(t) + \mu\_x(t - \tau)`

7.2.2.3.2 lclmMultipliers

Visibility: local

Parameters:
polyA::DelayPolynomial
polyB::DelayPolynomial

Return type: DelayPolynomial, DelayPolynomial

Description: This method computes the cofactors of polyA and polyB which lead to the least common left multiple. I.e. the result of the method are two DelayPolynomials $r, s \in \mathcal{K}[\delta]$ such that

$$r \cdot polyA = s \cdot polyB. \tag{7.7}$$

In addition, r and s satisfy the conditions defined in Definition 2 for being the cofactors of polyA and polyB. That means that

$$r \cdot polyA = s \cdot polyB = \text{lclm}(polyA, polyB). \tag{7.8}$$

The used algorithm corresponds algorithm 1, but the user do not have to take care of the degrees of the given polynomials. The method will automatically correct the order of the parameters.

Examples: In the first example, we compute the cofactors of the two DelayPolynomials

$$\begin{aligned} poly_1 &= x_1(t)\frac{d}{dt} \\ poly_2 &= x_2(t) \end{aligned} \tag{7.9}$$

with the commands

Listing 7.7: Compute cofactors

```
> poly_1 := DelayPolynomial(0, x1D0T0);
  poly_2 := DelayPolynomial(x2D0T0);
  mult_2, mult_1 := lclmMultipliers(poly_2, poly_1);
```

This returns

$$poly\_1 := [0, x1D0T0]$$
$$poly\_2 := [x2D0T0]$$
$$mult\_2, mult\_1 := \left[0, \frac{x1D0T0}{x2D0T1}\right], [1]$$

i.e. the cofactors are:

$$\begin{aligned} mult_1 &= 1 \\ mult_2 &= \frac{x_1(t)}{x_2(t-\tau)}\frac{d}{dt} \end{aligned} \tag{7.10}$$

In the second example, we compute the cofactors of the two DelayPolynomials

$$\begin{aligned} poly_1 &= \sin(x_1(t)) + x_1(t-\tau)\frac{d}{dt} \\ poly_2 &= \cos(x_1(t)) + \frac{1}{x_2(t+3\tau)}\frac{d}{dt} \end{aligned} \tag{7.11}$$

Listing 7.8: Compute cofactors

```
> poly_1 := DelayPolynomial(sin(x1D0T0), x1D0T1);
  poly_2 := DelayPolynomial(cos(x1D0T0), 1/x2D0P3);
  mult_1, mult_2 := lclmMultipliers(poly_1, poly_2);
```

$$poly\_1 := [\sin(x1D0T0), x1D0T1]$$

$$poly\_2 := \left[\cos(x1D0T0), \frac{1}{x2D0P3}\right]$$

$$mult\_1, mult\_2 := \left[\frac{\cos(x1D0T0)}{-\sin(x1D0T0) + x1D0T1\,x2D0P3\cos(x1D0T0)}, -\frac{1}{x2D0P3\,(\sin(x1D0T1) - x1D0T2\,x2D0P2\cos(x1D0T1))}\right],$$
$$\left[\frac{\sin(x1D0T0)}{-\sin(x1D0T0) + x1D0T1\,x2D0P3\cos(x1D0T0)}, -\frac{x1D0T2\,x2D0P2}{x2D0P3\,(\sin(x1D0T1) - x1D0T2\,x2D0P2\cos(x1D0T1))}\right]$$

This yields the cofactors:

$$\begin{aligned} mult_1 &= \frac{\cos(x_1(t))}{\Lambda} - \frac{1}{\Psi}\frac{d}{dt} \\ mult_2 &= \frac{\sin(x_1(t))}{\Lambda} - \frac{x_1(t-2\tau)x_2(t+2\tau)}{\Psi}\frac{d}{dt} \end{aligned} \tag{7.12}$$

with

$$\begin{aligned} \Lambda &= -\sin(x_1(t)) + x_1(t-\tau)x_2(t+3\tau)\cos(x_1(t)) \\ \Psi &= x_2(t+3\tau)(\sin(x_1(t-\tau)) - x_1(t-2\tau)x_2(t+2\tau)\cos(x_1(t-\tau))) \end{aligned} \tag{7.13}$$

7.2.2.3.3 shift

Visibility: local

Parameters:

term::Not(list)
shiftDegree :: integer:=1

Return type: Not(list)

Description: This method computes the application of δ^n to the given term,

which can be any coefficient of a DelayPolynomial. n corresponds to the value of the parameter shiftDegree, which is optional. If shiftDegree is negative, this method will compute the prediction of the given term.

Examples: For instance, to apply δ^1 to the polynomial

$$term_1 = x_1(t+\tau) + \frac{\sin(x_1(t))}{\cos(\ddot{x}_2(t-\tau))} \tag{7.14}$$

we may use the commands

Listing 7.9: Apply $delta^1$

```
> term_1 := x1D0P1 + sin(x1D0T0)/cos(x2D2T1);
  shift(term_1);
```

$$term\_1 := x1D0P1 + \frac{\sin(x1D0T0)}{\cos(x2D2T1)}$$
$$x1D0T0 + \frac{\sin(x1D0T1)}{\cos(x2D2T2)}$$

To apply δ^{-1} to the same polynomial, we may use the commands

Listing 7.10: Apply $delta^{-1}$

```
> term_1 := x1D0P1 + sin(x1D0T0)/cos(x2D2T1);
  shift(term_1, -1);
```

$$term\_1 := x1D0P1 + \frac{\sin(x1D0T0)}{\cos(x2D2T1)}$$
$$x1D0P2 + \frac{\sin(x1D0P1)}{\cos(x2D2T0)}$$

7.2.2.3.4 spellingChecker

Visibility: local

Parameters:
term::Not(list)

Return type: –

Description: This method is used by the constructor methods of the toolbox in order to check whether all functions and constants are spelled correctly referring to the regular expressions

$$\text{\^{}[a-zA-CE-OQ-SU-Z0-9\_]+D[0-9]+(T[0-9]+|P[1-9][0-9]*)\$} \tag{7.15}$$

for functions and

$$\text{\^{}[a-zA-CE-OQ-SU-Z0-9\_]+\$} \tag{7.16}$$

for constants (see section 6.1 for more details). This method evaluates all occurring functions and constants in the given term and will raise an error if there are wrong spelled functions or constants. Otherwise this method will simply return NULL.

7.2.2.3.5 substituteGreekLettersInTerm

Visibility: local

Parameters:
termAsLatexString::**string**

Return type: **string**

Description: This method substitutes all lowercased Greek letters except 'eta', 'delta', 'psi' and 'tau'. For more details see the description of method convertTermToLatex (7.2.2.3.1).

7.2.2.3.6 timeDerivative

Visibility: local

Parameters:
term::**Not**(**list**)

Return type: **Not**(**list**)

Description: This method differentiates the given term with respect to time. Since there are no more *Maple*-functions of t in the occurring terms (see section 6.1 for further details), the time derivative has to be implemented using the Lie derivative along the Cartan field. For a detailed description of the Lie derivative see section 6.1.3.

7.2.2.4 Exported methods

7.2.2.4.1 delayDegree

Visibility: export

Parameters:
poly::**DelayPolynomial**

Return type: **integer**

Description: This method returns the degree of the given DelayPolynomial according to δ.

7.2.2.4.2 delayDerivative

Visibility: export

Parameters:
toDifferentiate ::**DelayPolynomial**

Return type: DelayPolynomial

Description: This method computes the time derivative of the given DelayPolynomial $a \in \mathcal{K}[\delta]$ which is defined as:

$$\frac{d}{dt}(a) = \frac{d}{dt}\left(\sum_{i=0}^{\deg(a)} a_i(t)\delta^i\right) = \sum_{i=0}^{\deg(a)} \frac{d}{dt}(a_i(t))\delta^i \quad \text{with} \quad a_i(t) \in \mathcal{K} \tag{7.17}$$

Examples: In this case we compute the time derivative of the two polynomials $x_1(t) + \delta + \dot{x}_2(t+\tau)\delta^2$ and $2 + \cos(x_1(t))\delta$:

Listing 7.11: Differentiate DelayPolynomials

```
> poly_1 := DelayPolynomial(x1D0T0, 1, x2D1P1);
  result_1 := delayDerivative(poly_1);
> poly_2 := DelayPolynomial(2, cos(x1D0T0));
  result_2 := delayDerivative(poly_2);
```

The *Maple*-results are

$$result\_1 := [x1D1T0,\ 0,\ x2D2P1]$$
$$result\_2 := [0,\ -\sin(x1D0T0)\ x1D1T0]$$

i.e.

$$\begin{aligned} result_1 &= \dot{x}_1(t) + \ddot{x}_2(t+\tau)\delta^2 \\ result_2 &= -\sin(x_1(t)) \cdot \dot{x}_1(t)\delta \end{aligned} \tag{7.18}$$

7.2.2.4.3 delayEquals

Visibility: export

Parameters:
leftPolynomial :: DelayPolynomial
rightPolynomial::DelayPolynomial

Return type: boolean

Description: This method evaluates whether the two DelayPolynomials are equal. Two DelayPolynomials $a, b \in \mathcal{K}[\delta]$ are called equal if we have

$$\deg(a) = \deg(b) \text{ and } a_i = b_i\ \forall i \in \{0, \ldots, \deg(a)\}. \tag{7.19}$$

7.2.2.4.4 delayLatexPrinting

Visibility: export

Parameters:
argumentToPrint::DelayPolynomial
Option: substituteGreekLetters :: boolean:=true

Return type: string

Description: The internal representation of the data types in *Maple* was not designed to be human readable, but to suit the given requirements best. Nevertheless, the toolbox contains several methods to convert the internal data types into *LaTeX*-code, to offer a readable output. This method is used to convert DelayPolynomials.
Furthermore, this method has the boolean option substituteGreekLetters (with true as default value). This option forces the method to convert all lowercased Greek letters except 'eta', 'delta', 'psi' and 'tau' (see method convertTermToLatex (7.2.2.3.1) for more details).
The regular expressions as defined in section 6.1 will also be converted.

Examples:

Listing 7.12: Convert *DelayPolynomials*

```
> delay_1 := DelayPolynomial(0, 0, 1);
  delayLatexPrinting(delay_1);
> delay_2 := DelayPolynomial(0, alpha);
  delayLatexPrinting(delay_2);
> delay_3 := DelayPolynomial(thetaD0T0, _C3, upsilon1, 2 * sin(thetaD1P2));
  delayLatexPrinting(delay_3);
```

These three examples lead to the following *LaTeX*-output:

$$\begin{aligned} delay_1 &= \delta^2 \\ delay_2 &= (\alpha)\,\delta \\ delay_3 &= \theta(t) + (\_C3)\,\delta + (\upsilon\_1)\,\delta^2 + \left(2 \cdot sin(\dot{\theta}(t+2\tau))\right)\delta^3 \end{aligned} \tag{7.20}$$

7.2.2.4.5 delayMultiply

Visibility: export

Parameters:
leftPoly :: DelayPolynomial
rightPoly :: DelayPolynomial

Return type: DelayPolynomial

Description: This method computes the multiplication of two DelayPolynomials $\in \mathcal{K}[\delta]$ according to the computation rules defined in (3.11).

7.2.2.4.6 delayPlus

Visibility: export

Parameters:
leftSummand::DelayPolynomial
rightSummand::DelayPolynomial

Return type: DelayPolynomial

Description: This method computed the sum of two DelayPolynomials $\in \mathcal{K}[\delta]$ according to the computation rules defined in (3.10).

7.2.2.4.7 delayPrinting

Visibility: export

Parameters:
argumentToPrint::**DelayPolynomial**

Return type: symbol

Description: As told in the description of method delayLatexPrinting (7.2.2.4.4) the internal data structure of the toolbox is not designed to be human readable. Besides the thorough conversion into *LaTeX*-code using delayLatexPrinting, we can use the method delayPrinting which transforms DelayPolynomials into a more readable *Maple*-output. Note that this method does not substitute the regular expressions defined in section 6.1.

7.2.2.4.8 delayShift

Visibility: export

Parameters:
polyToShift::**DelayPolynomial**
shiftDegree::**integer**:=1

Return type: DelayPolynomial

Description: This method computes the multiplication of δ^n with the given DelayPolynomial. The second and optional parameter corresponds to n. Only positive integers and zero are allowed. The default value is 1. I.e. in case of a DelayPolynomial $a \in \mathcal{K}[\delta]$ with $\deg(a) = m$, this method computes

$$\begin{aligned} \delta^{shiftDegree} \cdot a &= \delta^{shiftDegree} \cdot \sum_{j=0}^{m} a_j \delta^j \\ &= \sum_{j=0}^{m} \delta^{shiftDegree}(a_j) \delta^{j+shiftDegree}. \end{aligned} \tag{7.21}$$

7.2.2.4.9 delaySimplifier

Visibility: export

Parameters:
toSimplify::**DelayPolynomial**
Option: simplifyCoefficients::**boolean**:=**true**
Option: eraseZeros::**boolean**:=**true**

Return type: DelayPolynomial

Description: This method is the equivalent to the *Maple* command simplify. It simplifies the given DelayPolynomial. Every method in this toolbox which manipulates DelayPolynomials automatically calls this simplifier. The delaySimplifier composes the features:

1.) simplify the coefficients of the DelayPolynomial

2.) substitute all derivatives of t

3.) erase dispensable leading zeros (if enabled)

1.) This is done by the *Maple* command simplify. This is very important and may not be spared since otherwise terms may be created which are actually zero, but *Maple* does not recognize that until they get simplified.

2.) Let us recall that the variable t is represented by **tD0T0**. The delays and predictions of t, i.e. $t \pm x\tau$ are represented by **tD0Tx** resp. **tD0Px** (for a detailed description see section 6.1). This feature substitutes its derivatives, i.e.

$$\frac{d^n}{dt^n}(t \pm \tau) = \begin{cases} 1 & \text{if } n = 1 \\ 0 & \text{if } n \geq 2 \end{cases} \tag{7.22}$$

3.) This feature erases all leading zeros from the DelayPolynomial, i.e. all coefficients a_i of the DelayPolynomial $a(t) \in \mathcal{K}[\delta]$ which satisfy $i > \deg(a)$. This is necessary due to the fact that the data type is implemented as list with the size equal to the degree of the polynomial plus 1.

7.2.2.4.10 delaySubtract

Visibility: export

Parameters:

leftArgument::**DelayPolynomial**
rightArgument::**DelayPolynomial**

Return type: **DelayPolynomial**

Description: This method subtracts the rightArgument from the leftArgument, i.e. the method computes the result of

$$\begin{aligned} c &= a - b = \sum_{i=0}^{n} a_i \delta^i - \sum_{j=0}^{m} b_j \delta^j \\ &= \begin{cases} \sum_{i=0}^{m} (a_i - b_i)\, \delta^i + \sum_{j=m+1}^{n} a_i \delta^j & \text{if } n \geq m \\ \sum_{i=0}^{n} (a_i - b_i)\, \delta^i + \sum_{j=n+1}^{m} -b_i \delta^j & \text{if } n \leq m \end{cases} \end{aligned} \tag{7.23}$$

with $a, b \in \mathcal{K}[\delta]$, $\deg(a) = n, \deg(b) = m$. The second parameter rightArgument is optional. If it is missing, the method will compute the negative of leftArgument.

7.2.2.4.11 lFractionDerivative

Visibility: export

Parameters:
toDifferentiate :: **LeftFraction**

Return type: **LeftFraction**

Description: This differentiates the given LeftFraction $\in \mathcal{K}(\delta)$ with respect to t according to the differentiation rules defined in Theorem 3.

7.2.2.4.12 lFractionEquals

Visibility: export

Parameters:
argumentOne::**LeftFraction**
argumentTwo::**LeftFraction**

Return type: **boolean**

Description: This method evaluates whether the two given LeftFractions argumentOne and argumentTwo are equal. This method is based on the following assumption: Two LeftFractions $b^{-1}a, d^{-1}c \in \mathcal{K}(\delta)$ are equal if and only if

$$ra = sc \text{ with } rb = sd = \text{lclm}(b, d) \tag{7.24}$$

is fulfilled.

7.2.2.4.13 lFractionInvert

Visibility: export

Parameters:
toInvert :: **LeftFraction**

Return type: **LeftFraction**

Description: This method returns the inverse of the given LeftFraction toInvert. Note that the inverse of a LeftFraction is both left and right inverse.

7.2.2.4.14 lFractionLatexPrinting

Visibility: export

Parameters:
rawArgument::**Or**(**LeftFraction**, **LeftFractionVector**, **LeftFractionMatrix**)
Option: substituteGreekLetters::**boolean**:=**true**

Return type: **string**

Description: Similar to method delayLatexPrinting (7.2.2.4.4), this method converts LeftFractions resp. LeftFractionVectors and LeftFractionMatrices into *LaTeX*-code in order to offer a more readable output. Like delayLatexPrinting this method offers the functionality to convert lowercased Greek letters by

using the option substituteGreekLetters (for more details see method delayLatexPrinting (7.2.2.4.4)).
All regular expressions for variables and constants as defined in section 6.1 will be converted, too.

Examples:

Listing 7.13: Convert LeftFractions

```
> fraction_1 := LeftFraction(DelayPolynomial(thetaD0T0, _C3, 0, 2 *
sin(thetaD1P2)), DelayPolynomial(cos(thetaD0T0), 0, 1));
  lFractionLatexPrinting(fraction_1);
> fraction_2 := LeftFraction(DelayPolynomial(0, 0, thetaD0T0, _C3, 0, 2 *
sin(thetaD1P2)));
  lFractionLatexPrinting(fraction_2);
```

These two examples lead to the following *LaTeX*-output:

$$\begin{aligned} fraction_1 &= \left(\theta(t) + (\_C3)\,\delta + \left(2 \cdot sin(\dot{\theta}(t+2\tau))\right)\delta^3\right)^{-1} \cdot \left(cos(\theta(t)) + \delta^2\right) \\ fraction_2 &= (\theta(t))\,\delta^2 + (\_C3)\,\delta^3 + \left(2 \cdot sin(\dot{\theta}(t+2\tau))\right)\delta^5 \end{aligned} \tag{7.25}$$

7.2.2.4.15 lFractionMultiply

Visibility: export

Parameters:
argumentOne::**LeftFraction**
argumentTwo::**LeftFraction**

Return type: **LeftFraction**

Description: This method computes the product of two LeftFractions according to the computation rules defined in Theorem 2.

7.2.2.4.16 lFractionPlus

Visibility: export

Parameters:
argumentOne::**LeftFraction**
argumentTwo::**LeftFraction**

Return type: **LeftFraction**

Description: This method computes the sum of the two given LeftFractions according to the computation rules defined in Theorem 1.

7.2.2.4.17 lFractionPrinting

Visibility: export

Parameters:
rawArgument::**Or**(**LeftFraction**, **LeftFractionVector**, **LeftFractionMatrix**)

Return type: **symbol**

Description: Similar to the method delayPrinting (7.2.2.4.7) this method transforms LeftFractions resp. LeftFractionVectors and LeftFractionMatrices into a more readable *Maple*-output. The transformation is not as detailed as it is in lFractionLatexPrinting, but it gives us a short overview of the given argument. Note that this method does not substitute the regular expressions defined in section 6.1.

7.2.2.4.18 lFractionSimplifier

Visibility: export

Parameters:

toSimplify :: Or(LeftFraction, LeftFractionMatrix, LeftFractionVector)
Option: simplifyCoefficients :: boolean:=true
Option: eraseZeros :: boolean:=true
Option: cancelFractions :: boolean:=false

Return type: Or(LeftFraction, LeftFractionMatrix, LeftFractionVector)

Description: This method simplifies the given argument toSimplify, which can be of type LeftFraction, LeftFractionMatrix or LeftFractionVector. This method uses the delaySimplifier to simplify all containing DelayPolynomials, using the features which are explained in the description of the method delaySimplifier (7.2.2.4.9).
In addition, this method is able to cancel LeftFractions. If the option cancelFractions is enabled, the method will evaluate whether denominator and numerator are equal (according to the computation rules described in method delayEquals (7.2.2.4.3)). If they are equal, the method will cancel the fractions returning ones as LeftFractions.

7.2.2.4.19 lFractionSubtract

Visibility: export

Parameters:

argumentOne::LeftFraction
argumentTwo::LeftFraction

Return type: LeftFraction

Description: This method subtracts argumentTwo from argumentOne, i.e. the method computes the result of $b^{-1}a - d^{-1}c$ with $a, b, c, d \in \mathcal{K}[\delta]$ which is given by (analog to Theorem 1)

$$\begin{aligned} b^{-1}a - d^{-1}c &= (rb)^{-1}(ra) - (sd)^{-1}(sc) \\ &= (rb)^{-1}(ra - sc) \end{aligned} \tag{7.26}$$

with $rb = sd = \text{lclm}(b, d)$. The second parameter argumentTwo is optional. If it is missing, the method will compute the negative of argumentOne.

7.2.2.4.20 oreDerivative

Visibility: export

Parameters:
toDifferentiate :: Or(OrePolynomial, OreMatrix)

Return type: Or(OrePolynomial, OreMatrix)

Description: This method computes the time derivative of the given OrePolynomial $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ resp. of an OreMatrix $\in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times m}$. In case of an OrePolynomial $p \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ with $p_i \in \mathcal{K}(\delta), i = 1..n$ the differentiation with respect to time is defined as

$$\begin{aligned}
\frac{d}{dt}p &= \frac{d}{dt}\left(\sum_{i=0}^{n} p_i \frac{d^i}{dt^i}\right) \\
&= \sum_{i=0}^{n} \frac{d}{dt}\left(p_i \frac{d^i}{dt^i}\right) \\
&= \sum_{i=0}^{n} \frac{d}{dt}(p_i)\frac{d^i}{dt^i} + p_i \frac{d^{i+1}}{dt^{i+1}} \\
&= \frac{d}{dt}(p_0) + \sum_{i=1}^{n}\left(\left(p_{i-1} + \frac{d}{dt}(p_i)\right)\frac{d^i}{dt^i}\right) + p_n \frac{d^{n+1}}{dt^{n+1}}
\end{aligned} \tag{7.27}$$

7.2.2.4.21 oreEquals

Visibility: export

Parameters:
argumentOne::Or(OrePolynomial, OreMatrix)
argumentTwo::Or(OrePolynomial, OreMatrix)

Return type: boolean

Description: This methods evaluates whether the two given arguments are equal. In case of two OrePolynomials $p, q \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$, the method will return true if the two polynomials fulfill

$$\deg(p) = \deg(q) \text{ and } p_i = q_i \; \forall i \in \{0, \ldots, \deg(p)\}. \tag{7.28}$$

Two OreMatrices are equal if they have the same dimensions and the corresponding OrePolynomials are equal.

7.2.2.4.22 oreIdentityMatrix

Visibility: export

Parameters:
dimension::integer

Return type: OreMatrix

Description: This method creates an identity matrix over OrePolynomials with the given dimension, i.e. it returns a matrix $M \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times n}$ with

$$M_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{7.29}$$

7.2.2.4.23 oreLatexPrinting

Visibility: export

Parameters:

argumentToPrint::Or(OrePolynomial, OreMatrix, OreVector)
Option: substituteGreekLetters::boolean:=true

Return type: string

Description: The purpose of this method is to convert OrePolynomials (resp. matrices and vectors which contain OrePolynomials) into *LaTeX*-code in order to offer a more readable output. Like the methods delayLatexPrinting (7.2.2.4.4) and lFractionLatexPrinting (7.2.2.4.14) this method offers the functionality to convert lowercased Greek letters by using the option substituteGreekLetters (for more details see method delayLatexPrinting (7.2.2.4.4)).
All regular expressions for variables and constants as defined in section 6.1 will be converted, too.

Examples:

Listing 7.14: Convert OrePolynomial and OreVector

```
> poly_1 := OrePolynomial(sin(theta1D0T0), DelayPolynomial(0, thetaD2T1,
mu_2));
  oreLatexPrinting(poly_1);
> vect_1 := Vector[column](3, {
  1 = OrePolynomial(LeftFraction(DelayPolynomial(sin(x1D0T0)))),
  2 = OrePolynomial(0),
  3 = OrePolynomial(x1D0T0, 0, 1)
  });
  oreLatexPrinting(vect_1);
```

These two examples lead to the following *LaTeX*-output:

$$\begin{aligned} poly_1 &= sin(\theta\_1(t)) + \left(\left(\ddot{\theta}(t-\tau)\right)\delta + (\mu\_2)\,\delta^2\right)\cdot\tfrac{d}{dt} \\ vect_1 &= \begin{pmatrix} sin(x1(t)) \\ 0 \\ x1(t) + \frac{d^2}{dt^2} \end{pmatrix} \end{aligned} \tag{7.30}$$

7.2.2.4.24 oreMultiply

Visibility: export

Parameters:

argumentOne::Or(OrePolynomial, OreMatrix)
argumentTwo::Or(OrePolynomial, OreMatrix)

Return type: Or(OrePolynomial, OreMatrix)

Description: This method computes the product of the two given arguments argumentOne and argumentTwo. In case of two OrePolynomials $a, b \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ the multiplication is defined as (similar to (3.9)):

$$\begin{aligned} c &= a \cdot b \\ &= \sum_{i=0}^{n} a_i \frac{d^i}{dt^i} \cdot \sum_{j=0}^{m} b_j \frac{d^j}{dt^j} \\ &= \sum_{i=0}^{n} \left(a_i \frac{d^i}{dt^i} \left(\sum_{j=0}^{m} b_j \frac{d^j}{dt^j} \right) \right) \\ &= \sum_{i=0}^{n} \left(a_i \sum_{j=0}^{m} \frac{d^i}{dt^i} \left(b_j \frac{d^j}{dt^j} \right) \right) \end{aligned} \tag{7.31}$$

with $a_i, b_j \in \mathcal{K}(\delta)$ and $\frac{d}{dt}(b_j)$ defined in Theorem 3.
In case of two OreMatrices the product of two matrices $A \cdot B = C \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n \times p}$ with $A \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n \times m}$ and $B \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{m \times p}$ is defines as[6]:

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} \cdot b_{k,j} \tag{7.32}$$

The multiplication of two LeftFractions is defined in Theorem 2. The multiplication of an OrePolynomial and an OreMatrix and vice versa is not defined and will return an error.

7.2.2.4.25 orePlus

Visibility: export

Parameters:

argumentOne::Or(OrePolynomial, OreMatrix)
argumentTwo::Or(OrePolynomial, OreMatrix)

Return type: Or(OrePolynomial, OreMatrix)

Description: This method computes the sum of two OrePolynomials or OreMatrices. The two parameters must be of the same type, otherwise an error will be raised. The sum of two OrePolynomials $a, b \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right], \deg(a) = n, \deg(b) = m$ is defined as

$$\begin{aligned} c &= a + b = \sum_{i=0}^{n} a_i \frac{d^i}{dt^i} + \sum_{j=0}^{m} b_j \frac{d^j}{dt^j} \\ &= \begin{cases} \sum_{i=0}^{m} (a_i + b_i) \frac{d^i}{dt^i} + \sum_{j=m+1}^{n} a_i \frac{d^j}{dt^j} & \text{if } n \geq m \\ \sum_{i=0}^{n} (a_i + b_i) \frac{d^i}{dt^i} + \sum_{j=n+1}^{m} b_i \frac{d^j}{dt^j} & \text{if } n \leq m \end{cases} \end{aligned} \tag{7.33}$$

[6]The number of columns of the left matrix and the number of rows of the second matrix must be the same. Otherwise the multiplication is not defined.

The sum of two OreMatrices is defined as the usual matrix addition, i.e. the sum of two OreMatrices with the same dimensions is computed element-wise.

7.2.2.4.26 orePrinting

Visibility: export

Parameters:
argumentToPrint::Or(OrePolynomial, OreMatrix, OreVector)

Return type: symbol

Description: Similar to the methods delayPrinting (7.2.2.4.7) and lFractionLatexPrinting (7.2.2.4.17) this method transforms OrePolynomials resp. OreVectors and OreMatrices into a more readable *Maple*-output. The transformation is not as detailed as it is in oreLatexPrinting, but it gives us a short overview of the given argument. Note that this method does not substitute the regular expressions defined in section 6.1.

Examples:

Listing 7.15: Print an OrePolynomial

```
> poly_1 := OrePolynomial(sin(x1D0T0), 0, 0, LeftFraction(DelayPolynomial(0,
-sD0T0, sD0T0), DelayPolynomial(1)));
  orePrinting(poly_1);
```

This method call produces the *Maple* result:

$$sin(x1D0T0) + (((-sD0T0) \cdot \delta + (sD0T0) \cdot `\delta`^2)^(-1) \cdot (1)) \cdot `(\partial/\partial t)`^3$$

7.2.2.4.27 oreSimplifier

Visibility: export

Parameters:
toSimplify :: Or(OrePolynomial, OreMatrix, OreVector)
Option: simplifyCoefficients :: boolean:=true
Option: eraseZeros :: boolean:=true
Option: cancelFractions :: boolean:=false

Return type: Or(OrePolynomial, OreMatrix, OreVector)

Description: This method is able to simplify OrePolynomials, OreMatrices and OreVectors. It uses the methods lFractionSimplifier and delaySimplifier to simplify all containing LeftFractions and DelayPolynomials. The features of the simplification process is described in the descriptions of the methods delaySimplifier (7.2.2.4.9) and lFractionSimplifier (7.2.2.4.18). In addition, the option eraseZeros, which is also used by the delaySimplifier, erases all leading zeros of the polynomials. All methods that handle OrePolynomials, OreMatrices or OreVectors usually use this method to simplify the result before returning it.

7.2.2.4.28 oreSubtract

Visibility: export

Parameters:
argumentOne::Or(OrePolynomial, OreMatrix)
argumentTwo::Or(OrePolynomial, OreMatrix)

Return type: Or(OrePolynomial, OreMatrix)

Description: This method subtracts argumentTwo from argumentOne. The two parameters must be of the same type, otherwise an error will be raised. The subtraction of two OrePolynomials $a, b \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right], \deg(a) = n, \deg(b) = m$ is defined as

$$\begin{aligned} c &= a - b = \sum_{i=0}^{n} a_i \frac{d^i}{dt^i} - \sum_{j=0}^{m} b_j \frac{d^j}{dt^j} \\ &= \begin{cases} \sum_{i=0}^{m} (a_i - b_i) \frac{d^i}{dt^i} + \sum_{j=m+1}^{n} a_i \frac{d^j}{dt^j} & \text{if } n \geq m \\ \sum_{i=0}^{n} (a_i - b_i) \frac{d^i}{dt^i} + \sum_{j=n+1}^{m} -b_i \frac{d^j}{dt^j} & \text{if } n \leq m \end{cases} \end{aligned} \tag{7.34}$$

The subtraction of two OreMatrices is defined as the usual matrix subtraction, i.e. the subtraction of two OreMatrices with the same dimensions is computed element-wise.
The second parameter argumentTwo is optional. If it is missing, the method will compute the negative of argumentOne.

7.2.3 Decompose

This module contains the methods needed for the minimal basis decomposition of OreMatrices over LeftFractions. There is only one exported method: decompose (7.2.3.2.1). All other methods are local methods which are used from the method decompose.

7.2.3.1 Local methods

7.2.3.1.1 degreeOfVector

Visibility: local

Parameters:
selectedVector :: OreVector

Return type: extended_numeric

Description: This method computes the row degree resp. column degree of the given row vector resp. column vector according to Definition 10. I.e. this method returns the highest occurring degree in $\frac{d}{dt}$. Since this method is called very often during the decomposition process, this method does not simplify the coefficients of the given vector. So it is important to simplify selectedVector

(if necessary) before calling this method to make sure that the right degree is returned.
It is necessary to use the data type extended_numeric in order to extend the integer with $-\infty$.

7.2.3.1.2 degreeVectorOfMatrix

Visibility: local

Parameters:
testedMatrix::**OreMatrix**
Option: rowwise::**boolean**:=**false**
Option: columnwise::**boolean**:=**false**

Return type: **Vector**(**extended_numeric**)

Description: This method computes a degree vector v of the given matrix $A \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times m}$. The degree vector is created row-wise or column-wise, so one of the two options rowwise or columnwise must be set to true. Otherwise the method will raise an error. I.e. in case of a row-wise created degree vector the method returns a vector v such that

$$v_i \;=\; \text{degree}_{row}(A_{i,1..m}),\; i = 1..n \tag{7.35}$$

resp.

$$v_i \;=\; \text{degree}_{column}(A_{1..n,i}),\; i = 1..m \tag{7.36}$$

in case of a column-wise created vector.
This method will be used in the algorithms 2 and 3 to compute the vectors α and $\widetilde{\alpha}$.

7.2.3.1.3 findModifiedAlpha

Visibility: local

Parameters:
testedMatrix::**OreMatrix**
Option: rowwise::**boolean**:=**false**
Option: columnwise::**boolean**:=**false**
Option: chooseFirstAlpha::**boolean**:=**false**
Option: chooseLowMemoryAlpha::**boolean**:=**false**
Option: chooseLowDegreeAlpha::**boolean**:=**false**
Option: chooseAlphaByUser::**boolean**:=**false**
Option: debugMode::**boolean**:=**false**

Return type: **OreVector**

Description: In every step of the minimal basis decomposition (see the algorithms 2 and 3) the row degree (resp. column degree) of the uppermost row

(resp. first column) of the remaining matrix with highest row degree (resp. column degree) among the linear dependent rows (resp. columns) of the leading coefficient matrix is reduced by at least 1. The vector α describes a possible linear combination of the rows (resp. columns) of the leading coefficient matrix.
To reduce the degree of a row (resp. a column) according to the description above, the chosen vector α has to be enhanced with degrees in $\frac{d}{dt}$. This enhanced α is called $\widetilde{\alpha}$ and will be returned by this method.
Since there may be several possible α which could reduce the row degree (resp. column degree) of the given matrix testedMatrix, this method computes all possible vectors α and chooses one of them according to the given options (we have to set exactly one of these options to true):

- chooseFirstAlpha
- chooseLowMemoryAlpha
- chooseLowDegreeAlpha
- chooseAlphaByUser

chooseFirstAlpha: This option forces the method to randomly choose one of the found α.

chooseLowMemoryAlpha: In this case the method evaluates the size of the α and chooses the α which has the lowest number of mathematical characters.

chooseLowDegreeAlpha: This option forces the method to choose the α which would cause the lowest occurring degree in $\frac{d}{dt}$ in $\widetilde{\alpha}$.

chooseAlphaByUser: This option opens a *maplet* which shows all possible α and allows the user to choose by himself.

Afterwards, the method computes the corresponding $\widetilde{\alpha}$ and returns it.
This method can be used for row-wise or column-wise decomposition. But we have to set one of the options rowwise or columnwise to true in order to compute the right α.
The option debugMode enables a few console logs which show information to the found resp. chosen α.

7.2.3.1.4 indexOfFirstEntryWithDegreeZero

Visibility: local

Parameters:
orePolynomialVector::**OreVector**

Return type: integer

Description: This method returns the index of the first entry of the given OreVector which has a degree in $\frac{d}{dt}$ of zero. This is used in the method decompose (7.2.3.2.1) to evaluate the row (resp. column) which shall be reduced during each step.

7.2.3.1.5 leadingCoeffMatrix

Visibility: local

Parameters:

testedMatrix::**OreMatrix**
Option: rowwise::**boolean**:=**false**
Option: columnwise::**boolean**:=**false**

Return type: LeftFractionMatrix

Description: This method computes the row (resp. column) leading coefficient matrix of the given OreMatrix testedMatrix, referring to Definition 13. Therefore, one of the two options rowwise or columnwise must be set to true. Note that the returned leading coefficient matrix is of type LeftFractionMatrix since we have a matrix filled with leading coefficients of OrePolynomials.

Examples: To compute the row and column leading coefficient matrices of the matrix

$$matrix_1 = \begin{pmatrix} x(t) + \frac{d^2}{dt^2} & \frac{d}{dt} & x(t) \cdot \frac{d}{dt} \\ x(t) + \frac{d}{dt} & \ddot{y}(t+\tau) \cdot sin(\dot{y}(t+\tau)) & \dot{y}(t) + \frac{d}{dt} \\ x(t) & \left((x(t-\tau))^{-1} \cdot (\dot{x}(t))\right) \cdot \frac{d}{dt} & 0 \\ -1 & 1 & \dot{y}(t) \cdot \frac{d}{dt} \end{pmatrix}, \quad (7.37)$$

we may use the commands

Listing 7.16: Compute the leading coefficient matrix

```
> matrix_1 := Matrix(4, 3, {
  (1,1) = OrePolynomial(LeftFraction(xD0T0, yD0T0), 0, 1),
  (1,2) = OrePolynomial(0, 1),
  (1,3) = OrePolynomial(0, xD0T0),
  (2,1) = OrePolynomial(xD0T0, 1),
  (2,2) = OrePolynomial(yD2P1 * sin(yD1P1)),
  (2,3) = OrePolynomial(yD1T0, 1),
  (3,1) = OrePolynomial(xD0T0, 0),
  (3,2) = OrePolynomial(0, LeftFraction(xD0T1, xD1T0)),
  (3,3) = OrePolynomial(0),
  (4,1) = OrePolynomial(-1, 0),
  (4,2) = OrePolynomial(1),
  (4,3) = OrePolynomial(0, yD1T0)
  });
> Decompose[leadingCoeffMatrix](matrix_1, rowwise = true);
> Decompose[leadingCoeffMatrix](matrix_1, columnwise = true);
```

This yields the row leading coefficient matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & (x(t-\tau))^{-1} \cdot (\dot{x}(t)) & 0 \\ 0 & 0 & \dot{y}(t) \end{pmatrix} \quad (7.38)$$

and the column leading coefficient matrix

$$\begin{pmatrix} 1 & 1 & x(t) \\ 0 & 0 & 1 \\ 0 & (x(t-\tau))^{-1} \cdot (\dot{x}(t)) & 0 \\ 0 & 0 & \dot{y}(t) \end{pmatrix} \tag{7.39}$$

7.2.3.1.6 mapletForCustomAlpha

Visibility: local

Parameters:

alphas :: **list** (**LeftFractionVector**)
degreeVector :: **Vector**(**extended_numeric**)

Return type: integer

Description: This method shows a maplet which displays the given LeftFractionVectors alphas together with their complexity (i.e. number of mathematical characters). The user can choose one of them and the method will dispose the maplet and return the index of the chosen vector α. In addition the given vector degreeVector is displayed. It represents the row (resp. column) degrees of the current matrix during the decomposition process. The data type extended_numeric extends integer with $-\infty$.

Examples:

Listing 7.17: Create the maplet

```
> degreeVector := Vector(4, {(1) = 0, (2) = 1, (3) = 0, (4) = 0});
> alpha_1 := Vector[column](4, {
  (1) = LeftFraction(2 * phi1, -1),
  (2) = LeftFraction(1),
  (3) = LeftFraction(0),
  (4) = LeftFraction(0)
  }):
> alpha_2 := Vector[column](4, {
  (1) = LeftFraction(2 * phi1, 2 * phi2),
  (2) = LeftFraction(0),
  (3) = LeftFraction(-2, 1),
  (4) = LeftFraction(1)
  }):
> alphas := [alpha_1, alpha_2];
> returnedValue := Decompose[mapletForCustomAlpha](alphas, degreeVector);
```

In the example above, the following dialog will be shown:

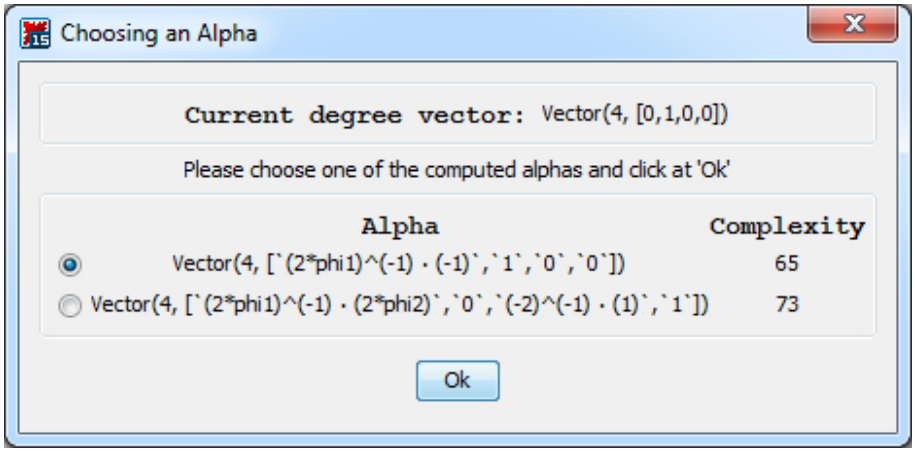

Figure 7.2: The dialog created by mapletForCustomAlpha

7.2.3.1.7 switchColumns

Visibility: local

Parameters:
matrixToChange::**OreMatrix**
indexOne::**integer**
indexTwo::**integer**

Return type: OreMatrix

Description: This method switches two columns of the given OreMatrix matrixToChange. It returns a copy in which the indexOne'th and indexTwo'th column had been switched. This does not affect the given matrixToChange.

7.2.3.1.8 switchRows

Visibility: local

Parameters:
matrixToChange::**OreMatrix**
indexOne::**integer**
indexTwo::**integer**

Return type: OreMatrix

Description: Similar to the method switchColumns (7.2.4.1.4), this method switches two rows of the given OreMatrix matrixToChange. It returns a copy in which the indexOne'th and indexTwo'th row had been switched. This does not affect the given matrixToChange.

7.2.3.2 Exported methods

7.2.3.2.1 decompose

Visibility: export

Parameters:

selectedMatrix::**OreMatrix**
Option: rowwise::**boolean**:=**false**
Option: columnwise::**boolean**:=**false**
Option: chooseFirstAlpha::**boolean**:=**false**
Option: chooseLowMemoryAlpha::**boolean**:=**false**
Option: chooseLowDegreeAlpha::**boolean**:=**false**
Option: chooseAlphaByUser::**boolean**:=**false**
Option: returnInverseOperator::**boolean**:=**false**
Option: debugMode::**boolean**:=**false**

Return type: **OreMatrix**, **OreMatrix**

Description: This method computes the minimal basis decomposition of the OreMatrix selectedMatrix according to the algorithms 2 (in case of row-wise decomposition) and 3 (in case of column-wise decomposition). We have to set the option rowwise or columnwise to true. The method has been optimized in order to increase the computational performance[7].
The method is able to additionally compute the inverse of the operator matrix. In order to do this, we have to set the option returnInverseOperator to true. This feature ensures that we can omit the second minimal basis decomposition needed to compute the defining operator P by computing P directly using $\widetilde{Q}^{-1}$ in Theorem 10.
In each step of the minimal basis decomposition the method computes a vector α which shows the linear dependent rows (resp. columns) of the leading coefficient matrix of the remaining matrix which shall be reduced. The options chooseFirstAlpha, chooseLowMemoryAlpha, chooseLowDegreeAlpha and chooseAlphaByUser determine how the vector α is chosen (see method findModifiedAlpha (7.2.3.1.3) for a detailed description).
The option debugMode enables a few console logs which show information about the steps of the minimal basis decomposition.
The method has four possible orders of the return values, depending on the set options:

[7] E.g. the method does not compute temporary operator matrices for each step, but modifies the remaining matrix and the operator matrix in the same way in order to omit redundant operations.

	returnInverseOperator = **false**
rowwise = **true** columnwise = **false**	operator, remainder
rowwise = **false** columnwise = **true**	remainder, operator

	returnInverseOperator = **true**
rowwise = **true** columnwise = **false**	inverse, operator, remainder
rowwise = **false** columnwise = **true**	remainder, operator, inverse

Let us recall that in case of a row-wise decomposition of $M \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{r\times s}$ we have

$$\underbrace{V}_{operator} M = \underbrace{\begin{pmatrix} I_s \\ 0_{(r-s)\times s} \end{pmatrix}}_{remainder}. \tag{7.40}$$

Since V is unimodular, there exists an inverse operator V^{-1}, which can be computed by using the option returnInverseOperator. In case of a column-wise decomposition we have

$$M \underbrace{U}_{operator} = \underbrace{\begin{pmatrix} I_r & 0_{r\times(s-r)} \end{pmatrix}}_{remainder}. \tag{7.41}$$

Since U is unimodular, there exists an inverse operator U^{-1}, which can be computed by using the option returnInverseOperator.
If the matrix selectedMatrix is not hyper-regular, the method also returns the operator matrix and remaining matrix, but adds a console log that the matrix is not hyper-regular.

7.2.4 LeftFractionUtils

This module contains several methods for handling LeftFractionMatrices. These methods are necessary since every leading coefficient matrix of an OreMatrix is a LeftFractionMatrix which *Maple* cannot handle by default. In case of nonlinear systems, this problem will not occur since every appearing leading coefficient matrix is still a matrix over meromorphic functions.

7.2.4.1 Local methods

7.2.4.1.1 computeNullSpace

Visibility: local

Parameters:

computeKernelOf::**LeftFractionMatrix**
Option: enforceComputationsFromRight::**boolean**:=**false**
Option: showRemainderMatrix::**boolean**:=**false**
Option: withoutSimpleBasis::**boolean**:=**false**
Option: debugMode::**boolean**:=**false**

Return type: **set**(**LeftFractionVector**)

Description: Computes the null space of the given matrix. I.e. for the given matrix computeKernelOf this method computes the basis vectors b_i such that $\forall k_i \in \mathcal{K}(\delta)$ we have

$$\text{computeKernelOf} \cdot \alpha = 0 \text{ with } \alpha = k_1 b_1 + \ldots + k_n b_n \tag{7.42}$$

The option enforceComputationsFromRight forces the method to apply all operations during the internal Gauß-Jordan algorithm from the right[8]. To compute the null space which is created by the rows of a certain matrix, we have to enter the transposed matrix and set the option enforceComputationsFromRight to true.

The option showRemainderMatrix adds a second result value which represents the remaining matrix after applying the Gauß-Jordan algorithm before evaluating the basis vectors of the null space. In this case the result values are of the order:

remaining matrix, basis vectors

If the option withoutSimpleBasis is set to true (false by default), the method will only result basis vectors which are different from an identity vector.
The option debugMode enables several console logs which show information about the applied Gauß-Jordan algorithm.

7.2.4.1.2 computeLeftInverse

Visibility: local

Parameters:

toInvert :: **LeftFractionMatrix**
showRemainderMatrix::**boolean**:=**false**
debugMode::**boolean**:=**false**

[8] This is important since the multiplication of DelayPolynomials and therefore LeftFractions is not commutative.

Return type: LeftFractionMatrix

Description: This methods computes the left inverse of the given LeftFractionMatrix toInvert by using the Gauß-Jordan algorithm.

The option showRemainderMatrix adds a second result value which represents the remaining matrix of the Gauß-Jordan algorithm. This matrix must always be the identity matrix. Therefore, this option is used for test and debug reasons. In this case the result values are of the order:

left inverse, remaining matrix

The option debugMode enables several console logs which show information about the applied Gauß-Jordan algorithm.

7.2.4.1.3 computeRightInverse

Visibility: local

Parameters:

toInvert :: LeftFractionMatrix
showRemainderMatrix::boolean:=false
debugMode::boolean:=false

Return type: LeftFractionMatrix

Description: Similar to method computeLeftInverse (7.2.4.1.2), this methods computes the left inverse of the given LeftFractionMatrix toInvert by using the Gauß-Jordan algorithm.
The options showRemainderMatrix and debugMode are explained in the description of the method computeLeftInverse (7.2.4.1.2).

7.2.4.1.4 switchColumns

Visibility: local

Parameters:

matrixToChange::LeftFractionMatrix
indexOne::integer
indexTwo::integer

Return type: LeftFractionMatrix

Description: This method switches two columns of the given LeftFractionMatrix matrixToChange. It returns a copy in which the indexOne'th and indexTwo'th column had been switched. This does not affect the given matrixToChange.

7.2.4.1.5 switchRows

Visibility: local

Parameters:
matrixToChange::**LeftFractionMatrix**
indexOne::**integer**
indexTwo::**integer**

Return type: LeftFractionMatrix

Description: Similar to the method switchColumns (7.2.4.1.4), this method switches two rows of the given LeftFractionMatrix matrixToChange. It returns a copy in which the indexOne'th and indexTwo'th row had been switched. This does not affect the given matrixToChange.

7.2.4.2 Exported methods

7.2.4.2.1 equalsMatrix

Visibility: export

Parameters:
leftMatrix :: **LeftFractionMatrix**
rightMatrix::**LeftFractionMatrix**

Return type: boolean

Description: This method evaluates whether the two given LeftFractionMatrices leftMatrix and rightMatrix are equal. Two LeftFractionMatrices $A \in \mathcal{K}(\delta)^{r\times s}, B \in \mathcal{K}(\delta)^{n\times m}$ are called equal if and only if the two assertions

i) $r = n \wedge s = m$

ii) $A_{i,j} = B_{i,j}, \quad i = 1..r, j = 1..s$

are satisfied. Let us recall that the equality of two LeftFractions $\in \mathcal{K}(\delta)$ is checked by the method lFractionEquals (7.2.2.4.12).

7.2.4.2.2 identityMatrix

Visibility: export

Parameters:
dimension::**integer**

Return type: LeftFractionMatrix

Description: This method returns an identity matrix over LeftFractions of the given dimension.

7.2.4.2.3 invertMatrix

Visibility: export

Parameters:

toInvert :: **LeftFractionMatrix**
Option: leftInverse :: **boolean**:=**false**
Option: rightInverse :: **boolean**:=**false**
Option: showRemainderMatrix::**boolean**:=**false**
Option: debugMode::**boolean**:=**false**

Return type: LeftFractionMatrix

Description: This method computes the inverse of the given matrix toInvert $\in \mathcal{K}(\delta)^{n\times m}$. Let us recall that the left inverse of a LeftFractionMatrix and the right inverse of a LeftFractionMatrix may be different. Therefore, we have to set one of the options leftInverse or rightInverse to true in order to force the method to compute the desired inverse.
This method uses the functionality of the local methods computeLeftInverse (7.2.4.1.2) and computeRightInverse (7.2.4.1.3) in order to fulfill its purpose.
The options showRemainderMatrix and debugMode are explained in the description of the method computeLeftInverse (7.2.4.1.2).

Examples: In this example, we compute the left inverse of

$$matrix_1 = \begin{pmatrix} 5^{-1}\cdot(x1(t)+x1(t)\delta) & \delta^{-1}\cdot(x1(t)+x2(t-\tau)\delta^2) \\ x1(t)^{-1}\cdot x2(t) & ((x1(t))\,\delta)^{-1}\cdot\delta \end{pmatrix} \quad (7.43)$$

by using the commands

Listing 7.18: Compute the left inverse of a LeftFractionMatrix

```
> matrix_1 := Matrix(2, 2, {
(1,1) = LeftFraction(5, DelayPolynomial(x1D0T0, x1D0T0)),
(1,2) = LeftFraction(DelayPolynomial(0,1), DelayPolynomial(x1D0T0, 0,
x2D0T1)),
(2,1) = LeftFraction(x1D0T0, x2D0T0),
(2,2) = LeftFraction(DelayPolynomial(0, x1D0T0), DelayPolynomial(0, 1))
});
> leftInverseMatrix := LeftFractionUtils[invertMatrix](matrix_1,
leftInverse=true):
```

7.2.4.2.4 multiplyMatrix

Visibility: export

Parameters:

leftMatrix :: **Or**(**LeftFractionMatrix**, **LeftFractionVector**)
rightMatrix :: **Or**(**LeftFractionMatrix**, **LeftFractionVector**)

Return type: LeftFractionMatrix

Description: This method computes the multiplication of two LeftFractionMatrices resp. LeftFractionVectors. We may multiply a LeftFractionMatrix with a LeftFractionVector and vice versa as long as they fit according to their dimensions.

7.2.4.2.5 nullSpaceMatrix

Visibility: export

Parameters:

computeKernelOf::**LeftFractionMatrix**
Option: rowwise::**boolean**:=**false**
Option: columnwise::**boolean**:=**false**
Option: showRemainderMatrix::**boolean**:=**false**
Option: withoutSimpleBasis::**boolean**:=**false**
Option: debugMode::**boolean**:=**false**

Return type: **set**(**LeftFractionVector**)

Description: Computes the basis vectors of the null space (kernel) of the given matrix computeKernelOf. There are two options rowwise and columnwise which allow to specify whether the rows or columns of the given matrix shall be used to compute the basis vectors of the null space. Let us recall that during the minimal basis decomposition we need the basis vectors of the null space to find the linear dependent rows resp. columns of the leading coefficient matrix. Therefore the two options correlate with the kind of decomposition:
In case of a row-wise decomposition we need to compute the row leading coefficient matrix of the matrix M we want to decompose in order to compute the basis vectors α of the null space of the rows. I.e. the basis vectors α have to satisfy the equation

$$\alpha^T \, \mathrm{LC}_{row}\left(M\right) = 0. \tag{7.44}$$

All possible vectors α can be computed by the method nullSpaceMatrix and the option rowwise:

Listing 7.19: Compute the row-wise null space basis

```
> alphas := LeftFractionUtils[nullSpaceMatrix](leadingCoeffzMatrix,
rowwise=true);
```

In case of a column-wise decomposition we are looking for all vectors α which satisfy the equation

$$\mathrm{LC}_{column}\left(M\right)\alpha = 0. \tag{7.45}$$

In order to compute all possible vectors α we use the option columnwise:

Listing 7.20: Compute the row-wise null space basis

```
> alphas := LeftFractionUtils[nullSpaceMatrix](leadingCoeffzMatrix,
columnwise=true);
```

The options showRemainderMatrix, withoutSimpleBasis and debugMode have already been described in the description of the method computeNullSpace (7.2.4.1.1).

Examples: Let us assume that we would like to compute the basis vectors of

the null space of the rows of

$$\begin{pmatrix} \delta+\delta^2 & x(t)+(x(t-2\tau))\,\delta & 0 & y(t)+(y(t-2\tau))\,\delta \\ 0 & \delta^2 & 0 & (y(t+7\tau))\,\delta \\ 0 & 0 & 0 & (x(t+7\tau))\,\delta \\ 0 & \delta+(x(t+2\tau))\,\delta^2 & 0 & 0 \end{pmatrix}. \tag{7.46}$$

This can be done by

Listing 7.21: Compute the basis vectors of a LeftFractionMatrix

```
> computeKernelOf := Matrix(4, 4, {
(1,1) = LeftFraction(DelayPolynomial(0, 1, 1)),
(1,2) = LeftFraction(DelayPolynomial(xD0T0,xD0T2)),
(1,3) = LeftFraction(0),
(1,4) = LeftFraction(DelayPolynomial(yD0T0,yD0T2)),
(2,1) = LeftFraction(0),
(2,2) = LeftFraction(DelayPolynomial(0, 0, 1)),
(2,3) = LeftFraction(0),
(2,4) = LeftFraction(DelayPolynomial(0, yD0P7)),
(3,1) = LeftFraction(0),
(3,2) = LeftFraction(0),
(3,3) = LeftFraction(0),
(3,4) = LeftFraction(DelayPolynomial(0, xD0P7)),
(4,1) = LeftFraction(0),
(4,2) = LeftFraction(DelayPolynomial(0, 1, xD0P2)),
(4,3) = LeftFraction(0),
(4,4) = LeftFraction(0)
}):
basisVectors := LeftFractionUtils[nullSpaceMatrix](computeKernelOf,
rowwise=true);
```

and yields the one basis vector

$$\begin{pmatrix} 0 \\ -\delta^{-1}\cdot(1+(x(t+\tau))\,\delta) \\ -\delta^{-1}\cdot\left(-\frac{y(t+7\tau)}{x(t+7\tau)}+\left(-x(t+\tau)\cdot\frac{y(t+6\tau)}{x(t+6\tau)}\right)\delta\right) \\ 1 \end{pmatrix}. \tag{7.47}$$

7.2.5 PiFlatUtils

In case of linear systems with delays, we encounter the need to compute the operator π which eliminates all predictions in the defining operators $\overline{P}$, $\overline{Q}$ and $\overline{R}$ (see Definition 16 and algorithm 5). This module offers methods to compute and verify this specific operator π. Therefore, this submodule is not needed in case of linear systems without delays.

7.2.5.1 Local methods

7.2.5.1.1 computePiForMatrix

Visibility: local

Parameters:

argumentOne::OreMatrix

Return type: DelayPolynomial

Description: This method computes the DelayPolynomial π which eliminates all predictions in the given OreMatrix argumentOne[9]. A possible way to compute π is evaluating all denominators of the given matrix argumentOne and compute the least common left multiple of these. I.e. in case of an OreMatrix $M \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{r\times s}$ the operator π is given by

$$\pi = \text{lclm}(\text{denominatorOf}(M_{1,1}), \ldots, \text{denominatorOf}(M_{r,s})). \tag{7.48}$$

By using this computation rule, we ensure that $\pi M \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{r\times s}$.

Examples: In this example, we compute the operator π for the matrix

$$matrix_1 = \begin{pmatrix} (s(t)\delta - s(t)\delta^2)^{-1} \cdot \frac{d^2}{dt^2} & 1 \\ \left(\left(\frac{s(t)^2}{\dot{s}(t)}\delta^2 - \frac{s(t)^2}{\dot{s}(t)}\delta^3\right)^{-1} \cdot (-1)\right) \cdot \frac{d}{dt} & \delta^{-1} \cdot \frac{d}{dt} \end{pmatrix}. \tag{7.49}$$

Listing 7.22: Compute the operator π

```
> matrix_1 := Matrix(2, 2, {
(1,1) = OrePolynomial(0, 0, LeftFraction(DelayPolynomial(0, sD0T0, -sD0T0),
1)),
(1,2) = OrePolynomial(1),
(2,1) = OrePolynomial(0, LeftFraction(DelayPolynomial(0, 0, sD0T0^2/sD1T0,
-sD0T0^2/sD1T0), -1)),
(2,2) = OrePolynomial(0, LeftFraction(DelayPolynomial(0, 1), 1))
});
> pi := PiFlatUtils[computePiForMatrix](matrix_1);
```

The resulting operator π is

$$\pi = \frac{s(t)^2}{\dot{s}(t)}\delta^2 - \frac{s(t)^2}{\dot{s}(t)}\delta^3. \tag{7.50}$$

7.2.5.2 Exported methods

7.2.5.2.1 computePi

Visibility: export

Parameters:

matrixP_bar::OreMatrix
matrixQ_bar::OreMatrix
matrixR_bar::OreMatrix

Return type: DelayPolynomial

Description: This method computes the operator π for the three defining operators $\overline{P} \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{m\times n}$, $\overline{Q} \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{n\times m}$ and $\overline{R} \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]^{m\times m}$ such that

$$\pi\overline{P} \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{m\times n}, \quad \pi\overline{Q} \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{n\times m}, \quad \pi\overline{R} \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]^{m\times m} \tag{7.51}$$

[9] **Remark:** The operator π is not unique.

(see Definition 16 and algorithm 5 for more details). This method uses the local method computePiForMatrix (7.2.5.1.1) to compute the temporary operators $\pi_{\overline{P}}$, $\pi_{\overline{Q}}$ and $\pi_{\overline{R}} \in \mathcal{K}[\delta]$ which eliminate the predictions in the particular matrix. Finally, the least common left multiple of these three operators is computed and returned.

7.2.5.2.2 verifyPi

Visibility: export

Parameters:

polynomialPi::**DelayPolynomial**
referringMatrix :: **OreMatrix**
Option: showTransformedMatrix::**boolean**:=**false**

Return type: boolean

Description: This method tests whether the given DelayPolynomial polynomialPi eliminates all predictions in the given OreMatrix referringMatrix. I.e. in case of a DelayPolynomial $\pi \in \mathcal{K}[\delta]$ and an OreMatrix $M \in \mathcal{K}(\delta)\left[\frac{d}{dt}\right]$ it evaluates the condition $\pi M \in \mathcal{K}\left[\delta, \frac{d}{dt}\right]$.
In addition, if the option showTransformedMatrix is set to true, the method will also return the product of the operator π and the matrix M as a second return value.

Examples: In this example, we use the matrix and the operator π from the example of the method computePiForMatrix (7.2.5.1.1) in order to show that the computed π satisfies the condition, which is given by Definition 16.

Listing 7.23: Verify a given π

```
> matrix_1 := Matrix(2, 2, {
(1,1) = OrePolynomial(0, 0, LeftFraction(DelayPolynomial(0, sD0T0, -sD0T0),
1)),
(1,2) = OrePolynomial(1),
(2,1) = OrePolynomial(0, LeftFraction(DelayPolynomial(0, 0, sD0T0^2/sD1T0,
-sD0T0^2/sD1T0), -1)),
(2,2) = OrePolynomial(0, LeftFraction(DelayPolynomial(0, 1), 1))
});
> pi := DelayPolynomial(0, 0, sD0T0^2/sD1T0, -sD0T0^2/sD1T0);
> PiFlatUtils[verifyPi](pi, matrix_1);
```

7.3 DifferentialForms

7.3.1 Internal Structure of the Toolbox

The toolbox for flatness determination in case of nonlinear systems consists of the main module and the only submodule MinimalbasisDecomp. All functionalities that affects the handling of differential forms and operators are kept by the main module. The purpose of the submodule MinimalbasisDecomp is to provide a fast way to decompose operator matrices with the ansatz of minimal bases.

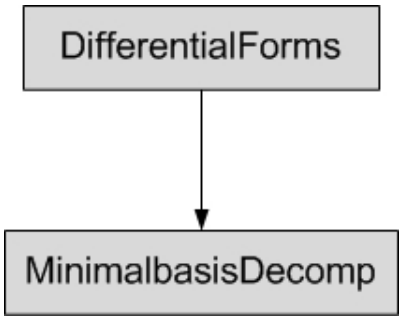

Figure 7.3: Structure of DifferentialForms

In the following sections the main module and its submodule will be explained in detail.

7.3.2 Main Module

In this section, all methods of the main module of the DifferentialForms-toolbox are described in detail. The main module contains all methods in order to perform the flatness determination in case of nonlinear systems, except for the minimal basis decomposition.
Unlike in the toolbox DifferentialDelays, most of the exported methods do not have a restriction of the data types for the parameters. The reason for this is that the methods can be used for most of the data types, i.e. the particular method will check the data types of the parameters and rise an error if the data types do not fit[10]. E.g. the method derivative (7.3.2.4.2) is used to compute the time derivative of DiffForms, OperForms, DiffSums, OperSums, DiffMatrices or OperMatrices.

7.3.2.1 Initialization

7.3.2.1.1 initializeMe

Visibility: local

Parameters: –

Return type: –

Description: This method will automatically be called when using the with-command of *Maple*. It initializes the data types defined in section 5.3. This method also displays the new data types and the current version of the toolbox. It also initializes the submodule MinimalbasisDecomp and its internal data types (since the minimal basis decomposition in case of nonlinear systems uses an internal data structure in order to improve the computational performance). Roughly speaking, it is the maintenance method of the toolbox.

[10] I.e. the internal algorithm determines the data type of the given arguments and specify the computation. Additional parameter checks in the method signature would be dispensable.

7.3.2.2 Constructors

7.3.2.2.1 MonoDiffForm

Visibility: export

Parameters:

extDerivative :: **boolean**:=**true**
variable :: **And**(**symbol**, **Not**(**boolean**)):=**D**
differentialDegree :: **integer**:=1

Return type: MonoDiffForm

Description: This constructor can be used to create a MonoDiffForm. This correlates to a monomial differential form from section 5.3. The constructor allows to create unknown monomial differential forms, such as an n-form $\omega \in \Lambda^n(\mathfrak{X})$. The three parameters are:
extDerivative: This (optional) boolean describes whether the monomial differential form is closed or not. If this parameter is missing, the constructor will assume that it is a closed monomial differential form.
variable: This (mandatory) symbol[11] represents the name of the meromorphic function resp. variable name. Let us recall that all functions and constants in the toolboxes are expressed by regular expressions without a time dependency in order to increase the computational performance - for more details see section 6.1.
The constructor automatically checks the spelling of variable as defined in section 6.1. differentialDegree: This (optional) integer shows the number of actual differentials (in case of an unknown monomial differential form) which is represented by the MonoDiffForm (i.e. n in case of an n-form $\omega \in \Lambda^n(\mathfrak{X})$). In case of a simple differential of a meromorphic function, we may omit this parameter, forcing the constructor to take the MonoDiffForm as a 1-form.

Examples:

Listing 7.24: Create MonoDiffForms

```
> diff_1 := MonoDiffForm(x1D0);
> diff_2 := MonoDiffForm(x3D2);
> diff_3 := MonoDiffForm(false, omegaD0, 2);
> diff_4 := MonoDiffForm(true, omegaD1, 3);
```

These four commands above create the MonoDiffForms:

$$\begin{aligned} diff_1 &= dx_1 \\ diff_2 &= d\ddot{x}_2 \\ diff_3 &= \omega \\ diff_4 &= d\dot{\omega} \end{aligned} \tag{7.52}$$

[11] The method signature says that variable is optional and must not be a boolean. This has technical reasons since it is not possible in *Maple* to have mandatory parameters after optional ones. If the parameter variable is missing, the constructor will raise an error after all.

Please note that in this case $diff_3$ represents a 2-form and $diff_4$ a 3-form since we set this in the parameters.

7.3.2.2.2 MonoDiffForms

Visibility: export

Parameters:
variables :: seq(And(symbol, Not(boolean)))

Return type: seq(MonoDiffForm)

Description: In many cases we will have to enter a sequence of MonoDiffForms in order to create differential forms or operators. This method offers a fast way to create a sequence of closed 1-forms, so we do not have to use the constructor MonoDiffForm when creating 1-forms. Using the method MonoDiffForms is not necessary, but can save us a lot of time describing the 1-forms of the differential forms resp. operators.
This constructor checks the spelling of the variables as defined in section 6.1.
Remark: A sequence of MonoDiffForms (used in the constructors for differential forms and operators) represents MonoDiffForms which are connected with the $\wedge$-operator (see method DiffForm (7.3.2.2.3) for more details).

Examples: To create an ordered sequence of 1-forms we usually have to use the constructor MonoDiffForm:

Listing 7.25: Create a sequence of 1-forms

```
> seq_1 := MonoDiffForm(x1D0), MonoDiffForm(x2D0), MonoDiffForm(x1D1);
```

If we use the method MonoDiffForms, we can just write

Listing 7.26: Create a sequence of 1-forms

```
> seq_1 := MonoDiffForms(x1D0, x2D0, x1D1);
```

in order to create the same sequence.

7.3.2.2.3 DiffForm

Visibility: export

Parameters:
coefficient :: Not(list)
monoDiffForms::seq(MonoDiffForm)

Return type: DiffForm

Description: This constructor can be used to create a monomial p-differential form (5.3). The first parameter is a meromorphic function of the state and its derivatives with respect to time as coefficient of the DiffForm. The second (and optional) parameter is a sequence of the MonoDiffForms (i.e. the differentials) of the DiffForm.

Examples: In the following example, we create a few simple monomial p-differential forms using the constructors DiffForm, MonoDiffForm and MonoDiffForms:

$$\begin{aligned} form_1 &= x_1 \cos(\dot{x_2}) dx_1 \wedge dx_2 \wedge d\dot{x}_1 \\ form_2 &= \sigma(x_1, x_2) dx_2 \wedge dx_1 \\ form_3 &= \_C1 \\ form_4 &= d\omega \wedge \theta \end{aligned} \tag{7.53}$$

with ω a 1-form (and therefore $d\omega$ a 2-form) and θ a 3-form.

Listing 7.27: Create DiffForms

```
> form_1 := DiffForm(x1D0 * cos(x2D1), MonoDiffForms(x1D0, x2D0, x1D1));
> form_2 := DiffForm(sigma(x1D0, x2D0), MonoDiffForms(x2D0, x1D0));
> form_3 := DiffForm(_C1);
> form_4 := DiffForm(1, MonoDiffForm(true, omegaD0, 2), MonoDiffForm(false,
thetaD0, 3));
```

7.3.2.2.4 OperForm

Visibility: export

Parameters:

coefficient ::Not(list)
monoDiffForms::seq(MonoDiffForm)
operatorDegree::integer:=0

Return type: OperForm

Description: This constructor can be used to create a monomial operator (5.4) which transforms p-forms into p+q-forms. The first parameter is a meromorphic function of the state and its derivatives with respect to time as coefficient of the OperForm. The second (and optional) parameter is a sequence of the MonoDiffForms (i.e. the differentials) of the OperForm. The third parameter is the degree in $\frac{d}{dt}$ of the monomial operator. If this parameter is missing, the constructor will create an operator with degree 0 in $\frac{d}{dt}$.

Examples: We would like to illustrate the usage of the constructor OperForm in order to create the following monomial operators:

$$\begin{aligned} oper_1 &= \mu(x_1, \dot{x}_1, x_2) dx_1 \wedge d\dot{x}_1 \wedge \frac{d}{dt} \\ oper_2 &= \sin(\dot{x_2}) dx_2 \wedge \omega \wedge \\ oper_3 &= \wedge \frac{d}{dt} \\ oper_4 &= dx_1 \wedge \end{aligned} \tag{7.54}$$

with ω a 3-form.

Listing 7.28: Create OperForms

```
> oper_1 := OperForm(mu(x1D0, x1D1, x2D0), MonoDiffForms(x1D0, x1D1), 1);
> oper_2 := OperForm(sin(x2D1), MonoDiffForm(x2D0), MonoDiffForm(false,
omegaD0, 3));
> oper_3 := OperForm(1, 1);
> oper_4 := OperForm(1, MonoDiffForm(x1D0));
```

7.3.2.2.5 DiffSum

Visibility: export

Parameters:

forms::seq(DiffForm)

Return type: Or(DiffForm, DiffSum)

Description: This constructor is used to create a general p-form (6.12). It also simplifies the given DiffForms and merges them, if possible.
If the result (after the simplification) turns out to be just one DiffForm, the method will return this very DiffForm instead of a DiffSum with only one summand.

7.3.2.2.6 OperSum

Visibility: export

Parameters:

forms::seq(OperForm)

Return type: Or(OperForm, OperSum)

Description: Similar to the constructor DiffSum (7.3.2.2.5), this constructor is used to create an operator (5.2). It also simplifies the given OperForms and merges them, if possible.
If the result (after the simplification) turns out to be just one OperForm, the method will return this very OperForm instead of an OperSum with only one summand.

7.3.2.3 Local methods

7.3.2.3.1 cleanEquations

Visibility: local

Parameters:

equations:: list

Return type: list

Description: This method removes all zeros from the given list equations. This method is used to remove dispensable equations of pde-systems.

7.3.2.3.2 compareTwoDifferentials

Visibility: local

Parameters:
leftMonoDiffForms::**list**(**MonoDiffForm**)
rightMonoDiffForms::**list**(**MonoDiffForm**)

Return type: integer

Description: This method is used by internal simplifier methods. In order to merge two DiffForms resp. OperForms, the two differentials of those two differential forms resp. operators must be the same but only permuted. If that is fulfilled, we can merge the two DiffForms resp. OperForms according to the computation rules in the description of method simplifier (7.3.2.4.10). For this purpose this method computes whether the two lists of MonoDiffForms are the same but permuted. If they are not the same, the method will return 0. If they are the same, but permuted, the method will return a multiplier. The right list of MonoDiffForms has to be multiplied with that multiplier, in order to permute the differentials[12].

7.3.2.3.3 convertTermToLatex

Visibility: local

Parameters:
term::**Not**(**list**)
Option: showTimeDependency::**boolean**:=**true**
Option: substituteGreekLetters::**boolean**:=**true**

Return type: string

Description: This method is used by the exported method latexPrinting (7.3.2.4.6). It converts the given term (any function or symbol) into *LaTeX*-code. Like the similar method in the linear toolbox, it contains several features, which shall be described in detail:

1.) substitute functions

2.) substitute certain characters

3.) substitute Greek letters

1.) As explained in section 6.1, all functions are represented by symbols which satisfy a certain regular expression containing the differentiation degree. The first conversion step is to convert these symbols into *LaTeX*-code. A few examples shall illustrate this:

[12]The reason behind this is the property (3.74) of the wedge product $\wedge$. This property may cause the algebraic sign of the differentials to change if we permute them.

Function as symbol	Converted function
x1D0	`x1(t)`
var11D2	`\ddot{var11}(t)`
yD4	`y^{(4)}(t)`

If the option showTimeDependency is set to false, the method will not show the dependency of t, thus forcing the method to transform e.g. x1D1 into `\dot{x1}` instead of `\dot{x1}(t)`.

2.) To increase the readability of the produced *LaTeX*-code we also substitute the characters `_` with `\_` and `*` with `\cdot`.

3.) This method uses the local method substituteGreekLettersInTerm (7.3.2.3.9) to substitute Greek letters. If the option substituteGreekLetters is enabled, all lowercased Greek letters except 'eta' and 'psi' will be substituted with the corresponding *LaTeX*-expressions. 'eta' and 'psi' may not be substituted to avoid conversion problems since they are literally contained in other Greek letters (e.g. in 'theta').
If a Greek letter is followed by another character, the toolbox will use the underscore to separate the Greek letter from the following characters. A few examples with Greek letters shall illustrate the substitution:

Function as symbol	Converted function
2*sin(thetaD1)	`2\cdot sin(\dot{\theta}(t))`
_upsilon1	`\_\upsilon\_1`
omega1D0 + muxD0	`\omega\_1(t) + \mu\_x(t)`

7.3.2.3.4 extDerivativeDifferentials

Visibility: local

Parameters:
argumentOne::list(MonoDiffForm)

Return type: list (DiffForm)

Description: This methods handles the anti-derivation property of the exterior derivative defined in (3.79). I.e. this method computes the exterior derivative of the given list of MonoDiffForms which represents the ordered n-forms of a DiffForm resp. OperForm. Since the result of an exterior derivative of a differential may be a differential form, this method returns a list of DiffForms[13].

Examples: First, we are going to compute the exterior derivative of the dif-

[13] Actually, this is **not** a DiffSum since due to technical reasons this method may return a list with only one DiffForm or even an empty list, which is not a DiffSum according to the data type definition.

ferential

$$dx_1 \wedge dx_2 \wedge d\dot{x}_1 \wedge dx_3, \tag{7.55}$$

which will obviously be zero since the exterior derivative of a closed form is always zero (see (3.80)). The corresponding method call

Listing 7.29: Compute the exterior derivative

```
> forms_1 := [MonoDiffForms(x1D0, x2D0, x1D1, x3D0)];
> result_1 := extDerivativeDifferentials(forms_1);
```

will return an empty list.
In a second example, we compute the exterior derivative of the differential

$$\dot{\omega} \wedge \omega \wedge \theta \tag{7.56}$$

with $\omega \in \Lambda^3(\mathfrak{X})$ and $\theta \in \Lambda^4(\mathfrak{X})$. The method call

Listing 7.30: Compute the exterior derivative

```
> forms_2 := [MonoDiffForm(false, omegaD1, 3), MonoDiffForm(false, omegaD0,
3), MonoDiffForm(false, thetaD0, 4)];
> result_2 := extDerivativeDifferentials(forms_2);
```

returns the DiffSum

$$d\dot{\omega} \wedge \omega \wedge \theta - \dot{\omega} \wedge d\omega \wedge \theta + \dot{\omega} \wedge \omega \wedge d\theta. \tag{7.57}$$

7.3.2.3.5 extractNONPDEs

Visibility: local

Parameters:
pdeSystem::list

Return type: list

Description: This method is used in the method solvePDEs (7.3.2.3.7) to separate partial differential equations from algebraic equations. In order to do this, the method simply evaluates the given list of equations and returns a list which contains only the algebraic equations that are in the given system.

7.3.2.3.6 integrableDiffsIncludedIn

Visibility: local

Parameters:
argumentOne::Or(DiffForm, DiffSum)

Return type: list (symbol)

Description: This method determines which integrable differentials are included in the given argumentOne, which can be of type DiffForm or DiffSum. In this case, *integrable* means that the contained differential forms have exactly

one closed 1-form each. I.e. argumentOne has to be $\in \Lambda^1(\mathfrak{X})$. If argumentOne does not fit that assertion, the method will raise an error. Otherwise the method returns a list which contains the variables of the contained MonoDiffForms.

Examples: In case of the following DiffSum

$$x_1^{(3)}\dot{x}_2 dx_1 + x_1^{(3)} dx_2 - d\dot{x}_2 \tag{7.58}$$

the method call

Listing 7.31: Get the variables of integrable differentials

```
> form_1 := DiffSum(DiffForm(x1D3 * x2D1, MonoDiffForm(x1D0)), DiffForm(x1D3,
MonoDiffForm(x2D0)), DiffForm(-1, MonoDiffForm(x2D1)));
> integrableDiffsIncludedIn(form_1);
```

returns the following list:

$$[x1D0, x2D0, x2D1]$$

7.3.2.3.7 solvePDEs

Visibility: local

Parameters:

pdeList :: list
Option: useSeparation::boolean:=false
Option: chooseByComplexity::boolean:=true

Return type: set

Description: This method solves the given system of partial differential equations. In addition, this method is capable of solving ordinary differential equations and algebraic equations as well. This method is used in the solver for equations of differential forms and operators and the integrator for differential forms of this toolbox.
The equation system has to be entered as list with each equation (the whole equation or, if the right side of the equation is zero, only the left side) as one entry.
The option useSeparation forces the method so separate between algebraic equations and differential equations. First, the method tries to solve the system of algebraic systems without using the differential ones. Afterwards, the method will substitute the result of the algebraic system (if a possible solution exists) and continue solving the whole system.
The option chooseByComplexity ensures that the method always chooses the solution with the lowest complexity. If this option is set to false, the internal algorithm will randomly pick among the possible solutions.

7.3.2.3.8 spellingChecker

Visibility: local

Parameters:
term::Not(list)
Option: onlyFunctionsAllowed::boolean:=false

Return type: –

Description: This method is used by the constructor methods of the toolbox in order to check whether all functions and constants are spelled correctly referring to the regular expressions

$$\hat{}[\text{a-zA-CE-Z0-9\_}]+\text{D}[0\text{-}9]+\$ \tag{7.59}$$

for functions and

$$\hat{}[\text{a-zA-CE-Z0-9\_}]+\$ \tag{7.60}$$

for constants. This method evaluates all occurring functions and constants in the given term and will raise an error if there are wrong spelled functions or constants. Otherwise this method will simply return NULL.
If the option onlyFunctionsAllowed is set to true, the method will only allow functions in the given term. This option is used in the constructors MonoDiffForm and MonoDiffForms in which no constants are allowed since a differential is the exterior derivative of a meromorphic function.

7.3.2.3.9 substituteGreekLettersInTerm

Visibility: local

Parameters:
termAsLatexString::string

Return type: string

Description: This method substitutes all lowercased Greek letters except 'eta' and 'psi' since they are literally contained in other Greek letters. For more details see the description of method convertTermToLatex (7.3.2.3.3).

7.3.2.3.10 timeDerivative

Visibility: local

Parameters:
term::Not(list)

Return type: Not(list)

Description: This method differentiates the given term with respect to time. Since there are no more *Maple*-functions of t in the occurring terms (see section 6.1 for further details), the time derivative has to be implemented using the Lie derivative along the Cartan field (for a detailed description see section 6.1.3).

7.3.2.4 Exported methods

7.3.2.4.1 createDiffFormVector

Visibility: export

Parameters:

functions :: **seq**(**symbol**)
Option: extDerivativeDegree::**integer**:=0
Option: timeDerivativeDegree::**integer**:=0

Return type: DiffMatrix

Description: During the process of flatness determination, we will often need a simple vector filled with differential forms (for instance if we want to create a general vector x, dy or ω). This method supports us, since we do not have to create such a vector manually. We just have to enter a sequence of the coefficients of the desired differential forms. This will create a vector[14] of DiffForms with the given coefficients.
The option extDerivativeDegree allows to compute directly the n'th exterior derivative of the vector (e.g. extDerivativeDegree = 1 computes the normal exterior derivative of the vector).
The option timeDerivativeDegree works similar. It computes the n'th derivative with respect to t of the vector (e.g. timeDerivativeDegree = 2 computes the 2nd time derivative of the vector).

Examples: In a first step we create the vector

$$x = \left(x_1, x_2, x_3 \right)^T. \tag{7.61}$$

Listing 7.32: Create a vector without options

```
> vector_1 := createDiffFormVector(x1D0, x2D0, x3D0);
```

In order to create the vector

$$dx = \left(dx_1, dx_2, dx_3 \right)^T, \tag{7.62}$$

we simply have to use the option extDerivativeDegree:

Listing 7.33: Create a vector with exterior derivative

```
> vector_1 := createDiffFormVector(x1D0, x2D0, x3D0, extDerivativeDegree=1);
```

[14] Please note that all vectors over differential forms and operators in the toolbox DifferentialForms are represented by DiffMatrices and OperMatrices, i.e. one-columned matrices instead of actual vectors.

7.3.2.4.2 derivative

Visibility: export

Parameters:

argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Return type: Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Description: This method computes the time derivative of the given argumentOne, whether it is a DiffForm, DiffSum, DiffMatrix or one of the corresponding operator types. The result of the method will automatically be simplified, using the method simplifier (7.3.2.4.10). The time derivative will be computed according to the following rules:
The time derivative of a monomial p-differential form (DiffForm) is given by the computation rules defined in (3.77)–(3.78):

$$\begin{aligned} & \frac{d}{dt}\left(\omega_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)}\right) \\ = \; & \frac{d}{dt}\left(\omega_c(\overline{x})\right) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \\ & + \omega_c(\overline{x}) d\dot{x}_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \\ & + \omega_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge d\dot{x}_{i_2}^{(j_2)} \wedge \ldots \wedge dx_{i_p}^{(j_p)} \\ & + \ldots \\ & + \omega_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge d\dot{x}_{i_p}^{(j_p)} \end{aligned} \tag{7.63}$$

with the coefficient of the monomial p-differential form ω_c as a meromorphic function of the state and its derivatives with respect to t.
The time derivative of a monomial operator, which transforms p-forms into p+q-forms, (OperForm) is given by the computation rules defined in (3.87) and (7.63):

$$\begin{aligned} & \frac{d}{dt}\left(\mu_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_q}^{(j_q)} \wedge \frac{d^n}{dt^n}\right) \\ = \; & \frac{d}{dt}\left(\mu_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_q}^{(j_q)}\right) \wedge \frac{d^n}{dt^n} \\ & + \mu_c(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_q}^{(j_q)} \wedge \frac{d^{n+1}}{dt^{n+1}} \end{aligned} \tag{7.64}$$

with the coefficient of the monomial operator μ_c as a meromorphic function of the state and its derivatives with respect to t.

In case of a DiffSum, we use the sum rule for differentiation:

$$\begin{aligned} &\frac{d}{dt}\left(\sum_{i_1,j_1,\ldots,i_p,j_p} \omega_{i_1,j_1,\ldots,i_p,j_p}(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)}\right) \\ = &\sum_{i_1,j_1,\ldots,i_p,j_p} \frac{d}{dt}\left(\omega_{i_1,j_1,\ldots,i_p,j_p}(\overline{x}) dx_{i_1}^{(j_1)} \wedge \ldots \wedge dx_{i_p}^{(j_p)}\right) \end{aligned} \tag{7.65}$$

In case of an OperSum, we also use the sum rule.
The time derivative of a matrix over differential forms or operators is given by differentiating the entries of the matrix.

7.3.2.4.3 equals

Visibility: export

Parameters:
leftArgument::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)
rightArgument::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Return type: boolean

Description: This method evaluates whether the two given arguments leftArgument and rightArgument are equal. In order to do this, the method evaluates the (simplified) result of

$$leftArgument - rightArgument = 0. \tag{7.66}$$

Two arguments must be of the same data type. There are only two exceptions: DiffForms may be compared with DiffSums and OperForms with OperSums since they could be equal after a simplification of the sum. Any other combination of data types will automatically return false.

7.3.2.4.4 extDerivative

Visibility: export

Parameters:
argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Return type: Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Description: This method computes the exterior derivative of the given argumentOne, whether it is a DiffForm, DiffSum, DiffMatrix or one of the corresponding operator types. The result of the method will automatically be simplified using the method simplifier (7.3.2.4.10). The exterior derivative will

be computed according to the computation rules defined in (3.79)–(3.82) and (3.85).

Examples: In the first example, we would like to compute the (very simple) exterior derivative of $\omega \in \Lambda^2(\mathfrak{X})$ with the commands

Listing 7.34: Exterior derivative of a 2-form

```
> form_1 := DiffForm(1, MonoDiffForm(false, omegaD0, 2));
> result_1 := extDerivative(form_1);
```

$$form\_1 := [1, [[false, omegaD0, 2]]]$$
$$result\_1 := [1, [[true, omegaD0, 3]]]$$

resulting in $d\omega \in \Lambda^3(\mathfrak{X})$.
In the second example, we would like to compute the exterior derivative of the operator (OperForm)

$$form_1 = \frac{\cos(x_2)}{\sqrt{1-\dot{x}_3^2}} d\dot{x}_1 \wedge \frac{d}{dt}. \tag{7.67}$$

The commands

Listing 7.35: Exterior derivative

```
> form_1 := OperForm(cos(x2D0)/sqrt(1-(x3D1)^2), MonoDiffForm(true, x1D1,
1),1);
> result_1 := extDerivative(form_1);
```

yield the result (an OperSum)

$$\begin{aligned} result_1 \;=\; & -\frac{\sin(x_2)}{\sqrt{1-\dot{x}_3^2}} dx_2 \wedge d\dot{x}_1 \wedge \frac{d}{dt} \\ & + \frac{\cos(x_2)\dot{x}_3}{\left(1-\dot{x}_3^2\right)^{3/2}} d\dot{x}_3 \wedge d\dot{x}_1 \wedge \frac{d}{dt}. \end{aligned} \tag{7.68}$$

7.3.2.4.5 integration

Visibility: export

Parameters:
argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)
Option: showSolution::boolean:=false
Option: useSeparation::boolean:=false
Option: chooseByComplexity::boolean:=true

Return type: Or(DiffForm, DiffMatrix)

Description: This method integrates the given DiffForm, DiffSum or DiffMatrix with respect to its differentials. I.e. this method is able to invert the

exterior derivative (for instance to compute the flat output of a nonlinear system based on the flat output of the variational system).
This method will only work if all the DiffForms that occur in argumentOne have exactly one closed 1-form. Before the actual integration, the method simplifies the given argument using the method simplifier (7.3.2.4.10).
In case of a DiffForm or DiffSum $\omega \in \Lambda^1(\mathfrak{X})$, the internal algorithm tries to compute a function y which satisfies

$$d(y) = \omega. \tag{7.69}$$

In case of a DiffMatrix, the algorithm expects a vector of DiffForms and/or DiffSums $\Omega \in \Lambda^1(\mathfrak{X})^{n\times 1}$ (i.e. a one-columned DiffMatrix). Then, the internal algorithm tries to compute a vector $y = (y_1, \ldots, y_n)^T$ which satisfies

$$d(y) = \Omega. \tag{7.70}$$

This method offers three options to customize the integration:
showSolution adds the result of the internal pde-solver as a second return value. It is recommended to enable this option in order to reveal dependencies among internal functions (see the second example below).
The options useSeparation and chooseByComplexity are given to the internal pde-solver. Please see the method description of solvePDEs (7.3.2.3.7) for more details.

Examples: In the first example, we integrate the DiffSum

$$sum_1 = d\psi + \sin(\dot{\psi})d\dot{\psi}. \tag{7.71}$$

Listing 7.36: Integrate a DiffSum

```
> form_1 := DiffForm(1, MonoDiffForm(psiD0));
> form_2 := DiffForm(sin(psiD1), MonoDiffForm(psiD1));
> sum_1 := DiffSum(form_1, form_2);
> resultOne, solutions := integration(sum_1, showSolution=true);
```

This yields the result

$$resultOne, solutions := [psiD0 - \cos(psiD1) + \_C1, [\,]], \{y1(psiD0, psiD1) = psiD0 - \cos(psiD1) + \_C1\}$$

which represents the function

$$resultOne = \psi - \cos(\dot{\psi}) + \_C1. \tag{7.72}$$

In the second example, we integrate the *DiffMatrix*

$$M\_omega = \begin{pmatrix} dx_1 - \arcsin(\dot{x}_3)dx_2 + \Psi_1 dx_3 + \Psi_2 d\dot{x}_3 \\ dx_3 \end{pmatrix} \tag{7.73}$$

with

$$\Psi_1 = \left(\frac{\partial}{\partial x_3}K(x_3,\dot{x}_3) + \_F5(x_3)\right) \tag{7.74}$$

$$\Psi_2 = \frac{-x_2 + \frac{\partial}{\partial \dot{x}_3}K(x_3,\dot{x}_3)\sqrt{1-\dot{x}_3^2}}{\sqrt{1-\dot{x}_3^2}} \tag{7.75}$$

Listing 7.37: Integrate a DiffMatrix

```
> Psi_1 := (D[1](K))(x3D0, x3D1)+_F5(x3D0);
> Psi_2 := (-x2D0+(D[2](K))(x3D0, x3D1)*sqrt(1-x3D1^2))/sqrt(1-x3D1^2);
> M_omega := Matrix(2, 1, {
(1,1) = DiffSum(DiffForm(1, MonoDiffForms(x1D0)), DiffForm(-arcsin(x3D1),
MonoDiffForms(x2D0)), DiffForm(Psi_1, MonoDiffForms(x3D0)), DiffForm(Psi_2,
MonoDiffForms(x3D1))),
(2,1) = DiffForm(1, MonoDiffForm(x3D0))
});
> y := integration(M_omega);
```

This yields the vector

$$y = \begin{pmatrix} x_1 - \arcsin(\dot{x}_3)\cdot x_2 + \_F6(x_3,\dot{x}_3) \\ x_3 + \_C1 \end{pmatrix}. \tag{7.76}$$

But we encounter one problem. We do not know how the functions $K(x_3,\dot{x}_3)$, $\_F5(x_3)$ and $\_F6(x_3,\dot{x}_3)$ are connected. Therefore, we use the option showSolution to reveal the dependencies:

Listing 7.38: Integrate a DiffMatrix

```
> y, solutions := integration(M_omega, showSolution = true);
```

This yields the output

$$y, solutions := \left[\begin{matrix} [x1D0 - \arcsin(x3D1)\,x2D0 + \_F6(x3D0, x3D1), [\,]] \\ [x3D0 + \_C1, [\,]] \end{matrix}\right], \left\{K(x3D0, x3D1) = \_F6(x3D0, x3D1) + \_F7(x3D0),\right.$$
$$\_F5(x3D0) = -\left(\frac{d}{dx3D0}\_F7(x3D0)\right), y1(x1D0, x2D0, x3D0, x3D1) = x1D0 - \arcsin(x3D1)\,x2D0 + \_F6(x3D0, x3D1), y2(x3D0)$$
$$\left. = x3D0 + \_C1\right\}$$

showing the dependencies between the functions.

7.3.2.4.6 latexPrinting

Visibility: export

Parameters:

argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Option: substituteGreekLetters::boolean:=true

Return type: string

Description: The internal representation of the data types of this toolbox

was not designed to be human readable but to suit the given requirements best. Nevertheless, the toolbox offers this method to convert the data types DiffForm, DiffSum, DiffMatrix and the corresponding operator types into *LaTeX*-code, to offer a readable output.
Furthermore, this method has the boolean option substituteGreekLetters (with true as default value). This option forces the method to convert all lowercased Greek letters except 'eta' and 'psi' since they are literally contained in other Greek letters. The actual substitution of Greek letters is done by the method substituteGreekLettersInTerm (7.3.2.3.9).
The regular expressions as defined in section 6.1 will also be converted.

Examples: At this point, we want to demonstrate the functionality of this method by converting the following arguments into *LaTeX*-code:

Listing 7.39: Instantiate several data structure

```
> diff_form := DiffForm(mu1D1*mu2D0, MonoDiffForm(true, xD0, 1),
MonoDiffForm(true, xD1, 1), MonoDiffForm(false, omegaD1, 3));
> oper_form := OperForm(xD1, 3);
> diff_sum := DiffSum(DiffForm(omegaD0, MonoDiffForms(omegaD1)), DiffForm(1,
MonoDiffForms(mu1D0)), DiffForm(mu2D0, MonoDiffForms(zD0)));
> diff_matrix := Matrix(3, 2, {
(1,1) = DiffForm(0),
(1,2) = DiffForm(1),
(2,1) = DiffForm(xD0, MonoDiffForms(xD0)),
(2,2) = DiffForm(0),
(3,1) = DiffForm(0),
(3,2) = DiffSum(DiffForm(-1, MonoDiffForms(xD0)), DiffForm(1,
MonoDiffForms(yD0)), DiffForm(zD0, MonoDiffForms(zD0)))
});
```

By using this method

Listing 7.40: latexPrinting

```
> latexPrinting(diff_form);
> latexPrinting(oper_form);
> latexPrinting(diff_sum);
> latexPrinting(diff_matrix);
```

we get these unmodified representations in *LaTeX*:

$$
\begin{aligned}
\text{diff\_form} &= \left(\dot{\mu\_1}(t) \cdot \mu\_2(t)\right) \cdot d\left(x\right) \wedge d\left(\dot{x}\right) \wedge \dot{\omega} \\
\text{oper\_form} &= \dot{x}(t) \wedge \frac{d^3}{dt^3} \\
\text{diff\_sum} &= \left(\omega(t)\right) \cdot d\left(\dot{\omega}\right) + d\left(\mu\_1\right) + \left(\mu\_2(t)\right) \cdot d\left(z\right) \\
\text{diff\_matrix} &= \begin{pmatrix} 0 & 1 \\ \left(x(t)\right) \cdot d\left(x\right) & 0 \\ 0 & (-1) \cdot d\left(x\right) + d\left(y\right) + \left(z(t)\right) \cdot d\left(z\right) \end{pmatrix}
\end{aligned}
$$

7.3.2.4.7 multiply

Visibility: export

Parameters:
coefficient :: And(Not(list), Not(Matrix))
argumentTwo::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Return type: Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Description: This method is used for the multiplication of an arbitrary coefficient with a DiffForm, OperForm, DiffSum, OperSum, DiffMatrix or OperMatrix. The result will automatically be simplified.
Remark: The multiplication of differential forms or operators (resp. matrices over them) with other differential forms or operators is not defined! Please use the method wedge (7.3.2.4.14) to apply operators to differential forms or other operators.

Examples: In the first example, we just compute the negative of the given DiffSum by multiplying with -1:

Listing 7.41: Multiply a DiffSum

```
> form_1 := DiffForm(-1/(arcsin(x3D1)^2*sqrt(1-(x3D1)^2)), MonoDiffForm(true,
x1D0, 1));
> form_2 := DiffForm(cos(x2D0)/sqrt(1-(x3D1)^2), MonoDiffForm(true, x1D1, 1));
> form_3 := DiffForm(cos(x2D0), MonoDiffForm(true, x1D3, 1));
> sum_1 := DiffSum(form_1, form_2, form_3);
> result_1 := multiply(-1, sum_1);
```

In the second example, we want to multiply an OperMatrix with the coefficient $\sin(x_1) - \dot{x}_2$:

Listing 7.42: Multiply an OperMatrix

```
> form_1 := OperForm(-1/(arcsin(x3D1)^2*sqrt(1-(x3D1)^2)),
MonoDiffForm(x1D0), 1);
> form_2 := OperForm(cos(x2D0)/sqrt(1-(x3D1)^2), MonoDiffForm(x1D1), 0);
> form_3 := OperForm(cos(x2D0), MonoDiffForm(x1D3), 3);
> form_4 := OperForm(cos(x2D0), MonoDiffForm(x1D3), 2);
> matrix_1 := Matrix(2, 2, {
(1,1) = OperSum(form_1, form_2),
(1,2) = OperSum(form_1, form_3),
(2,1) = OperSum(form_3, form_2),
(2,2) = OperSum(form_1, form_4)
});
> result_1 := multiply(sin(x1D0) - x2D1, matrix_1);
```

7.3.2.4.8 plus

Visibility: export

Parameters:
summandOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

summandTwo::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Return type: Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Description: This method computes the sum of the two given terms summandOne and summandTwo. The result will automatically be simplified. We may only add differential forms (DiffForms and/or DiffSums) to other differential forms and operators (OperForm and/or OperSum) to other operators. In case of matrices we may only add DiffMatrices to DiffMatrices and OperMatrices to OperMatrices.
Please note that in case of DiffForms and OperForms it does not make any difference whether we use the method plus or DiffSum (resp. OperSum) to create DiffSums (resp. OperSums).

Examples: In this example, we add the two DiffSums

$$\begin{aligned} sum_1 &= \frac{\cos(x_2)}{\sqrt{1-\dot{x}_3^2}} d\dot{x}_1 + \theta dx_2 \\ sum_2 &= \sin(x_1) dx_1 + \delta d\dot{x}_1 \end{aligned} \tag{7.77}$$

with the commands

Listing 7.43: Compute the sum of two DiffSums

```
> sum_1 := DiffSum(DiffForm(cos(x2D0)/sqrt(1-(x3D1)^2), MonoDiffForm(x1D1)),
DiffForm(theta, MonoDiffForm(x2D0)));
> sum_2 := DiffSum(DiffForm(sin(x1D0), MonoDiffForm(x1D0)), DiffForm(delta,
MonoDiffForm(x1D1)));
> result_1 := plus(sum_1, sum_2);
```

to get the DiffSum

$$result_1 = \left(\frac{\cos(x_2)}{\sqrt{1-\dot{x}_3^2}} + \delta \right) d\dot{x}_1 + \theta dx_2 + \sin(x_1) dx_1. \tag{7.78}$$

7.3.2.4.9 printing

Visibility: export

Parameters:
argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Return type: symbol

Description: As told in the description of method latexPrinting (7.3.2.4.6) the internal data structure of the toolbox is not designed to be human readable. Besides the thorough conversion into *LaTeX*-code using latexPrinting (7.3.2.4.6), we can use this method to transform DiffForms, DiffSums, DiffMatrices, OperForms, OperSums and OperMatrices into a more readable *Maple*-output. Note that this method does not substitute the regular expressions defined in section 6.1.

7.3.2.4.10 simplifier

Visibility: export

Parameters:

argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Option: deleteZeros :: boolean:=true

Option: gatherForms::boolean:=true

Option: simplifyCoefficients :: boolean:=true

Option: subSimplifier :: boolean:=true

Return type: Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Description: This method is the equivalent to the *Maple* command simplify. It simplifies the given parameter (which can be of type DiffForm, DiffSum, DiffMatrix, OperForm, OperSum or OperMatrix) according to certain rules. Almost every method in this toolbox simplifies the result automatically with this method. Some of the features can be disabled by using the options of the method, but this is not recommended!
The features of this method are:

1.) checking for doubled differentials

2.) simplifying the coefficients (if enabled)

3.) erasing dispensable zeros (if enabled)

4.) merging differential forms and operators (if enabled)

1.) The method will determine the differentials which are contained by the differential forms and operators of argumentOne. If the method finds the same differential two times (or more) in the same differential form or operator, it will replace this differential form (resp. operator) with zero since we have the property (3.75) of the wedge-product.
Let us also recall that the differential $\omega \wedge \theta \wedge \omega = 0$ with $\omega \in \Lambda^1(\mathfrak{X}), \theta \in \Lambda^p(\mathfrak{X}), p \in \mathbb{N}_0$ since we have

$$\omega \wedge \theta \wedge \omega = (-1)^p \underbrace{\omega \wedge \omega}_{=0} \wedge \theta = 0. \tag{7.79}$$

2.) This feature is triggered by the option simplifyCoefficients. If this option is set to true, the method will simplify all coefficients with the *Maple*-command simplify. This is important since it is possible to have mathematical terms as coefficients which are actually zero but without having *Maple* recognizing that. But on the other hand, the *Maple*-command simplify needs a lot of computation time, thus it can make sense to omit the simplification of the coefficients during computations if we are aware of the fact that some of the coefficients

might be zero.
The exported methods of this toolbox apply the coefficient-simplification automatically in order to ensure valid coefficients.

3.) If the argument is of type DiffSum, DiffMatrix, OperSum or OperMatrix, this method will remove all containing DiffForms resp. OperForms which are actually zero. It is possible (but not recommended) to disable this feature by setting the option deleteZeros to false.

4.) This is triggered by the option gatherForms. It will force this method to merge DiffForms and OperForms in all occurring DiffSums resp. OperSums if they have the same differentials (and degree in $\frac{d}{dt}$ in case of OperForms). It will automatically permute the differentials (and will therefore modify the sign of the coefficient) if the order of the differentials is different among the forms which shall be merged. This feature is very important to speed up the computations. Disabling this option (without a specific purpose) is not recommended.

The option subSimplifier will only take effect if we use the method on DiffSums, OperSums, DiffMatrices or OperMatrices. If it is disabled, the method will not apply the simplifications 1.) and 2.).
Remark: If a DiffSum or OperSum turns out to be a DiffForm or OperForm after the simplification, this method will return a DiffForm resp. OperForm instead of a DiffSum resp. OperSum with only one entry.

7.3.2.4.11 solver

Visibility: export

Parameters:

solveEquation::**Or**(**list** = **list**, **Matrix** = **Matrix**, **list**, **Matrix**)
Option: useSeparation::**boolean**:=**false**
Option: chooseByComplexity::**boolean**:=**true**

Return type: set

Description: This method solves the given equation and returns a set with the solutions for the degrees of freedom. If we enter only one side of the equation, the method will assume the other side to be zero. If we enter an equation, the data types of the left and right side will have to be compatible. The following tables will give an overview over possible combinations[15]:

left side \ right side	DiffForm	DiffSum	DiffMatrix
DiffForm	X	X	-
DiffSum	X	X	-
DiffMatrix	-	-	X

[15] X describes a possible combination.

left side \ right side	OperForm	OperSum	OperMatrix
OperForm	X	X	-
OperSum	X	X	-
OperMatrix	-	-	X

Operators and differential forms are not compatible.
The options useSeparation and chooseByComplexity are given to the internal pde-solver. Please see the method description of solvePDEs (7.3.2.3.7) for more details.

Examples: Let us assume, we want to compute the solution of the equation

$$sum_1 = sum_2 \tag{7.80}$$

with

$$\begin{aligned} sum_1 &= \phi(x,\dot{x})dx_1 \wedge dx_2 + \theta(x,\dot{x})d\dot{x}_1 \wedge dx_2 \\ &\quad + \psi(x,\dot{x})d\dot{x}_1 \wedge d\dot{x}_2 + \phi(x,\dot{x})dx_1 \wedge d\dot{x}_2 \qquad (7.81) \\ sum_2 &= (\sin(\dot{x}_1) + \cos(\dot{x}_2))\, dx_2 \wedge d\dot{x}_1 + \frac{1}{\sin(x_2)} d\dot{x}_2 \wedge dx_1 \\ &\quad + \frac{\partial}{\partial x_1}\upsilon(x,\dot{x})dx_1 \wedge dx_2 \qquad (7.82) \end{aligned}$$

and with $(x,\dot{x}) = (x_1,\dot{x}_1,x_2,\dot{x}_2)$. This can be done by the commands

Listing 7.44: Solve an equation

```
> form_1 := DiffForm(phi(x1D0, x1D1, x2D0, x2D1), MonoDiffForm(x1D0),
MonoDiffForm(x2D0));
> form_2 := DiffForm(theta(x1D0, x1D1, x2D0, x2D1), MonoDiffForm(x1D1),
MonoDiffForm(x2D0));
> form_3 := DiffForm(psi(x1D0, x1D1, x2D0, x2D1), MonoDiffForm(x1D1),
MonoDiffForm(x2D1));
> form_4 := DiffForm(phi(x1D0, x1D1, x2D0, x2D1), MonoDiffForm(x1D0),
MonoDiffForm(x2D1));
> sum_1 := DiffSum(form_1, form_2, form_3, form_4);

> form_5 := DiffForm(sin(x1D1) + cos(x2D1), MonoDiffForm(x2D0),
MonoDiffForm(x1D1));
> form_6 := DiffForm(1/sin(x2D0), MonoDiffForm(x2D1), MonoDiffForm(x1D0));
> form_7 := DiffForm(diff(upsilon(x1D0, x1D1, x2D0, x2D1), x1D0),
MonoDiffForm(x1D0), MonoDiffForm(x2D0));
> sum_2 := DiffSum(form_5, form_6, form_7);

> solver(sum_1 = sum_2);
```

This will return the *Maple*-result[16]:

[16]The number of possible solutions is the number of solutions of the equation system which are found by *Maple*.

```
Found 4 function(s) in the pde system: {phi(x1D0,x1D1,x2D0,x2D1), psi(x1D0,x1D1,x2D0,x2D1),
theta(x1D0,x1D1,x2D0,x2D1), upsilon(x1D0,x1D1,x2D0,x2D1)}
Number of possible solutions: 1
Complexity of chosen solution: 236
```

$$\left\{\phi(x1D0, x1D1, x2D0, x2D1) = -\frac{1}{\sin(x2D0)}, \psi(x1D0, x1D1, x2D0, x2D1) = 0, \theta(x1D0, x1D1, x2D0, x2D1) = -\sin(x1D1)\right.$$
$$\left. - \cos(x2D1), \upsilon(x1D0, x1D1, x2D0, x2D1) = -\frac{x1D0}{\sin(x2D0)} + \_F1(x1D1, x2D0, x2D1)\right\}$$

I.e. the found solution is

$$\begin{aligned} \phi(x, \dot{x}) &= -\frac{1}{\sin(x_2)} \\ \psi(x, \dot{x}) &= 0 \\ \theta(x, \dot{x}) &= -\sin(\dot{x}_1) - \cos(\dot{x}_2) \\ \upsilon(x, \dot{x}) &= -\frac{x_1}{\sin(x_2)} + \_F1(x_1, x_2, \dot{x}_2) \end{aligned} \tag{7.83}$$

Remark: We may easily substitute the solution into the initial expressions by using the method substitute (7.3.2.4.12) of this toolbox.

7.3.2.4.12 substitute

Remark: This method has been overloaded three times, i.e. it has three different parameter sequences which offer a different feature each[17].

Visibility: export

Parameters (feature 1):

subsEquation::(Not(list) = Not(list))

argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Option: subsDiffDegree::integer:=0

Parameters (feature 2):

subsEquations::Or(set((Not(list)= Not(list))),list((Not(list) = Not(list))))

argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Option: subsDiffDegree::integer:=0

Parameters (feature 3):

subsEquation::(MonoDiffForm = Or(DiffForm, DiffSum))

argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Option: subsDiffDegree::integer:=0

Return type: Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

[17] In fact, the method has been overloaded four times, but the fourth overloaded method only catches illegal parameter combinations.

Description: This method is used for different kinds of substitutions in DiffForms, DiffSums, DiffMatrices, OperForms, OperSums and OperMatrices. The three features of the method are:

Feature 1: Substitute a single expression in coefficients: The method can be used to substitute a single expression (e.g. $\phi(x_1) = \sin(x_1)$) in all occurring coefficients of argumentOne. The first parameter subsEquation is the equation which shall be substituted.

Feature 2: Substitute multiple expressions in coefficients: The method can also be used to handle multiple substitutions simultaneously. In order to do that, we may enter a set or list of substitutions as parameter subsEquations.

Feature 3: Substitute a monomial differential form: The method can be used to substitute all occurrences of a monomial differential form by a DiffForm or DiffSum in the given argumentOne (e.g. $d\omega = dx_1 \wedge dx_2 + d\dot{x}_1 \wedge dx_2$).

In all three cases, the option subsDiffDegree allows us to automatically substitute time derivatives of the substitute, too. I.e. if we substitute $\phi(x_1) = \sin(x_1)$ with subsDiffDegree set to 2, the method will also substitute $\dot{\phi}(x_1) = \dot{x}_1 \cos(x_1)$ and $\ddot{\phi}(x_1) = -\dot{x}_1^2 \sin(x_1) + \ddot{x}_1 \cos(x_1)$. The same applies to MonoDiffForms.

Examples:
For feature 1: Substitute a single expression in coefficients: Let us substitute the expression $\upsilon(x_1, x_2) = \sin(x_1) + \cos(x_2)$ in the vector

$$matrix_1 = \begin{pmatrix} \upsilon(x_1, x_2) + x_1 dx_2 \\ \frac{\partial}{\partial x_2}\upsilon(x_1, x_2) d\dot{x}_2 \end{pmatrix}. \tag{7.84}$$

This can be done by the commands

Listing 7.45: Substitute a single expression

```
> matrix_1 := Matrix(2, 1, {
  (1,1) = DiffForm(upsilon(x1D0, x2D0) + x1D0, MonoDiffForms(x2D0)),
  (2,1) = DiffForm(diff(upsilon(x1D0, x2D0), x2D0), MonoDiffForms(x2D1))
  });
> result_1 := substitute(upsilon(x1D0, x2D0) = sin(x1D0) + cos(x2D0),
matrix_1):
> result_1;
```

and yields

$$result_1 = \begin{pmatrix} \sin(x_1) + \cos(x_2) + x_1 dx_2 \\ -\sin(x_2) d\dot{x}_2 \end{pmatrix}. \tag{7.85}$$

For feature 2: Substitute multiple expressions in coefficients: Let us assume

that we had to substitute the expressions

$$\begin{aligned} \upsilon(x_1, x_2) &= \sin(x_2) \\ \theta(\dot{x}_1) &= 0 \\ \mu(\dot{x}_2) &= \frac{1}{\dot{x}_2} \end{aligned} \tag{7.86}$$

in the vector

$$matrix_1 = \begin{pmatrix} \upsilon(x_1, x_2) + \theta(\dot{x}_1) + \mu(\dot{x}_2)dx_2 \\ \upsilon(x_1, x_2) + \frac{\partial}{\partial \dot{x}_1}\theta(\dot{x}_1)dx_2 \\ \frac{\partial}{\partial x_2}\upsilon(x_1, x_2) + \mu(\dot{x}_2)d\dot{x}_2 \end{pmatrix}. \tag{7.87}$$

This can easily be accomplished with the method substitute

Listing 7.46: Substitute multiple expressions

```
> matrix_1 := Matrix(3, 1, {
  (1,1) = DiffForm(upsilon(x1D0, x2D0) + theta(x1D1) + mu(x2D1),
  MonoDiffForms(x2D0)),
  (2,1) = DiffForm(upsilon(x1D0, x2D0) + diff(theta(x1D1), x1D1),
  MonoDiffForms(x2D0)),
  (3,1) = DiffForm(diff(upsilon(x1D0, x2D0), x2D0) + mu(x2D1),
  MonoDiffForms(x2D1))
  });
> substitutions := {upsilon(x1D0, x2D0) = sin(x2D0), theta(x1D1) = 0,
mu(x2D1) = 1/x2D1};
> result_1 := substitute(substitutions, matrix_1):
> result_1;
```

and yields the vector

$$result_1 = \begin{pmatrix} \sin(x_2) + \frac{1}{\dot{x}_2}dx_2 \\ \sin(x_2)dx_2 \\ \cos(x_2) + \frac{1}{\dot{x}_2}d\dot{x}_2 \end{pmatrix}. \tag{7.88}$$

For feature 3: Substitute a monomial differential form: If we want to substitute a formerly unknown monomial differential form by the written-out p-differential form, we may do this like in the following: In this example, we have the operator (in this case an OperSum)

$$\mu = \dot{\omega} \wedge + \omega \wedge \frac{d}{dt} \tag{7.89}$$

with $\omega \in \Lambda^1(\mathfrak{X})$. Let us assume that we discovered that $\omega = \sin(x_1)dx_1 + \cos(x_2)dx_2$. In order to substitute ω and $\dot{\omega}$ correctly, we may use

Listing 7.47: Substitute a monomial differential form

```
> mu := OperSum(OperForm(1, MonoDiffForm(false, omegaD1, 1), 0), OperForm(1,
MonoDiffForm(false, omegaD0, 1), 1));
> subs_1 := DiffSum(DiffForm(sin(x1D0), MonoDiffForm(x1D0)),
DiffForm(cos(x2D0), MonoDiffForm(x2D0)));
> mu := substitute(MonoDiffForm(false, omegaD0, 1) = subs_1, mu,
subsDiffDegree = 1);
```

which yields

$$\begin{aligned}\mu &= \cos(x_1)\dot{x}_1 dx_1 \wedge + \sin(x_1) d\dot{x}_1 \wedge - \sin(x_2)\dot{x}_2 dx_2 \wedge \\ &+ \cos(x_2) d\dot{x}_2 \wedge + \sin(x_1) dx_1 \wedge \frac{d}{dt} + \cos(x_2) dx_2 \wedge \frac{d}{dt}. \end{aligned} \quad (7.90)$$

7.3.2.4.13 subtract

Visibility: export

Parameters:

summandOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

summandTwo::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Return type: Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Description: This method subtracts summandTwo from summandOne. The result will automatically be simplified. We can only subtract differential forms (DiffForms and/or DiffSums) from other differential forms and operators (OperForm and/or OperSum) from other operators. In case of matrices we can only subtract DiffMatrices from DiffMatrices and OperMatrices from OperMatrices.
The second parameter summandTwo is optional. If it is missing, the method will compute the negative of summandOne.

Examples: In this example, we compute $\omega - \theta$ with

$$\begin{aligned}\omega &= \frac{\cos(x_2)}{\sqrt{1-\dot{x}_3^2}} d\dot{x}_1 + x_1 dx_2 \\ \theta &= \sin(x_1) dx_1 + d\dot{x}_1 \end{aligned} \quad (7.91)$$

with the commands

Listing 7.48: Subtract a DiffSum from a DiffSum

```
> omega := DiffSum(DiffForm(cos(x2D0)/sqrt(1-(x3D1)^2), MonoDiffForm(x1D1)),
DiffForm(x1D0, MonoDiffForm(x2D0)));
> theta := DiffSum(DiffForm(sin(x1D0), MonoDiffForm(x1D0)), DiffForm(1,
MonoDiffForm(x1D1)));
> result_1 := subtract(omega, theta);
```

to get the DiffSum

$$result_1 = \left(\frac{\cos(x_2)}{\sqrt{1-\dot{x}_3^2}} - 1 \right) d\dot{x}_1 + x_1 dx_2 - \sin(x_1) dx_1. \quad (7.92)$$

7.3.2.4.14 wedge

Visibility: export

Parameters:

argumentOne::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

argumentTwo::Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Return type: Or(DiffForm, DiffSum, DiffMatrix, OperForm, OperSum, OperMatrix)

Description: This method computes the wedge product of the two given arguments as defined in section 3.3.3. Note that not every combination of differential forms and operators is defined. Therefore, argumentOne and argumentTwo have to be suitable. In case of differential forms and operators the allowed combinations are[18]:

argumentOne \ argumentTwo	DiffForm	DiffSum	OperForm	OperSum
DiffForm	X	X	-	-
DiffSum	X	X	-	-
OperForm	X	X	X	X
OperSum	X	X	X	X

In case of matrices of differential forms and operators we may apply the wedge product according to:

argumentOne \ argumentTwo	DiffMatrix	OperMatrix
DiffMatrix	X	-
OperMatrix	X	X

Examples: In the first example, we want to compute

$$\omega \wedge \theta,\ \omega \in \Lambda^1(\mathfrak{X}), \theta \in \Lambda^1(\mathfrak{X}) \tag{7.93}$$

with

$$\omega = dx_1 + dx_2 \tag{7.94}$$
$$\theta = x_1 dx_1 + x_2 d\dot{x}_2 \tag{7.95}$$

by using the commands

Listing 7.49: Wedge a DiffSum to a DiffSum

```
> omega := DiffSum(DiffForm(1, MonoDiffForm(x1D0)), DiffForm(1,
MonoDiffForm(x2D0)));
```

[18] X describes a possible combination.

```
> theta := DiffSum(DiffForm(x1D0, MonoDiffForm(x1D0)), DiffForm(x2D0,
MonoDiffForm(x2D1)));
> result_1 := wedge (omega, theta);
```

This returns the DiffSum

$$result_1 = x_2 dx_1 \wedge d\dot{x}_2 + x_1 dx_2 \wedge dx_1 + x_2 dx_2 \wedge d\dot{x}_2. \tag{7.96}$$

In the second example, we want to compute

$$\mu \wedge \omega,\ \mu \in \mathcal{L}\left(\Lambda^1(\mathfrak{X}), \Lambda^2(\mathfrak{X})\right), \omega \in \Lambda^1(\mathfrak{X}) \tag{7.97}$$

with

$$\mu = \upsilon(x_1, x_2) dx_1 \wedge \frac{d}{dt} \tag{7.98}$$

$$\omega = \sin(x_1) dx_1 + \tan(x_2) dx_2 \tag{7.99}$$

by using the commands

Listing 7.50: Wedge an OperForm to a DiffSum

```
> mu := OperForm(upsilon(x1D0, x2D0), MonoDiffForm(x1D0), 1);
> omega := DiffSum(DiffForm(sin(x1D0), MonoDiffForm(x1D0)),
DiffForm(tan(x2D0), MonoDiffForm(x2D0)));
> result_1 := wedge(mu, omega);
```

This returns the DiffSum

$$\begin{aligned} result_1 = {} & \upsilon(x_1, x_2) \sin(x_1) dx_1 \wedge d\dot{x}_1 + \frac{\upsilon(x_1, x_2)\dot{x}_2}{\cos(x_2)^2} dx_1 \wedge dx_2 \\ & + \upsilon(x_1, x_2) \tan(x_2) dx_1 \wedge d\dot{x}_2. \end{aligned} \tag{7.100}$$

7.3.3 MinimalbasisDecomp

This module contains the methods needed for the minimal basis decomposition of OperMatrices. There is only one exported method: decompose (7.3.3.4.1) which offers the minimal basis decomposition of OperMatrices. All other methods are local methods which are used for the minimal basis decomposition.
Since we use a specialized data structure[19] in this submodule (in order to increase the computational performance), all OperMatrices entered will automatically be transformed into the internal data type SkewMatrix. The result will also be automatically transformed back.
Many methods of this submodule work similar to the methods implemented in the Decompose submodule of the DifferentialDelays-toolbox. Therefore, we will refer to those methods when appropriate and only emphasize the differences.

[19] For more information see section 5.4.

7.3.3.1 Initialization

7.3.3.1.1 initializeSubPackage

Visibility: local

Parameters: -

Return type: –

Description: This method initializes the three internal data types SkewPolynomial, SkewVector and SkewMatrix. This method will automatically be called when the DifferentialForms-toolbox is loaded by the with-command of *Maple*.

7.3.3.2 Constructors

7.3.3.2.1 SkewPolynomial

Visibility: export

Parameters:

coefficients :: seq(Not(list))

Return type: SkewPolynomial

Description: This is the only constructor method in this submodule. It is used by other methods of this submodule in order to create SkewPolynomials[20]. A SkewPolynomial is the technical representation of a polynomial $\in \mathcal{K}\left[\frac{d}{dt}\right]$, i.e. it has the form

$$p(t) = \sum_{i=0}^{N} c_i(t)\frac{d^i}{dt^i},\ c_i(t) \in \mathcal{K},\ \deg(p) = N \in \mathbb{N}_0 \tag{7.101}$$

The parameter coefficients is a comma separated list of the coefficients of the desired polynomial, starting with the coefficient to the degree 0 in $\frac{d}{dt}$. The parameter coefficients is optional. If it is missing, the method will create a polynomial $p(t) = 0$.

Examples: In this example, we will create three different SkewPolynomials:

$$\begin{aligned} poly_1 &= \frac{d}{dt} \\ poly_2 &= \beta \\ poly_3 &= x_2 + (1+\sin(x_3))\frac{d^2}{dt^2} + \frac{sin(\dot{x}_1)}{\cos(\ddot{x}_1)}\frac{d^4}{dt^4} \end{aligned} \tag{7.102}$$

Listing 7.51: Create SkewPolynomials

```
> poly_1 := MinimalbasisDecomp[SkewPolynomial](0, 1);
> poly_2 := MinimalbasisDecomp[SkewPolynomial](beta);
> poly_3 := MinimalbasisDecomp[SkewPolynomial](x2D0, 0, 1+sin(x3D0), 0,
sin(x1D1)/cos(x1D2));
```

[20] It is not intended to use this method outside the toolbox. The visibility of this method is set to export due to technical reasons.

Since all coefficients are checked by the spellingChecker (7.3.2.3.8) of the main module, calls with invalid coefficients will cause an error:

```
> poly_2 := MinimalbasisDecomp[SkewPolynomial](xDD0);
Error, (in MinimalbasisDecomp:-SkewPolynomial) The expression xDD0 is neither a function nor a
constant. Please spell your functions compatible to [a-zA-CE-Z0-9_]+D[0-9]+ and your constants
compatible to [a-zA-CE-Z0-9_]+
```

7.3.3.3 Local methods

7.3.3.3.1 convertToDiffForms

Visibility: local

Parameters:
objectToConvert::Or(SkewMatrix, SkewPolynomial)

Return type: Or(OperForm, OperSum, OperMatrix)

Description: This methods converts the given SkewMatrix or SkewPolynomial into an OperForm, OperSum or OperMatrix, depending on the structure of the objectToConvert. This method is used to convert the result of the minimal basis decomposition back into the usual data structure of the toolbox DifferentialForms.

Examples: In this example, we will convert the SkewPolynomial $\in \mathcal{K}\left[\frac{d}{dt}\right]$

$$poly_1 = x_2 + (1 + \sin(x_3))\,\frac{d^2}{dt^2} \tag{7.103}$$

into an OperSum $\in \mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^p\left(\mathfrak{X}\right)\right)$

$$oper_1 = x_2 \wedge + (1 + \sin(x_3)) \wedge \frac{d^2}{dt^2} \tag{7.104}$$

by using

Listing 7.52: Convert a SkewPolynomial

```
> poly_1 := MinimalbasisDecomp[SkewPolynomial](x2D0, 0, 1+sin(x3D0));
> oper_1 := MinimalbasisDecomp[convertToDiffForms](poly_1);
```

7.3.3.3.2 convertToMinBasis

Visibility: local

Parameters:
objectToConvert::Or(OperForm, OperSum, OperMatrix)

Return type: Or(SkewMatrix, SkewPolynomial)

Description: This methods converts the given OperForm, OperSum or OperMatrix into a SkewPolynomial (in case of OperForms and OperSums) or SkewMatrix (in case of OperMatrices). This submodule uses this method to transform

the incoming data types into the internal data structure in order to increase the computational performance. The result will automatically be simplified. There are restrictions for the structure of the given operators. All operators in objectToConvert must not contain MonoDiffForms. If they do, an error will be raised since they cannot be transformed into elements $\in \mathcal{K}\left[\frac{d}{dt}\right]$.

Examples: In this example, we will convert the OperSum $\in \mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^p\left(\mathfrak{X}\right)\right)$

$$oper_1 = x_2 \wedge + (1 + \sin(x_3)) \wedge \frac{d^2}{dt^2} \tag{7.105}$$

into a SkewPolynomial $\in \mathcal{K}\left[\frac{d}{dt}\right]$

$$poly_1 = x_2 + (1 + \sin(x_3)) \frac{d^2}{dt^2} \tag{7.106}$$

by using

Listing 7.53: Convert an OperSum

```
> oper_1 := OperSum(OperForm(x2D0), OperForm(1 + sin(x3D0), 2));
> poly_1 := MinimalbasisDecomp[convertToMinBasis](oper_1);
```

7.3.3.3.3 degreeOfVector

Visibility: local

Parameters:

selectedVector :: SkewVector

Return type: extended_numeric

Description: This method is similar to method degreeOfVector (7.2.3.1.1) but works on SkewVectors instead of OreVectors.

7.3.3.3.4 degreeVectorOfMatrix

Visibility: local

Parameters:

testedMatrix::SkewMatrix
Option: rowwise::boolean:=false
Option: columnwise::boolean:=false

Return type: Vector(extended_numeric)

Description: This method is similar to method degreeVectorOfMatrix (7.2.3.1.2) but works on SkewMatrices instead of OreMatrices.

7.3.3.3.5 findModifiedAlpha

Visibility: local

Parameters:
testedMatrix::**SkewMatrix**
Option: rowwise::**boolean**:=**false**
Option: columnwise::**boolean**:=**false**
Option: chooseFirstAlpha::**boolean**:=**false**
Option: chooseLowMemoryAlpha::**boolean**:=**false**
Option: chooseLowDegreeAlpha::**boolean**:=**false**
Option: chooseAlphaByUser::**boolean**:=**false**
Option: debugMode::**boolean**:=**false**

Return type: **SkewVector**

Description: This method is similar to method findModifiedAlpha (7.2.3.1.3) but works on SkewMatrices instead of OreMatrices.

7.3.3.3.6 indexOfFirstEntryWithDegreeZero

Visibility: local

Parameters:
skewPolynomialVector::**SkewVector**

Return type: **integer**

Description: This method is similar to method indexOfFirstEntryWithDegreeZero (7.2.3.1.4) but works on SkewVectors instead of OreVectors.

7.3.3.3.7 leadingCoeffMatrix

Visibility: local

Parameters:
testedMatrix::**SkewMatrix**
Option: rowwise::**boolean**:=**false**
Option: columnwise::**boolean**:=**false**

Return type: **Matrix**(**Not**(**list**))

Description: This method is similar to method leadingCoeffMatrix (7.2.3.1.5) but works on SkewMatrices instead of OreMatrices.

7.3.3.3.8 mapletForCustomAlpha

Visibility: local

Parameters:
alphas:: **list** (**Vector**(**Not**(**list**)))
degreeVector::**Vector**(**extended_numeric**)

Return type: **integer**

Description: This method is similar to method mapletForCustomAlpha (7.2.3.1.6) but expects usual *Maple* functions as entries of the vectors alphas instead of LeftFractions.

7.3.3.3.9 skewDerivative

Visibility: local

Parameters:
argumentOne::SkewPolynomial

Return type: SkewPolynomial

Description: This method differentiates the given SkewPolynomial argumentOne with respect to t. I.e. in case of an arbitrary skew polynomial $p(t) \in \mathcal{K}\left[\frac{d}{dt}\right]$, $c_i(t) \in \mathcal{K}$, $\deg(p) = N \in \mathbb{N}_0$ it returns

$$\begin{aligned} \frac{d}{dt}\left(p(t)\right) &= \frac{d}{dt}\left(\sum_{i=0}^{N} c_i(t)\frac{d^i}{dt^i}\right) \\ &= \frac{d}{dt}\left(c_0(t)\right) + \sum_{i=1}^{N} c_{i-1}(t) + \frac{d}{dt}\left(c_i(t)\right)\frac{d^i}{dt^i} \\ &\quad + c_N(t)\frac{d^{N+1}}{dt^{N+1}} \end{aligned} \tag{7.107}$$

The result will automatically be simplified by the method skewSimplifier (7.3.3.3.13).

7.3.3.3.10 skewIdentityMatrix

Visibility: local

Parameters:
dimension::integer

Return type: SkewMatrix

Description: This method returns a SkewMatrix in form of an identity matrix with the given dimension dimension.

Examples: For instance, we may easily create an identity matrix $\in \mathcal{K}\left[\frac{d}{dt}\right]^{3\times 3}$ with the command

Listing 7.54: Create a skew identity matrix

```
> matrix_1 := MinimalbasisDecomp[skewIdentityMatrix](3);
```

which will create the matrix

$$matrix_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \tag{7.108}$$

7.3.3.3.11 skewMultiply

Visibility: local

Parameters:
argumentOne::Or(SkewPolynomial, SkewMatrix)
argumentTwo::Or(SkewPolynomial, SkewMatrix)

Return type: Or(SkewPolynomial, SkewMatrix)

Description: This method computes the product of two SkewPolynomials or two SkewMatrices. Note that the product of two SkewPolynomials $p(t), q(t) \in \mathcal{K}\left[\frac{d}{dt}\right]$, $\deg(p) = N \in \mathbb{N}_0$, $\deg(q) = M \in \mathbb{N}_0$ with the coefficients $p_i(t), q_j(t) \in \mathcal{K}$ is given by

$$\begin{aligned} p(t) \cdot q(t) &= \sum_{i=0}^{N} p_i(t) \frac{d^i}{dt^i} \cdot \sum_{j=0}^{M} p_j(t) \frac{d^j}{dt^j} \\ &= \sum_{i=0}^{N} p_i(t) \cdot \underbrace{\frac{d^i}{dt^i} \left(\sum_{j=0}^{M} p_j(t) \frac{d^j}{dt^j} \right)}_{\text{as defined in (7.107)}} . \end{aligned} \tag{7.109}$$

Examples: In this example, we want to compute $p(t) \cdot q(t)$ with

$$\begin{aligned} p(t) &= x_1 + \frac{d}{dt} \\ q(t) &= \dot{x}_2 + \frac{d}{dt} + \sin(\dot{x}_3) \frac{d^2}{dt^2}. \end{aligned} \tag{7.110}$$

This can be done by the commands

Listing 7.55: Multiply two SkewPolynomials

```
> p := MinimalbasisDecomp[SkewPolynomial](x1D0, 1);
> q := MinimalbasisDecomp[SkewPolynomial](x2D1, 1, sin(x3D1));
> result_1 := MinimalbasisDecomp[skewMultiply](p, q);
```

and yields the result

$$\begin{aligned} result_1 &= x_1 \dot{x}_2 + \ddot{x}_2 + (x_1 + \dot{x}_2) \frac{d}{dt} \\ &\quad + (x_1 \sin(\dot{x}_3) + 1 + \cos(\dot{x}_3) \ddot{x}_3) \frac{d^2}{dt^2} + \sin(\dot{x}_3) \frac{d^3}{dt^3}. \end{aligned} \tag{7.111}$$

7.3.3.3.12 skewPlus

Visibility: local

Parameters:
argumentOne::Or(SkewPolynomial, SkewMatrix)
argumentTwo::Or(SkewPolynomial, SkewMatrix)

Return type: Or(SkewPolynomial, SkewMatrix)

Description: This method computes the sum of two SkewPolynomials or two SkewMatrices.

7.3.3.3.13 skewSimplifier

Visibility: local

Parameters:

theArgument::Or(SkewPolynomial, SkewMatrix)
Option: simplifyCoefficients :: boolean:=true
Option: eraseZeros :: boolean:=true

Return type: Or(SkewPolynomial, SkewMatrix)

Description: This method is the equivalent to the *Maple*-command simplify. It simplifies the given parameter theArgument (which can be of type SkewPolynomial or SkewMatrix) according to certain rules.
The features of this method are:

1.) simplifying the coefficients (if enabled)

2.) erasing dispensable leading zeros (if enabled)

1.) This feature is triggered by the option simplifyCoefficients. If this option is set to true, the method will simplify all coefficients with the *Maple*-command simplify. This is important since it is possible to have mathematical terms as coefficients which are actually zero but without having *Maple* recognizing that. On the other hand, the *Maple*-command simplify needs a lot of computation time. Therefore, it can make sense to omit the simplification of the coefficients during computations as long as we are aware of the fact that some of the coefficients might be zero.

2.) This feature erases all leading zeros from the SkewPolynomial, i.e. all coefficients c_i of the SkewPolynomial $p(t) \in \mathcal{K}[\delta]$ which satisfy $i > \deg(p)$. This is necessary due to the fact that the data type is implemented as list with the size equal to the degree of the polynomial plus 1.

7.3.3.3.14 skewSubtract

Visibility: local

Parameters:

argumentOne::Or(SkewPolynomial, SkewMatrix)
argumentTwo::Or(SkewPolynomial, SkewMatrix)

Return type: Or(SkewPolynomial, SkewMatrix)

Description: This method subtracts argumentTwo from argumentOne. The result will automatically be simplified. We can only subtract SkewPolynomials

from other SkewPolynomials and SkewMatrices from other SkewMatrices.
The second parameter argumentTwo is optional. If only the parameter argumentOne is given, the method will compute the negative of argumentOne.

Examples: In this example, we want to compute $p(t) - q(t)$ with

$$\begin{aligned} p(t) &= 1 + x_1 \frac{d}{dt} + \sin(x_3)\frac{d^3}{dt^3} \\ q(t) &= \dot{x}_2 + \frac{d}{dt} + \tan(\dot{x}_3)\frac{d^2}{dt^2} \end{aligned} \tag{7.112}$$

This can easily be done by using

Listing 7.56: Subtract two SkewPolynomials

```
> p := MinimalbasisDecomp[SkewPolynomial](1, x1D0, 0, sin(x3D0));
> q := MinimalbasisDecomp[SkewPolynomial](x2D1, 1, tan(x3D1));
> difference_1 := MinimalbasisDecomp[skewSubtract](p, q);
```

This yields the result

$$difference_1 = 1 - \dot{x}_2 + (x_1 - 1)\frac{d}{dt} - \tan(\dot{x}_3)\frac{d^2}{dt^2} + \sin(x_3)\frac{d^3}{dt^3}. \tag{7.113}$$

7.3.3.3.15 switchColumns

Visibility: local

Parameters:
matrixToChange::**SkewMatrix**
indexOne::**integer**
indexTwo::**integer**

Return type: SkewMatrix

Description: This method is similar to method switchColumns (7.2.4.1.4) but works on SkewMatrices instead of OreMatrices.

7.3.3.3.16 switchRows

Visibility: local

Parameters:
matrixToChange::**SkewMatrix**
indexOne::**integer**
indexTwo::**integer**

Return type: SkewMatrix

Description: This method is similar to method switchRows (7.2.4.1.5) but works on SkewMatrices instead of OreMatrices.

7.3.3.4 Exported methods

7.3.3.4.1 decompose

Visibility: export

Parameters:

selectedMatrix :: **OperMatrix**
Option: rowwise::**boolean**:=**false**
Option: columnwise::**boolean**:=**false**
Option: chooseFirstAlpha::**boolean**:=**false**
Option: chooseLowMemoryAlpha::**boolean**:=**false**
Option: chooseLowDegreeAlpha::**boolean**:=**false**
Option: chooseAlphaByUser::**boolean**:=**false**
Option: returnInverseOperator::**boolean**:=**false**
Option: debugMode::**boolean**:=**false**

Return type: **SkewMatrix**, **SkewMatrix**

Description: This method is similar to method decompose (7.2.3.2.1) but works on OperMatrices instead of OreMatrices. The given OperMatrix selectedMatrix will automatically be transformed into a SkewMatrix and the resulting matrices will automatically be transformed back into OperMatrices once the decomposition is finished.

Chapter 8

Usage of the Toolboxes on the Basis of a Few Examples

In this chapter, the usage of the toolboxes will be illustrated by computing the flat output resp. the defining operators for a few systems. The corresponding worksheets and packages can be found in the appendix.

8.1 Linear Time-Varying System Without Delays

In the first example, we want to compute the defining matrices of a linear time-varying system from [24] in order to show whether the system is flat or not. This example can be found in the worksheet Compute flat output (linear time-varying system) - Example.mw. The system matrices for the system representation (3.32) are

$$A = \begin{pmatrix} \frac{d}{dt} & -1 & 0 \\ 2 \cdot sin(\mu t) & -6 + \frac{d}{dt} & 0 \\ 0 & 0 & -2 \cdot cos(\mu t) + \frac{d}{dt} \end{pmatrix}, \; B = \begin{pmatrix} 0 & 0 \\ 1 & \frac{1}{2} \\ 0 & 1 \end{pmatrix} \tag{8.1}$$

To describe the system in *Maple*, we can use the constructor OrePolynomial of the toolbox DifferentialDelays. The other constructors will not be used in this example since we do not have any delays in the system.

Listing 8.1: Initializing the system matrices

```
> A := Matrix(3, 3, {
  (1,1) = OrePolynomial(0, 1),
  (1,2) = OrePolynomial(-1),
  (1,3) = OrePolynomial(0),
  (2,1) = OrePolynomial(2 * sin(mu * tD0T0)),
  (2,2) = OrePolynomial(-6, 1),
  (2,3) = OrePolynomial(0),
  (3,1) = OrePolynomial(0),
  (3,2) = OrePolynomial(0),
  (3,3) = OrePolynomial(-2 * cos(mu * tD0T0), 1)
  });
```

```
> B := Matrix(3, 2, {
  (1,1) = OrePolynomial(0),
  (1,2) = OrePolynomial(0),
  (2,1) = OrePolynomial(1),
  (2,2) = OrePolynomial(1/2),
  (3,1) = OrePolynomial(0),
  (3,2) = OrePolynomial(1)
  });
```

Note that (as told in section 6.1) **tD0T0** is used to represent t. The toolbox will automatically substitute all differentiations of t.
Now we have to check whether the preconditions

i) the system matrix B is hyper-regular

ii) the rows of (A, B) are linearly independent

are fulfilled. To ensure i), we use the row-wise minimal basis decomposition of B from (3.34):

Listing 8.2: Minimal basis decomposition of B

```
> M_tilde, remainderOfB := Decompose[decompose](B, rowwise = true,
chooseFirstAlpha = true);
```

By analyzing the structure of remainderOfB, we can evaluate whether B is hyper-regular, referring to Definition 5. In this case the result is

$$M\_tilde, remainderOfB := \left[\begin{array}{lll} [[[1],[0]]] & [[[1],[1]]] & \left[\left[[1],\left[-\frac{1}{2}\right]\right]\right] \\ [[[1],[0]]] & [[[1],[0]]] & [[[1],[1]]] \\ [[[1],[1]]] & [[[1],[0]]] & [[[1],[0]]] \end{array}\right], \left[\begin{array}{ll} [[[1],[1]]] & [[[1],[0]]] \\ [[[1],[0]]] & [[[1],[1]]] \\ [[[1],[0]]] & [[[1],[0]]] \end{array}\right]$$

To convert the result into a more readable form, we can use the two methods orePrinting and oreLatexPrinting since we have matrices of type OreMatrix. This yields

$$\begin{aligned} \widetilde{M} &= \begin{pmatrix} 0 & 1 & -1/2 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \\ \text{remainderOfB} &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}. \end{aligned} \tag{8.2}$$

So we can easily see that B is hyper-regular because of the structure of remainderOfB.
To ensure precondition ii), we simply determine the form of (A, B):

Listing 8.3: Construct A_B

```
> A_B := Matrix(3, 5):
  A_B[..,1..3] := copy(A):
  A_B[..,4..5] := copy(B):
```

The resulting structure of the OreMatrix A_B shows that the rows are linear independent since there is no possible way to create a row filled with zeros by using a linear combination of the rows of A_B:

$$A\_B = \begin{pmatrix} \frac{d}{dt} & -1 & 0 & 0 & 0 \\ 2 \cdot sin(\mu t) & -6 + \frac{d}{dt} & 0 & 1 & 1/2 \\ 0 & 0 & -2 \cdot cos(\mu t) + \frac{d}{dt} & 0 & 1 \end{pmatrix} \tag{8.3}$$

Since we have computed $\widetilde{M}$ in (8.2), we may also compute $F \in \mathcal{K}\left[\frac{d}{dt}\right]^{1\times 3}$ using (3.37):

Listing 8.4: Compute F

```
> F := oreMultiply(M_tilde, A)[3..3,..];
```

This yields the result

$$F = \begin{pmatrix} \frac{d}{dt} & -1 & 0 \end{pmatrix}. \tag{8.4}$$

According to Theorem 9, the linear system is flat if and only if F is hyper-regular, so we will use the minimal basis decomposition to evaluate the hyper-regularity of F:

Listing 8.5: Minimal basis decomposition of F

```
> remainderOfF, Q_tilde, Q_tilde_inverse := Decompose[decompose](F,
columnwise = true, chooseFirstAlpha = true, returnInverseOperator = true);
```

Since the method decompose allows us to compute and return additionally the inverse operator matrix, we do this in order to omit the second minimal basis decomposition (3.101). This yields the three matrices

$$\begin{aligned} \text{remainderOfF} &= \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \\ \widetilde{Q} &= \begin{pmatrix} 0 & 1 & 0 \\ -1 & \frac{d}{dt} & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ \widetilde{Q}^{-1} &= \begin{pmatrix} \frac{d}{dt} & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}. \end{aligned} \tag{8.5}$$

Because of the structure of remainderOfF, we can be sure that F is hyper-regular and therefore the linear system is flat referring to Theorem 9. Finally, we want to compute the defining operators P, Q and R from Theorem 10.

Listing 8.6: Compute the defining operators

```
> P := Q_tilde_inverse[2..3,..];
> Q := Q_tilde[..,2..3];
> R := oreMultiply(M_tilde, oreMultiply(A, Q))[1..2,..];
```

This leads to the matrices

$$P := \begin{bmatrix} [[1],[1]] & [[1],[0]] & [[1],[0]] \\ [[1],[0]] & [[1],[0]] & [[1],[1]] \end{bmatrix}$$

$$Q := \begin{bmatrix} [[1],[1]] & [[1],[0]] \\ [[1],[0]],[[1],[1]] & [[1],[0]] \\ [[1],[0]] & [[1],[1]] \end{bmatrix}$$

$$R := \begin{bmatrix} [[1],[2\sin(\mu\, tD0T0)]],[[1],[-6]],[[1],[1]] & [[1],[\cos(\mu\, tD0T0)]],\left[[1],\left[-\frac{1}{2}\right]\right] \\ [[1],[0]] & [[1],[-2\cos(\mu\, tD0T0)]],[[1],[1]] \end{bmatrix}$$

$$\begin{aligned} P &= \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ Q &= \begin{pmatrix} 1 & 0 \\ \frac{d}{dt} & 0 \\ 0 & 1 \end{pmatrix} \\ R &= \begin{pmatrix} 2\cdot sin(\mu t) - 6\cdot\frac{d}{dt} + \frac{d^2}{dt^2} & cos(\mu t) - 1/2\cdot\frac{d}{dt} \\ 0 & -2\cdot cos(\mu t) + \frac{d}{dt} \end{pmatrix}. \end{aligned} \tag{8.6}$$

8.2 Linear Time-Varying System With Delays

In the next example, we want to compute the defining operators and the operator π of a linear time-varying system with delays from [4]. This example can be found in the worksheet Compute flat output (linear system with delays) - Example 1.mw.
The system matrices for the system representation (3.32) are

$$A = \begin{pmatrix} \frac{d}{dt} & -s(t)\delta + s(t)\delta^2 \\ 0 & \frac{d}{dt} \end{pmatrix},\ B = \begin{pmatrix} 0 \\ \delta \end{pmatrix} \tag{8.7}$$

In contrast to the example in section 8.1, we additionally have to use the constructor DelayPolynomial since we encounter delays in this example. Since all denominators of occurring left fractions in this example are of degree 0 in δ, we do not have to initialize them as LeftFractions explicitly.

Listing 8.7: Initializing the system matrices

```
> A := Matrix(2, 2, {
  (1,1) = OrePolynomial(0, 1),
  (1,2) = OrePolynomial(DelayPolynomial(0, -sD0T0, sD0T0)),
  (2,1) = OrePolynomial(0),
  (2,2) = OrePolynomial(0, 1)
  });
> B := Matrix(2, 1, {
  (1,1) = OrePolynomial(0),
  (2,1) = OrePolynomial(DelayPolynomial(0, 1))
  });
```

At first, we will check for the preconditions:

i) the system matrix B is $\mathcal{K}(\delta)\left[\frac{d}{dt}\right]$-hyper-regular

ii) the rows of (A, B) are $\mathcal{K}(\delta)\left[\frac{d}{dt}\right]$-linearly independent

Again, to ensure i), we use the row-wise minimal basis decomposition of B from (3.34):

Listing 8.8: Minimal basis decomposition of B

```
> M_tilde, remainderOfB := Decompose[decompose](B, rowwise = true,
chooseFirstAlpha = true);
```

This yields the result

$$M\_tilde, remainderOfB := \left[\begin{matrix} [[[1],[0]]] & [[[0,1],[1]]] \\ [[[1],[1]]] & [[[1],[0]]] \end{matrix}\right], \left[\begin{matrix} [[[1],[1]]] \\ [[[1],[0]]] \end{matrix}\right]$$

This corresponds to the matrices[1]

$$\begin{aligned} \widetilde{M} &= \begin{pmatrix} 0 & \delta^{-1} \\ 1 & 0 \end{pmatrix} \\ \mathsf{remainderOfB} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{aligned} \tag{8.8}$$

Obviously, remainderOfB satisfies Definition 5 and therefore B is hyper-regular. To ensure precondition ii), we simply determine the form of (A, B):

Listing 8.9: Construct A_B

```
> A_B := Matrix(2, 3):
  A_B[..,1..2] := copy(A):
  A_B[..,3] := copy(B):
```

The resulting structure of the OreMatrix A_B shows that the rows are linear independent since there is no possible way to create a row filled with zeros by using a linear combination of the rows of A_B:

$$A\_B = \begin{pmatrix} \frac{d}{dt} & -s(t)\delta + s(t)\delta^2 & 0 \\ 0 & \frac{d}{dt} & \delta \end{pmatrix} \tag{8.9}$$

As a next step, we compute the matrix $F \in \mathcal{K}\left[\frac{d}{dt}\right]^{1\times 2}$ using (3.37)

Listing 8.10: Compute F

```
> F := Matrix(oreMultiply(M_tilde, A)[2,..]);
```

This yields

$$F = \begin{pmatrix} \frac{d}{dt} & -s(t)\delta + s(t)\delta^2 \end{pmatrix}. \tag{8.10}$$

Referring to Theorem 9, we will now evaluate whether F is hyper-regular using the minimal basis decomposition:

[1] We may use the method oreLatexPrinting to convert the matrices into *LaTeX* code.

Listing 8.11: Minimal basis decomposition of F

```
> remainderOfF, Q_tilde, Q_tilde_inverse := Decompose[decompose](F,
columnwise = true, chooseFirstAlpha = true, returnInverseOperator = true);
```

Like in example 8.1, we use the option returnInverseOperator to force the method to additionally compute the inverse operator matrix $\widetilde{Q}^{-1}$. This yields the three matrices

$$\begin{aligned} \text{remainderOfF} &= \begin{pmatrix} 1 & 0 \end{pmatrix} \\ \widetilde{Q} &= \begin{pmatrix} 0 & 1 \\ (-s(t)\delta + s(t)\delta^2)^{-1} & (s(t)\delta - s(t)\delta^2)^{-1} \cdot \frac{d}{dt} \end{pmatrix} \\ \widetilde{Q}^{-1} &= \begin{pmatrix} \frac{d}{dt} & -s(t)\delta + s(t)\delta^2 \\ 1 & 0 \end{pmatrix} \end{aligned} \tag{8.11}$$

Because of the structure of *remainderOfF*, we can be sure that F is hyper-regular and therefore the linear system is flat referring to Theorem 9. Moreover, we want to compute the defining operators $\overline{P}$, $\overline{Q}$ and $\overline{R}$ from Definition 16.

Listing 8.12: Compute the defining operators

```
> P_bar := Matrix(Q_tilde_inverse[2,..]);
> Q_bar := Matrix(Q_tilde[..,2]);
> R_bar := Matrix(oreMultiply(M_tilde, oreMultiply(A, Q_bar))[1,..]);
```

This leads to the matrices

$$P\_bar := \left[[[1], [1]]] \; [[[1], [0]]] \right]$$

$$Q\_bar := \begin{bmatrix} [[1], [1]]] \\ [[[1], [0]], [[0, sD0T0, -sD0T0], [1]]] \end{bmatrix}$$

$$R\_bar := \left[\left[[[1], [0]], \left[\left[0, 0, \frac{sD0T0^2}{sD1T0}, -\frac{sD0T0^2}{sD1T0} \right], [-1] \right], [[0, 0, sD0T0, -sD0T0], [1]] \right] \right]$$

$$\begin{aligned} \overline{P} &= \begin{pmatrix} 1 & 0 \end{pmatrix} \\ \overline{Q} &= \begin{pmatrix} 1 \\ (s(t)\delta - s(t)\delta^2)^{-1} \cdot \frac{d}{dt} \end{pmatrix} \\ \overline{R} &= \begin{pmatrix} \left(\frac{s(t)^2}{\dot{s}(t)}\delta^2 - \frac{s(t)^2}{\dot{s}(t)}\delta^3 \right)^{-1} \cdot \frac{d}{dt} + (s(t)\delta^2 - s(t)\delta^3)^{-1} \cdot \frac{d^2}{dt^2} \end{pmatrix} \end{aligned} \tag{8.12}$$

Finally, we have to compute the operator π which eliminates all predictions in the defining operators $\overline{P}$, $\overline{Q}$ and $\overline{R}$:

Listing 8.13: Compute the operator π

```
> pi := PiFlatUtils[computePi](P_bar, Q_bar, R_bar);
```

This yields the DelayPolynomial

$$\pi = \frac{s(t)^2}{\dot{s}(t)}\delta^2 + \frac{-s(t)^2}{\dot{s}(t)}\delta^3 \tag{8.13}$$

We may verify the computed π and simultaneously compute the transformed matrices P, Q and R by using the verify method:

Listing 8.14: Verify π and compute the transformed matrices

```
> returnValue, P := PiFlatUtils[verifyPi](pi, P_bar, showTransformedMatrix =
true);
> returnValue, Q := PiFlatUtils[verifyPi](pi, Q_bar, showTransformedMatrix =
true);
> returnValue, R := PiFlatUtils[verifyPi](pi, R_bar, showTransformedMatrix =
true);
```

In all three cases, returnValue is true indicating that the operator π is valid. Furthermore, we obtain the three matrices:

$$\begin{aligned} P &= \begin{pmatrix} \frac{s(t)^2}{\dot{s}(t)}\delta^2 - \frac{-s(t)^2}{\dot{s}(t)}\delta^3 & 0 \end{pmatrix} \\ Q &= \begin{pmatrix} \frac{s(t)^2}{\dot{s}(t)}\delta^2 - \frac{-s(t)^2}{\dot{s}(t)}\delta^3 \\ \frac{s(t)^2}{\dot{s}(t)s(t-\tau)}\delta \cdot \frac{d}{dt} \end{pmatrix} \\ R &= \begin{pmatrix} -\frac{d}{dt} + \frac{s(t)}{\dot{s}(t)} \cdot \frac{d^2}{dt^2} \end{pmatrix} \end{aligned} \tag{8.14}$$

Remark: A second (and more complex) example from [4] has been computed in the worksheet Compute flat output (linear system with delays) - Example 2.mw. You can find this commented worksheet in the appendix. Since this example has the same computation steps as the example above, we omit an extra section for this example in this chapter.

8.3 Nonlinear System - Non-Holonomic Car

In this example, we will analyze the system of the so called *non-holonomic car* (see [21] for more details). The system matrix of the variational system (3.92) is given by

$$P_F = \begin{pmatrix} \sin(x_3) \wedge \frac{d}{dt} & -\cos(x_3) \wedge \frac{d}{dt} & \dot{x}_1 \cdot \cos(x_3) + \dot{x}_2 \cdot \sin(x_3) \wedge \end{pmatrix}. \tag{8.15}$$

Remark: To shorten the expressions, we use the notation x_i instead of $x_i(t)$ for the states in this example.
To initialize this matrix $P_F \in \mathcal{L}\left(\Lambda^p\left(\mathfrak{X}\right), \Lambda^p\left(\mathfrak{X}\right)\right)$, we simply use the constructor OperForm:

Listing 8.15: Initialize the system matrix

```
> P_F := Matrix(1, 3, {
  (1,1) = OperForm(sin(x3D0), 1),
  (1,2) = OperForm(-cos(x3D0), 1),
  (1,3) = OperForm(x1D1*cos(x3D0)+x2D1*sin(x3D0))
  });
```

Referring to Theorem 12, we have to determine whether P_F is hyper-regular and compute the matrix $\widetilde{Q}$ which transforms P_F into *Smith-Jacobson*-form. We can check this by using the method decompose of the submodule MinimalbasisDecomp:

Listing 8.16: Minimal basis decomposition of P_F

```
> remainderOfP_F, Q_tilde := MinimalbasisDecomp[decompose](P_F, columnwise =
true, chooseAlphaByUser = true);
```

This yields

$$remainderOfP\_F, Q\_tilde := \left[[1, [\,], 0] \; [0, [\,], 0] \; [0, [\,], 0] \right], \left[\left[[0, [\,], 0], [0, [\,], 0], [1, [\,], 0] \right], \left[[0, [\,], 0], [1, [\,], 0], [0, [\,], 0] \right], \left[\left[\frac{1}{x1D1\cos(x3D0) + x2D1\sin(x3D0)}, [\,], 0 \right], \left[\frac{\cos(x3D0)}{x1D1\cos(x3D0) + x2D1\sin(x3D0)}, [\,], 1 \right], \left[-\frac{\sin(x3D0)}{x1D1\cos(x3D0) + x2D1\sin(x3D0)}, [\,], 1 \right] \right] \right]$$

which corresponds to the matrices[2]

$$remainderOfP\_F = \begin{pmatrix} 1\wedge & 0\wedge & 0\wedge \end{pmatrix} \tag{8.16}$$

and

$$\widetilde{Q} = \begin{pmatrix} 0\wedge & 0\wedge & 1\wedge \\ 0\wedge & 1\wedge & 0\wedge \\ \Gamma\wedge & \cos(x_3)\cdot\Gamma\wedge\frac{d}{dt} & -\sin(x_3)\cdot\Gamma\wedge\frac{d}{dt} \end{pmatrix} \tag{8.17}$$

with

$$\Gamma = \frac{1}{\dot{x}_1\cdot cos(x_3) + \dot{x}_2\cdot sin(x_3)}. \tag{8.18}$$

Then, we compute Q like in (3.100) and compute the operator matrix which transforms Q into *Smith-Jacobson*-form:

Listing 8.17: Minimal basis decomposition of Q

```
> Q := Q_tilde[.., 2..3];
> P_tilde, remainderOfQ := MinimalbasisDecomp[decompose](Q, rowwise = true,
chooseFirstAlpha = true);
```

This yields the matrices

$$remainderOfP\_Q = \begin{pmatrix} 1\wedge & 0\wedge \\ 0\wedge & 1\wedge \\ 0\wedge & 0\wedge \end{pmatrix} \tag{8.19}$$

[2] To convert the result into a more readable form we may use the two methods printing and latexPrinting.

and

$$\widetilde{P} = \begin{pmatrix} 0\wedge & 1\wedge & 0\wedge \\ 1\wedge & 0\wedge & 0\wedge \\ \sin(x_3)\cdot\Gamma\wedge\frac{d}{dt} & -\cos(x_3)\wedge\frac{d}{dt} & 1\wedge \end{pmatrix} \tag{8.20}$$

with

$$\Gamma = \frac{1}{\dot{x}_1\cdot cos(x_3)+\dot{x}_2\cdot sin(x_3)}. \tag{8.21}$$

Since P is given by the first two rows of $\widetilde{P}$ (3.102), we may compute P by

Listing 8.18: Compute P

```
> P := P_tilde[1..2,..];
```

which yields the *Maple* output

$$P := \begin{bmatrix} [0,[\,],0] & [1,[\,],0] & [0,[\,],0] \\ [1,[\,],0] & [0,[\,],0] & [0,[\,],0] \end{bmatrix}$$

In order to compute the flat output of the variational system ω from (3.102), we need to construct a vector $dx = \begin{pmatrix} dx_1 & dx_2 & dx_3 \end{pmatrix}^T$. The toolbox DifferentialForms offers a shortcut to instantiate this vector:

Listing 8.19: Initialize dx

```
> dx := createDiffFormVector(x1D0, x2D0, x3D0, extDerivativeDegree = 1);
```

In a second step, we compute $\omega = Pdx$ by using the method wedge:

Listing 8.20: Compute ω

```
> omega := wedge(P, dx):
```

This yields the vector

$$\omega = \begin{pmatrix} dx_2 \\ dx_1 \end{pmatrix}. \tag{8.22}$$

Now we are able to determine whether ω is integrable by computing the exterior derivative (see (3.105)):

Listing 8.21: Exterior derivative of ω

```
> extDerivative(omega);
```

This yields the vector $\begin{pmatrix} 0 & 0 \end{pmatrix}^T$, thus ω is integrable and a flat output of the nonlinear system is given by the integration of ω. We use the method integration in order to compute the flat output y:

Listing 8.22: Integrate ω

```
> y := integration(omega):
```

This finally yields the flat output of the nonlinear system

$$y = \begin{pmatrix} x_2 + \_C1 \\ x_1 + \_C2 \end{pmatrix},\ \_C1, \_C2 \in \mathbb{R} \tag{8.23}$$

8.4 Nonlinear System - Sine Example

In this example, we will analyze the so called *sine example* (see [3] for more details). The system matrix of the variational system (3.92) is given by

$$P_F = \begin{pmatrix} -\frac{\cos(\frac{\dot{x}_1}{\dot{x}_2})}{\dot{x}_2} \wedge \frac{d}{dt} & \frac{\dot{x}_1 \cdot \cos(\frac{\dot{x}_1}{\dot{x}_2})}{\dot{x}_2^2} \wedge \frac{d}{dt} & 1 \wedge \frac{d}{dt} \end{pmatrix}. \tag{8.24}$$

Remark: To shorten the expressions, we use the notation x_i instead of $x_i(t)$ for the states in this example.
In *Maple* we simply write

Listing 8.23: Initialize the system matrix

```
> P_F := Matrix(1, 3, {
  (1,1) = OperForm(-cos(x1D1/x2D1)/x2D1, 1),
  (1,2) = OperForm(x1D1*cos(x1D1/x2D1)/x2D1^2, 1),
  (1,3) = OperForm(1, 1)
  });
```

in order to initialize the system matrix P_F. Referring to Theorem 13, we have to evaluate whether P_F is hyper-regular. This can be done by a minimal basis decomposition of P_F and will give us the operator matrix $\widetilde{Q}$. By using the shortcut mentioned in (3.45) – (3.46), we simultaneously compute $\widetilde{Q}^{-1}$ in order to compute P directly without the need of a second minimal basis decomposition. This can be done by using a special parameter of the method decompose:

Listing 8.24: Minimal basis decomposition of P_F

```
> remainderOfP_F, Q_tilde, Q_tilde_inverse :=
MinimalbasisDecomp[decompose](P_F, columnwise = true, returnInverseOperator =
true, chooseAlphaByUser = true);
```

Since the matrix $remainderOfP\_F$ has the structure

$$remainderOfP\_F = \begin{pmatrix} 1\wedge & 0\wedge & 0\wedge \end{pmatrix}, \tag{8.25}$$

we can be sure that P_F is hyper-regular and therefore P may be computed referring to Theorem 13:

Listing 8.25: Compute P

```
> P := Q_tilde_inverse[2..3,..];
```

This yields the matrix

$$P = \begin{pmatrix} -\frac{\dot{x}_2}{\dot{x}_1}\wedge & 1\wedge & 0\wedge \\ 0\wedge & 0\wedge & 1\wedge \end{pmatrix}. \tag{8.26}$$

In order to compute the flat output of the variational system ω from (3.102), we need to construct a vector $dx = \begin{pmatrix} dx_1 & dx_2 & dx_3 \end{pmatrix}^T$. The toolbox DifferentialForms offers a shortcut to instantiate this vector:

Listing 8.26: Initialize dx

```
> dx := createDiffFormVector(x1D0, x2D0, x3D0, extDerivativeDegree = 1);
```

In a second step, we compute $\omega = Pdx$ by using the method wedge:

Listing 8.27: Compute ω

```
> omega := wedge(P, dx):
```

This yields the *Maple* result

$$\omega := \begin{bmatrix} \left[\left[-\frac{x2D1}{x1D1}, [[true, x1D0, 1]]\right], [1, [[true, x2D0, 1]]]\right] \\ [1, [[true, x3D0, 1]]] \end{bmatrix}$$

This corresponds to the vector

$$\omega = \begin{pmatrix} -\frac{\dot{x}_2}{\dot{x}_1} dx_1 + dx_2 \\ dx_3 \end{pmatrix}. \tag{8.27}$$

Furthermore, we have to check whether (3.105) is satisfied. We may do this by using the method extDerivative:

Listing 8.28: Exterior derivative of ω

```
> extDerivative(omega);
```

Examining the result, we notice that ω is not directly integrable:

$$d(\omega) = \begin{pmatrix} \frac{\dot{x}_2}{\dot{x}_1^2} d\dot{x}_1 \wedge dx_1 - \frac{1}{\dot{x}_1} \dot{x}_2 \wedge dx_1 \\ 0 \end{pmatrix} \neq 0 \tag{8.28}$$

I.e. we have to find an operator $\mu \in \mathcal{L}\left(\Lambda^1(\mathfrak{X}), \Lambda^2(\mathfrak{X})\right)^{m \times m}$ and an unimodular matrix $M \in \mathcal{K}\left[\frac{d}{dt}\right]^{m \times m}$ which satisfy Theorem 14. At first, we construct a $\mu \in \mathcal{L}\left(\Lambda^1(\mathfrak{X}), \Lambda^2(\mathfrak{X})\right)^{m \times m}$ which satisfies

$$d\omega = \mu\omega. \tag{8.29}$$

But before that, we substitute

$$\dot{x}_1 = \arcsin(\dot{x}_3) \cdot \dot{x}_2 \tag{8.30}$$

which is given by the system equation of the nonlinear system into ω using the method substitute of this toolbox:

Listing 8.29: Substitute

```
> omega := substitute(x1D1 = arcsin(x3D1) * x2D1, omega);
```

Since we need to find a μ which satisfies (8.29), we will take a look at $d\omega$ after the substitution (8.30) by using extDerivative

Listing 8.30: Exterior derivative of ω

```
> extDerivative(omega);
```

which yields

$$d\omega = \begin{pmatrix} \frac{1}{\arcsin(\dot{x}_3)^2\sqrt{1-\dot{x}_3^2}} d\dot{x}_3 \wedge dx_1 \\ 0 \end{pmatrix}. \tag{8.31}$$

Therefore, a suitable operator μ may be created with the structure

$$\mu = \begin{pmatrix} 0\wedge & \Psi_0 \wedge + \Psi_1 \wedge \frac{d}{dt} \\ 0\wedge & 0\wedge \end{pmatrix} \tag{8.32}$$

with

$$\begin{aligned} \Psi_i &= \mu_{i,1}(x,\dot{x})dx_1 + \mu_{i,2}(x,\dot{x})dx_2 + \mu_{i,3}(x,\dot{x})dx_3 \\ &\quad + \mu_{i,4}(x,\dot{x})d\dot{x}_1 + \mu_{i,5}(x,\dot{x})d\dot{x}_2 + \mu_{i,6}(x,\dot{x})d\dot{x}_3 \end{aligned} \tag{8.33}$$

with $(x,\dot{x}) = (x_1,\dot{x}_1,x_2,\dot{x}_2,x_3,\dot{x}_3)$. We are able to create this μ using the constructors OperForm, OperSum and MonoDiffForm:

Listing 8.31: Initialize μ

```
> coordinates := x1D0, x1D1, x2D0, x2D1, x3D0, x3D1;
> mu := Matrix(2, 2, {
 (1,1) = OperForm(0),
 (1,2) = OperSum(
 OperForm(mu01(coordinates), MonoDiffForm(x1D0)),
 OperForm(mu02(coordinates), MonoDiffForm(x2D0)),
 OperForm(mu03(coordinates), MonoDiffForm(x3D0)),
 OperForm(mu04(coordinates), MonoDiffForm(x1D1)),
 OperForm(mu05(coordinates), MonoDiffForm(x2D1)),
 OperForm(mu06(coordinates), MonoDiffForm(x3D1)),
 OperForm(mu11(coordinates), MonoDiffForm(x1D0), 1),
 OperForm(mu12(coordinates), MonoDiffForm(x2D0), 1),
 OperForm(mu13(coordinates), MonoDiffForm(x3D0), 1),
 OperForm(mu14(coordinates), MonoDiffForm(x1D1), 1),
 OperForm(mu15(coordinates), MonoDiffForm(x2D1), 1),
 OperForm(mu16(coordinates), MonoDiffForm(x3D1), 1)
 ),
 (2,1) = OperForm(0),
 (2,2) = OperForm(0)
 });
```

Now we are able to compute $\mu\omega$, i.e. the right side of (8.29):

Listing 8.32: Compute $\mu\omega$

```
> mu_omega := wedge(mu, omega);
```

Solving the actual equation can be done by the method solver of this toolbox:

Listing 8.33: Solve the equation

```
> result := solver(d_omega = mu_omega);
```

The method will return

```
Found 10 function(s) in the pde system: {mu01(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1), mu02(x1D0,
x1D1,x2D0,x2D1,x3D0,x3D1), mu04(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1), mu05(x1D0,x1D1,x2D0,x2D1,
x3D0,x3D1), mu06(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1), mu11(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1), mu12
(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1), mu13(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1), mu14(x1D0,x1D1,x2D0,
x2D1,x3D0,x3D1), mu15(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1)}
Number of possible solutions: 1
Complexity of chosen solution: 545
```

$$result := \left\{ \mu01(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = 0,\ \mu02(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = 0,\ \mu04(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = 0,\ \mu05(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = 0,\ \mu06(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = \mu06(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1),\ \mu11(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = -\frac{1}{\arcsin(x3D1)^2\sqrt{1 - x3D1^2}},\ \mu12(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = 0,\ \mu13(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = \mu06(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1),\ \mu14(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = 0,\ \mu15(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = 0 \right\}$$

Hence, we have

$$\begin{aligned}
\mu_{0,1}(x,\dot{x}) &= 0\\
\mu_{0,2}(x,\dot{x}) &= 0\\
\mu_{0,4}(x,\dot{x}) &= 0\\
\mu_{0,5}(x,\dot{x}) &= 0\\
\mu_{1,1}(x,\dot{x}) &= -\frac{1}{\arcsin(\dot{x}_3)^2\sqrt{1-\dot{x}_3^2}}\\
\mu_{1,2}(x,\dot{x}) &= 0\\
\mu_{1,3}(x,\dot{x}) &= \mu_{0,6}(x,\dot{x})\\
\mu_{1,4}(x,\dot{x}) &= 0\\
\mu_{1,5}(x,\dot{x}) &= 0
\end{aligned} \tag{8.34}$$

We can easily substitute this equation system into μ using the substitute-method of DifferentialForms:

Listing 8.34: Substitute the result into μ

```
> mu := substitute(result, mu);
```

According to Theorem 14, the operator μ must also satisfy the assertion

$$\mathfrak{d}(\mu) = \mu\mu. \tag{8.35}$$

Therefore, we compute the two sides of equation (8.35):

Listing 8.35: Compute left and right side of equation

```
> dmu := extDerivative(mu):
> mu_square := wedge(mu, mu);
```

Again, we let the toolbox solve the equation system:

Listing 8.36: Solve the equation

```
> result := solver(dmu = mu_square);
```

We obtain the *Maple*-result:

```
Found 3 function(s) in the pde system: {mu03(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1), mu06(x1D0,x1D1,
x2D0,x2D1,x3D0,x3D1), mu16(x1D0,x1D1,x2D0,x2D1,x3D0,x3D1)}
Number of possible solutions: 1
Complexity of chosen solution: 456
```

$$result := \left\{ \mu03(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = \int\int \frac{\partial^2}{\partial x3D0^2} \_F1(x3D0, x3D1)\, dx3D1\, dx3D1 + \left(\frac{d}{dx3D0} \_F2(x3D0) \right) x3D1 \right.$$

$$+ \_F3(x3D0), \mu06(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = \int \frac{\partial}{\partial x3D0} \_F1(x3D0, x3D1)\, dx3D1 + \_F2(x3D0) + \_C1, \mu16(x1D0,$$

$$\left. x1D1, x2D0, x2D1, x3D0, x3D1) = -\frac{2\, x1D0}{\arcsin(x3D1)^3 \left(-1 + x3D1^2\right)} - \frac{x1D0\, x3D1}{\arcsin(x3D1)^2 \left(1 - x3D1^2\right)^{3/2}} + \_F1(x3D0, x3D1) \right\}$$

Since the used *Maple* version is unable to solve the partial differential equation systems with more complex integrals, we have to substitute those using the fictional functions $G(x_3, \dot{x}_3)$ and $H(x_3)$:

$$\begin{aligned} G(x_3, \dot{x}_3) &= \int\int \_F1(x_3, \dot{x}_3) \mathrm{d}\dot{x}_3 \mathrm{d}\dot{x}_3 \\ H(x_3) &= \int \_F2(x_3) \mathrm{d}x_3 \end{aligned} \tag{8.36}$$

In order to substitute all occurrences of $G(x_3, \dot{x}_3)$ and $H(x_3)$ into the result of equation (8.35), we use default *Maple* functionality:

Listing 8.37: Substitute with G and H

```
> result := subs(Int(Int(diff(_F1(x3D0, x3D1), x3D0, x3D0), x3D1), x3D1) =
D[1,1](G)(x3D0,x3D1), result):
> result := subs(Int(diff(_F1(x3D0, x3D1), x3D0), x3D1) = D[1,2](G)(x3D0,
x3D1), result):
> result := subs(_F1(x3D0, x3D1) = D[2,2](G)(x3D0,x3D1), result):
> result := subs(_F2(x3D0) = D[1](H)(x3D0), result);
```

Afterwards, we may substitute the final result into μ:

Listing 8.38: Substitute the result into μ

```
> mu := substitute(result, mu);
```

This leads to the operator

$$\mu = \begin{pmatrix} 0\wedge & \Psi_0 \wedge + \Psi_1 \wedge \frac{d}{dt} \\ 0\wedge & 0\wedge \end{pmatrix} \tag{8.37}$$

with

$$\begin{aligned} \Psi_0 &= \mu_{0,3}(x, \dot{x}) dx_3 + \mu_{0,6}(x, \dot{x}) d\dot{x}_3 \\ \Psi_1 &= -\frac{1}{\arcsin(\dot{x}_3)^2 \sqrt{1 - \dot{x}_3^2}} dx_1 + \mu_{0,6}(x, \dot{x}) dx_3 + \mu_{1,6}(x, \dot{x}) d\dot{x}_3 \end{aligned} \tag{8.38}$$

Next, referring to Theorem 14, we have to construct an unimodular matrix $M \in \mathcal{K}\left[\frac{d}{dt}\right]^{2\times 2}$ which satisfies

$$\mathfrak{d}(M) = -M\mu. \tag{8.39}$$

Because of the structure of the chosen μ, we use the following approach for an unimodular M:

$$M = \begin{pmatrix} 1\wedge & M_0(x,\dot{x}) \wedge + M_1(x,\dot{x}) \wedge \frac{d}{dt} \\ 0\wedge & 1\wedge \end{pmatrix} \tag{8.40}$$

with $(x, \dot{x}) = (x_1, \dot{x}_1, x_2, \dot{x}_2, x_3, \dot{x}_3)$. To initialize this matrix in *Maple* we use[3]

Listing 8.39: Initialize M

```
> M := Matrix(2,2, {
  (1,1) = OperForm(1),
  (1,2) = OperSum(OperForm(M0(coordinates)), OperForm(M1(coordinates),1)),
  (2,1) = OperForm(0),
  (2,2) = OperForm(1)
  });
```

Then, we compute $\mathfrak{d}(M)$ and $M\mu$:

Listing 8.40: Compute $\mathfrak{d}(M)$ and $M\mu$

```
> d_M := extDerivative(M);
> M_mu := wedge(M,mu);
```

Afterwards, we may solve the equation (8.39) using the already known method solver:

Listing 8.41: Solve the equation

```
> result := solver(d_M = subtract(M_mu));
```

We obtain the result:

$$result := \left[G(x3D0, x3D1) = -\_F4(x3D0, x3D1) + \_F6(x3D0)\, x3D1 + \_F7(x3D0),\ H(x3D0) = -\_F6(x3D0) - \_C1\, x3D0 + \_C2, \right.$$

$$M0(x1D0, x1D1, x2D0, x2D1, x3D0, x3D1) = \int \frac{\partial^2}{\partial x3D1\, \partial x3D0} \_F4(x3D0, x3D1)\, dx3D1 + \_F5(x3D0),\ M1(x1D0, x1D1, x2D0,$$

$$x2D1, x3D0, x3D1) = \frac{x1D0}{\arcsin(x3D1)^2 \sqrt{1 - x3D1^2}} + \frac{\partial}{\partial x3D1} \_F4(x3D0, x3D1),\ \_F3(x3D0) = -\left(\int \frac{\partial^3}{\partial x3D1\, \partial x3D0^2} \_F4(x3D0,$$

$$\left. x3D1)\, dx3D1 \right) - \left(\frac{d}{dx3D0} \_F5(x3D0) \right) - \left(\int \left(-\left(\frac{\partial^3}{\partial x3D1\, \partial x3D0^2} \_F4(x3D0, x3D1) \right) \right) dx3D1 \right) - \left(\frac{d^2}{dx3D0^2} \_F7(x3D0) \right) \Bigg]$$

Again, we encounter larger integrals, which have to be substituted. Therefore, we use the fictional function

$$K(x_3, \dot{x}_3) = \int \_F4(x_3, \dot{x}_3) d\dot{x}_3 \tag{8.41}$$

to substitute into the result:

Listing 8.42: Substitute with K

```
> result := subs(Int(diff(_F4(x3D0,x3D1),x3D1,x3D0),x3D1) =
D[2,1](K)(x3D0,x3D1), result):
> result := subs(diff(_F4(x3D0,x3D1),x3D1) = D[2,2](K)(x3D0,x3D1), result):
> result := subs(_F4(x3D0,x3D1) = D[2](K)(x3D0,x3D1), result);
```

[3]Note that the variable coordinates has been assigned before in listing 8.31 in order to construct μ.

In order to gain the final μ and M, we simply substitute the result into them:

Listing 8.43: Substitute into μ and M

```
> mu := substitute(result, mu);
> M := substitute(result, M);
```

Finally, we compute the integrable[4] matrix $M\omega$:

Listing 8.44: Compute $M\omega$

```
> M_omega := wedge(M, omega);
```

which returns

$$\begin{aligned} M\omega &= \begin{pmatrix} \xi_1 dx_1 + dx_2 + \xi_2 dx_3 + \xi_3 d\dot{x}_3 \\ dx_3 \end{pmatrix} \\ \xi_1 &= -\frac{1}{\arcsin(\dot{x}_3)} \\ \xi_2 &= \frac{\partial^2 K(x_3, \dot{x}_3)}{\partial x_3 \partial \dot{x}_3} + \_F5(x_3) \\ \xi_3 &= \frac{x_1 + \frac{\partial^2 K(x_3, \dot{x}_3)}{\partial \dot{x}_3 \partial \dot{x}_3} \arcsin(\dot{x}_3)^2 \sqrt{1 - \dot{x}_3^2}}{\arcsin(\dot{x}_3)^2 \sqrt{1 - \dot{x}_3^2}} \end{aligned} \tag{8.42}$$

As a last step, we compute the flat output y of the nonlinear system using the method integration of the toolbox DifferentialForms:

Listing 8.45: Compute the flat output

```
> flat_output := integration(M_omega);
```

This finally yields

$$y = \begin{pmatrix} x_2 - \frac{x_1}{\arcsin(\dot{x}_3)} + \frac{\partial}{\partial \dot{x}_3} \_F6(x_3, \dot{x}_3) \\ x_3 + \_C1 \end{pmatrix}. \tag{8.43}$$

[4] This can easily be shown by using the method extDerivative of the toolbox.

Chapter 9

Conclusions and Future Work

It was demonstrated in this thesis that the mathematical framework which is needed for the flatness determination of linear and nonlinear systems can successfully be implemented using *Maple*. The two toolboxes DifferentialDelays and DifferentialForms, which have been developed in the context of this thesis, provide a suitable framework with an outstanding computational performance. Both are designed for being used as basic structures for future toolboxes which extend the functional range in respect of the flatness determination of linear and nonlinear systems.

This thesis put emphasis on computational performance, reusability and being as much general as possible. Nevertheless, the performance might still be improved by using aspects such as multi-threading.
During the development, the toolboxes were thoroughly tested by using unit tests for each method in order to reach a code coverage above 90%. In order to reduce the time needed for testing in the future, it might be promising to develop a fully automated test framework for the toolboxes which uses the *Maple*-own test suit CodeTools[Test].

Furthermore, the toolbox DifferentialDelays was designed to handle linear systems regardless of whether delays in the system matrices exist or not. The internal algorithms will evaluate whether there are delays and react appropriately. In case of linear systems without delays, this creates a computational overhead which can be omitted by using a specialized toolbox for linear systems without delays.

According to the functional range, there are several promising approaches for both toolboxes: In case of linear systems, the most interesting new functionality would be the implementation of the ring of formal *Laurent* series $\mathcal{K}((\delta))$ in δ with coefficients in $\mathcal{K}$ as described in detail in [2] in order to describe the signal space properly.
In case of nonlinear systems, the next step with good prospects would be to

develop an algorithm for an automated creation of approaches for the matrices μ and M in Theorem 14. The current main problem is that increasing the degrees of freedom in the approaches for μ and M increases the computation time, which is needed for solving the resulting partial differential equations, massively. In addition, we have to consider that the system equations contain possible simplifications which have to be applied in order to solve the occurring equations. At the moment, this still takes a skilled user to decide how to create the approaches and interpret the result.
It might be also advantageous to integrate fraction-free algorithms (see [7]) into the toolboxes to enhance the decomposition of skew polynomial matrices.

Another problem which we are facing right now are the limitations which are given by the pde-solver of *Maple*. Though it is very powerful, the pde-solver of *Maple* cannot solve larger systems of partial differential equations with too many degrees of freedom or integrals. We encountered this in some examples. Thus, we need the user to simplify the equations to a certain level at which *Maple* can solve the remaining pde-system. It may also be promising to determine whether other powerful algebraic pde-solvers are available which could be used in this context.

Appendix A

Files and Worksheets

The following folders and files are also part of this thesis. They include the source code, component tests and examples.

A.1 Source code

- DifferentialDelays - module.map
  source code of the main module of the toolbox DifferentialDelays (see section 7.2.2)

- DifferentialDelays[Decompose].map
  source code of the submodule Decompose (see section 7.2.3)

- DifferentialDelays[LeftFractionUtils].map
  source code of the submodule LeftFractionUtils (see section 7.2.4)

- DifferentialDelays[PiFlatUtils].map
  source code of the submodule PiFlatUtils (see section 7.2.5)

- DifferentialDelays - module.mw
  worksheet, which transforms the map-files into a single mla-file

- DifferentialForms - module.map
  source code of the main module of the toolbox DifferentialForms (see section 7.3.2)

- DifferentialForms[MinimalbasisDecomp].map
  source code of the submodule MinimalbasisDecomp (see section 7.3.3)

- DifferentialForms - module.mw
  worksheet, which transforms the map-files into a single mla-file

A.2 Component tests

- DifferentialDelays - unit tests
  folder, which contains all unit tests of the toolbox DifferentialDelays
- DifferentialForms - unit tests
  folder, which contains all unit tests of the toolbox DifferentialForms

A.3 Examples

- Compute flat output (linear time-varying system) - Example.mw
  example from section 8.1
- Compute flat output (linear system with delays) - Example 1.mw
  example from section 8.2
- Compute flat output (linear system with delays) - Example 2.mw
  advanced example, which is mentioned in section 8.2
- Compute flat output (nonlinear system) - Example 1.mw
  example from section 8.3
- Compute flat output (nonlinear system) - Example 2.mw
  example from section 8.4

These files can be received from the Chair for Automation and Control of the Department for Measurement and Automation of the University of the German Armed Forces in Munich (www.unibw.de/eit8_1).

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

Bibliography

[1] F. Antritter: *On Computational Aspects of Differentially Flat Systems*, Habilitation Treatise, Neubiberg, 2010. 7, 19, 20

[2] F. Antritter, F. Cazaurang, J. Lévine, J. Middeke: *On the computation of π-flat outputs for linear time-varying differential-delay systems*, Systems & Control Letters, 71:14-22, 2014. 1, 10, 20, 22, 23, 25, 159

[3] F. Antritter, J. Lévine: *Flatness Characterization: Two Approaches*, In *Advances in the Theory of Control, Signals and Systems with Physical Modeling, J. Lévine, P. Müllhaupt*, Lecture Notes in Control and Information Sciences, 407:127-139, Springer, 2011. 152

[4] F. Antritter, J. Middeke: *A Toolbox for the Analysis of Linear Systems with Delays*, Proceedings of IEEE CDC - ECC 2011, Orlando, USA, 2011. 1, 7, 8, 9, 10, 12, 15, 23, 24, 146, 149

[5] F. Antritter, G. Verhoeven: *On Symbolic Computation of Flat Outputs for Differentially Flat Systems*, Proceedings of IFAC NOLCOS 2010, Bologna, 2010. 26

[6] F. Antritter, G. Verhoeven: *Ein Werkzeug zur automatisierten Flachheitsanalyse nichtlinearer Systeme*, at-Automatisierungstechnik, 1/2013:60-71, 2013. 32, 33

[7] B. Beckermann, H. Cheng, G. Labahn: *Fraction-free row reduction of matrices of Ore polynomials*, Journal of Symbolic Computation 41, 513-543, 2006. 160

[8] I. Bronstein, K. Semendjajew, G. Musiol, H. Mühling: *Taschenbuch der Mathematik*, Frankfurt am Main, 2006.

[9] M. Bronstein, M. Petkovsek: *An introduction to pseudo-linear algebra*, Theoretical Computer Science 157:3-33, Elsevier Science, 1996. 7, 8, 10, 11, 12

[10] P. Cohn: *Free Rings and Their Relations*, Academic Press, London, 1985. 1, 8, 13, 14

[11] E. Chirka: *Meromorphic function*, *Encyclopaedia of Mathematics*, Springer, Berlin, 2002. 23

[12] F. Chyzak, A. Quadrat, D. Robertz: *Effective algorithms for parametrizing linear control systems over Ore algebras*, Appl. Algebra Eng., Commun. Comput., 16(5):319–376, 2005. 1

[13] M. Fliess, J. Lévine, Ph. Martin, P. Rouchon: *Sur les systèmes non linéaires différentiellement plats*, C.R. Acad. Sci. Paris, I-315:619-624, 1992. 1

[14] M. Fliess, J. Lévine, Ph. Martin, P. Rouchon: *Flatness and defect of nonlinear systems: introductory theory and examples*, International Journal of Control, 61(6):1327-1361, 1995. 19, 20, 25

[15] M. Fliess, J. Lévine, Ph. Martin, P. Rouchon: *A Lie-Bäcklund approach to equivalence and flatness of nonlinear systems*, IEEE Trans. Automat. Control, 44(5):922-937, 1999. 19, 25, 27

[16] M. Fliess: *Some basic structural properties of generalized linear systems*, Systems & Control Letters, 15:391-396, 1990.

[17] G. David Forney, JR: *Minimal bases of rational vector spaces with applications to multivariable linear systems*, SIAM J. Control, 13:493-520, 1975. 15

[18] J. Jezek: *Non-commutative rings of fractions in algebraical approach to control theory*, Kybernetika, vol. 32, no. 1, 81-94, 1996. 10, 12

[19] S. Lang: *Undergraduate Algebra, Third Edition*, New Haven, 2004. 12, 14

[20] J. Lévine: *On Necessary and Sufficient Conditions for Differential Flatness*, Proceedings of IFAC NOLCOS 2004, Stuttgart, 2004. 25, 27, 30

[21] J. Lévine: *Analysis and Control of Nonlinear Systems*, Springer, Heidelberg, 2009. 1, 19, 20, 25, 33, 149

[22] J. Lévine: *On necessary and sufficient conditions for differential flatness*, Applicable Algebra in Engineering, Communication and Computing, 22(1):47-80, 2010. 1, 2, 7, 20, 22, 25, 27, 29, 30, 31, 32, 33, 34

[23] J. Lévine, D. Nguyen: *Flat output characterization for linear systems using polynomial matrices*, Systems & Control Letters, 48:69-75, 2003. 1, 13, 19, 20

[24] S. Limanond: *Adaptive and non-adaptive control of multivariable linear time-varying plants*, Arizona State University, 1994. 143

[25] T. Linz, A. Spillner: *Basiswissen Softwaretest*, dpunkt.verlag, Heidelberg, 2012.

[26] Maplesoft: *Maple Programming Guide*, 2011. 4, 37, 39, 65

[27] Ph. Martin: *Contribution à l'Étude des Systèmes Diffèrentiellement Plats*, PhD thesis, École des Mines de Paris, 1992. 1, 19

[28] J. Middeke: *Normalformen von Matrizen über Schiefpolynomringen und ihre Berechnung*, Diploma Thesis, Oldenburg, 2007. 12, 15, 16

[29] J. Middeke: *A computational view on normal forms of matrices of Ore polynomials*, Doctoral Thesis, Linz, 2011. 7, 9, 15

[30] Ø. Ore: *Theory of non-commutative polynomials*, Annals of Mathematics, vol. 34, 480-508, 1933. 7, 8

[31] P. Rocha, J. Willems: *Behavioural controllability of delay-differential systems*, SIAM J. Control Optimiz., pp. 254–264, 1997. 1

[32] T. Stubblebine: *Reguläre Ausdrücke - kurz & gut*, O'Reilly, Köln, 2008. 58

[33] G. Verhoeven: *Automatisierte Auswertung notwendiger und hinreichender Kriterien für differentielle Flachheit mittels Computer Algebra*, Diploma Thesis, Neubiberg, 2009. 1, 15, 17, 62

[34] http://www.maplesoft.com 3

[35] http://www.notepad-plus-plus.org 4