A Library for Computing with Trees and B-Series

Henrik Sperre Sundklakk

Master of Science in Physics and Mathematics
Submission date: June 2015
Supervisor: Brynjulf Owren, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Preface

This is the result of my master's thesis work, and represents the end of my studies in the Programme in Applied Physics and Mathematics at NTNU.

I am thankful to my supervisor, professor Brynjulf Owren, for his support and help throughout the work on the thesis. I also wish to thank associate professor Magnus Lie Hetland for helping with some of the programming issues.

Henrik Sperre Sundklakk,
Gløshaugen, June 2015

Abstract

Unordered rooted trees and B-series can be used to analyze the properties of many one-step methods for autonomous ordinary differential equations. This thesis describes aspects of trees and B-series necessary to use them to check whether or not a given numerical method has certain properties. This is then used to implement a software library in Python with the ability to investigate the properties of a numerical method given its B-series.

The library represents unordered rooted trees as nested multisets, and include some common functions on trees including order, symmetry coefficient, density and an ordering relation. Operations such as the Butcher product and the grafting product are also implemented. Free trees are also implemented for the use in some of the test for properties of numerical methods.

Furthermore the library includes forests and the product, coproduct and antipode of the Hopf algebra of Butcher, Connes and Kreimer.

B-series are represented as functions corresponding to either characters or infinitesimal characters of the Hopf algebra. The library contains functionality to compose, invert, find the adjoint and conjugate of the functions corresponding to the B-series of numerical methods. Since several of the properties of numerical methods have simpler formulations for the B-series of the modified equation, the library includes functionality to go back and forth between the B-series of a method and its modified equation.

With the help of the above, the library can find a numerical methods order of convergence as well as the order to which a method is symplectic, energy preserving or conjugate to symplectic.

Norsk Sammendrag

Uordnede, rotfestede trær og B-rekker kan brukes til å analysere egenskapene til mange en-skritts metoder for autonome ordinære differensiallikninger. Denne mastergraden tar for seg sider ved trær og B-rekker som trengs for å bruke dem til å sjekke om en gitt numerisk metode har visse egenskaper. Dette er deretter brukt til å implementere et bibliotek i Python som kan brukes til å undersøke egenskapene til en numerisk metode gitt B-rekken dens.

Biblioteket representerer uordnede rotfestede trær som nøstede multimengder, og inkluderer en del funksjoner på trær, blandt annet orden, symmetrikoeffisient, tetthet og en ordningsrelasjon. Operasjoner som Butcher-produktet og poding av trær er også implementert. Frie trær er også implementert for bruk i noen av testene av egenskaper for numeriske metoder.

Videre håndterer biblioteket skoger samt produktet, koproduktet og antipoden i Hopf-algebraen oppkalt etter Butcher, Connes og Kreimer.

B-rekker representeres av funksjoner som tilsvarer karakterer eller infinitesimale karakterer i Hopf-algebraen. Biblioteket kan finne komposisjoner, inverse, adjungerte og konjugerte til B-rekker til numeriske metoder. Siden flere av egenskapene til numeriske metoder er enkere å sjekke på B-rekken til den modifiserte likningen, inkluderer biblioteket funksjoner for å veksle mellom B-rekken til en metode og B-rekken til den modifiserte likningen dens.

De nevnte funksjonene brukes til å finne konvergensordenen til en metode så vel som opp til hvilken orden den er symplektisk, energibevarende eller konjugert til symplektisk.

Contents

Chapter 1

Introduction

Ordinary differential equations (ODEs) are used to model a wide variety of problems in the natural sciences. Common examples include mechanical problems governed by Newton's second law of motion, reaction rates in chemistry, and population models in biology. Some ODEs can be solved analytically, for example the motion of two mass particles moving in each others gravity field. However, in many cases one can not find an analytic solution and must resort to numerical approximations.

This thesis focuses on certain numerical methods for initial value problems of the form

$$\dot{y} = f(y), \qquad y(0) = y_0, \qquad y : \mathbb{R} \to \mathbb{R}^n, \qquad f : \mathbb{R}^n \to \mathbb{R}^n, \tag{1.1}$$

that is first order autonomous ODEs.

This scope is not quite as limited as is might seem, as many non-autonomous and higher order ODEs can be recast in this format, often by increasing the number of dimensions, n, in equation (1.1). Take for example the non-autonomous version of equation (1.1), that is when f depends explicitly on the argument of y, usually named t. Thus $f(y)$ is replaced by $f(t, y)$. It can be made autonomous by renaming t to y_{n+1} and adding the equation $\dot{y}_{n+1} = 1$.

However, when aiming to find good numerical approximations for problems that have to be coaxed into the form (1.1), one might be better off looking for a scheme that takes this additional structure into account. An example of this is mechanical problems where it can be sensible to approximate variables representing position and speed by different methods.

1.1 Numerical Methods

The history of Butcher trees and B-series can be traced back to a growing interest in approximating solutions to ODEs at the turn of the twentieth century. This led among other things, to the development of Runge-Kutta methods ("RK methods" from now on). These are a class of methods based on evaluating f several times per step. They were first thought of and investigated systematically by Runge, Heun and Kutta around 1900.

By 1963 Butcher had developed both a standardized notation to specify an RK method, the Butcher tableau, and a systematic approach to the order conditions. The approach involves rooted trees and B-series. Today these tools are used to investigate a larger class of numerical methods called the B-series methods. It is the wish to use B-series to analyze numerical methods that restricts the discussion to problems on the form (1.1). Furthermore f is assumed to be C^∞ throughout this thesis.

The purpose of a numerical method for (1.1) is to find an approximation to $y(t^*)$ for some given $t^* > 0$. This is typically done by successively calculating approximations y_k to $y(t_k)$ for closely spaced times t_k, basing the next approximation on the already approximated values for y from previous steps. This very general approach allows numerous strategies for calculating the next approximation. However, they all include the explicit Euler method as the most basic method.

The explicit Euler method estimates the next y-value by

$$y_{k+1} = y_k + hf(y_k),$$

where $h = t_{k+1} - t_k$ is called the step size and is assumed to be small. That this gives a reasonable approximation to y_{k+1} is easily justified by

$$\frac{y_{k+1} - y_k}{h} \approx \frac{y(t_{k+1}) - y(t_k)}{h} = \frac{1}{h}\int_{t_k}^{t_{k+1}} f(y(t))\,\mathrm{d}t \approx \dot{y}(t_k) \approx f(y_k). \tag{1.2}$$

Equation (1.2) also exposes the Euler method's most obvious weakness, namely that $f(y_k)$ is not necessarily a particularly good approximation to the average of f for $t \in [t_k, t_{k+1}]$. Another shortcoming of the Explicit Euler method is that it tends to always err in the same direction. When applied to mechanical problems, it inevitably causes an increase in energy over time for no physical reason. This is particularly problematic when simulating a system over a long period of time.

According to [4], three fruitful strategies for improving on the Euler method is to use several past values, perform more calculations per step, and to utilize derivatives of f. These all have advantages and disadvantages, and they may be pursued alone or in combination. In any case the idea is to do somewhat more effort in each step to get a better approximation, thus being able to use a larger step size, which ultimately results in a reduction of the total computational effort.

1.2 One Step Methods and B-Series

RK methods and the other B-series methods are all one step methods, that is methods where y_n is the only previous step used to calculate y_{n+1}. As a consequence y_{n+1} is a function of the previous value and the step size, not unlike a Taylor expansion. This similarity is the underlying concept of B-series, which are little more than Taylor series where the higher derivatives of y are expressed in terms of f and its derivatives instead of y. When doing this, each derivative of y splits into a sum of more complex derivatives of f, so called elementary differentials. These elementary differentials stand in a one-to-one relationship with the unordered rooted trees. This makes it possible to view B-series as series indexed by trees.

Many of the properties of a numerical method are reflected in its B-series or B-series derived from it. Among these are convergence order, symplecticity, energy preservation and symmetry. In many cases working with B-series is more convenient than working directly with the method. This is connected to the fact that somewhat arbitrary formulas for calculating y_{n+1} from y_n are brought into a standardized form where different properties can be tested for in a systematic way.

1.3 The Goals of the Thesis

While B-series offer the convenience of standardized procedures, their use demand a considerable amount of arithmetic operations for even the simplest applications. The goal of this master thesis is a computer library with functionality to manipulate unordered rooted trees and B-series, and that can be used to check the properties of B-series methods.

The usefulness of such a library is obvious to anyone who have ever checked some property of a B-series up to an order larger than three with pen and paper. The existence of at least three more or less ad-hoc efforts by different researchers to automate calculations with trees and B-series further underpins this. Each of these efforts are written by a single person, Ketcheson[15], Murua and Owren respectively, and vary somewhat in scope. It is not unlikely that other researchers have made similar scripts to deal with B-series related tasks too.

1.4 Downloading and Running the Code

Readers of this thesis is encouraged to download and test the code produced during the work with the thesis, collected in a library named PyBS for “Python Butcher Series”. In addition to the classes and functions making up the library, the source code also contains tests of many of the implemented functions. These serve both to demonstrate that the code does what it is supposed to do, and as examples of use.

The source code for PyBS can be downloaded and installed in two different ways. The first will automatically download PyBS and all its dependencies[1] and make PyBS importable as `import pybs` in any Python instance on the system. This is probably the easier and better approach for those who only want to run PyBS.

The second approach is to download the source code from GitHub to some folder on the computer. This is more suitable for those who need to modify or extend the library.

1.4.1 Automated Installation

PyBS has been uploaded to `https://pypi.python.org/pypi`. The command

```
$easy_install pybs
```

or

```
$pip install pybs
```

will download PyBS and install as a "site package" on a Linux based system. After this it should be possible to import `pybs` in any instance of Python on the machine.

1.4.2 Manual Installation

The source for PyBS can be downloaded from `https://github.com/henriksu/pybs` as a zip file or using the version control system Git. Once downloaded, the package can be run by launching the Python interpreter from the outermost `pybs` directory, and writing `import pybs`. At this point PyBS can also be installed in two different ways in order to make it accessible from any Python instance:

Again, starting in the outermost `pybs` directory,

```
python setup.py install
```

will copy the PyBS files to where Python is installed, resulting in an installation similar to the automated one described above. In particular, any changes to the downloaded source files after this will not affect the installation.

Alternatively,

```
python setup.py develop
```

will also make PyBS importable from any instance of the interpreter (or Python script). However, Python will always use the downloaded files, ensuring any modifications are included.

The tests are found in the directory `pybs/pybs/tests`. An overview of the majority of the functions and methods can be built as Sphinx documentation by running the command

```
sphinx-build -b html source build
```

in the directory `pybs/docs`. This overview also reveals how the functions and classes are organized in the library, this is left out from the more mathematically oriented presentation in chapter 4.

[1]PyBS depends on numpy, scipy and enum34.

1.5 The Structure of the Report

Chapter 2 covers the theory of unordered rooted trees and B-series in a bottom up fashion. It starts with the trees before moving on to some linear spaces based on trees and collections of trees called forests. This is followed by sections on numerical methods and B-series in their own right before the use of B-series to investigate numerical methods is explored. The chapter is finalized by a section presenting some numerical methods and their B-series.

Chapter 3 is a shorter chapter describing the tools and strategies used to implement B-series. This includes choice of programming language, program structure and other non-mathematical sides of the implementation.

Chapter 4 describes in detail how trees and B-series are represented and manipulated as well as how this is used to investigate the properties of numerical methods.

Chapter 5 evaluates the result, compares it to the other implementations and discusses how the library can be developed further.

1.6 Relation to Specialization Project

This masters thesis is a continuation of the authors specialization project [24] this fall. It covered much of the matter concerning rooted trees in their own right, forests and linear combinations. B-series of RK methods and finding the B-series of the modified vector field were also implemented. Some of the code and the description in the report concerning these subjects are identical to the specialization project, while other parts are heavily modified, such as the removal of the `FrozenMultiset` class in favor of objects than can be marked as immutable (this also affected forests), and the correction of the fact that the empty tree is actually the empty forest. Furthermore, nothing regarding the ordering of trees, free trees, and coproduct and antipode was implemented before embarking on the master's thesis. The same is true for all the operations and checks on B-series except for the Lie derivative, the modified equation and checking convergence order.

The sections that are taken completely or mainly from the specialization project are marked as such. In addition are some of the paragraphs in 1.1 identical to the introduction to the specialization report.

Chapter 2

Theory

This chapter discusses the tools needed to analyze numerical methods using B-series. Very little of the theory in this chapter is new. However, the bottom-up ordering of the subjects differs somewhat from what is common in the literature, where trees and B-series are defined up front, usually in connection with Runge-Kutta methods, and the rest is introduced as necessary. The bottom-up approach is more appealing when working on implementing B-series. Underlying structure can be made use of to get cleaner implementations with fewer edge cases.

The chapter starts by defining unlabeled, unordered rooted trees in 2.1. The section defines many properties of trees, as well as describing the structure of the set of trees. Free trees are described in subsection 2.1.4, and in subsection 2.1.6 it is pointed out that the real linear space with the set of trees as Hamel basis is the free pre-Lie algebra in one generator.

Section 2.2 concerns the algebra of forests. The algebra can be extended to a Hopf algebra in two different ways. The one introduced by Connes and Kreimer [10] is described in some detail.

Section 2.3 deals with numerical methods and flows. It describes some properties they can have, including symmetry, being symplectic. Conjugacy and modified equations are also introduced.

Section 2.4 defines B-series and shows how they are related to flows. Here the connection between characters and infinitesimal characters on one side and B-series on the other is introduced. It is shown how many manipulations of B-series can be reduced to manipulation of characters and infinitesimal characters.

Section 2.5 describes the patterns left by desirable properties of numerical flows in the B-series coefficients. This includes order of convergence, pseudo-symplecticity, pseudo-energy preserving and being conjugate to symplectic.

The last section, section 2.6 gives some examples of B-series methods and their B-series. The thesis makes no new claims about particular methods. On the contrary, the methods are included since known properties of these common methods have been used extensively to check for flaws in the implementation.

2.1 Trees

2.1.1 The Set of Unlabeled, Unordered, Rooted Trees

The unlabeled, unordered, rooted trees[1] are at the center of this thesis. Their central role in the analysis of certain numerical methods for the equation (1.1) is due to the fact that they stand in a one-to-one correspondence with the elementary differentials. This is explained in section 2.4.

Although the highlight of this section will be to point out that the trees constitute a basis for the free pre-Lie algebra in one generator, we will start with two more descriptive definitions.

The following definition of a multiset will be used throughout the thesis in connection with trees and later forests.

[1]From this point on ‘tree’ is taken to be unlabeled, unordered rooted tree unless otherwise specified.

Definition 1. A **multiset** A is a set in which each element is associated to a natural number, called its **multiplicity** in A. The multiplicity of a non-member is 0.
Notation: In the following, sets are denoted $\{\dots\}$, multisets $[\dots]$, and n-tuples $(\dots)$.

An example of a multiset is the well known notion of a monomial. In a monomial the variables are the elements and the exponents are the multiplicities. In fact, the forests introduced in section 2.2.1 are monomials in trees.

The definition and notation for trees used in this thesis is

Definition 2. Let T denote the set of (unlabeled, unordered and rooted) **trees**. Then

- $\bullet \in T$, and
- if $\tau_1, \dots, \tau_m \in T$, then $\tau = [\tau_1, \dots, \tau_m] \in T$ is the tree obtained by connecting the roots of each of $\tau_1, \dots, \tau_m$ onto a new root.

A definition along the lines of definition 2 can for example be found in [12, def. III.1.1]. This formulation emphasizes the unordered and recursive nature of the trees.

When looping through the child trees of a root, it is customary to denote the total number of child trees by m and the number of distinct child trees by k, thus $k \leq m$. Furthermore, μ_i denoted the multiplicity of τ_i in τ.

While many concepts are simpler when formulated as loops over m trees, they can often be made computationally more efficient by looping over distinct trees and account for the multiplicities directly.

A second definition, based on [20, eq. 2.14], emphasises the graph nature of the trees:

Definition 3. A unlabeled, unordered and rooted **tree** is an (isomorphism class of) partially ordered sets of vertices, U, with exactly one minimal vertex, and fulfilling

$$x, y, z \in U \quad x < z, \;\; y < z \;\Rightarrow\; x < y \text{ or } y < x. \tag{2.1}$$

In graph theory the current trees are called *arborescences* [26] and are typically defined as a directed graph whose underlying undirected graph is a tree in the graph theoretical sense of the word and with a root from which all the other nodes can be reached while respecting the direction of the edges.

In the following the graph view will reappear only in connection with *free trees*. These are the trees, in the graph theoretic sense, obtained by forgetting the direction of edges, or equivalently abandoning the concept of a distinguished root vertex.

One important operation on trees is the Butcher product. It amounts to adding one tree to the multiset of child trees of another tree's root, or more formally

Definition 4. For two trees $u, v \in T$ where $u = [\tau_1, \tau_2, \dots, \tau_m], \tau_i \in T$, the **Butcher product** is the tree $u \circ v = [\tau_1, \tau_2, \dots, \tau_m, v] \in T$.

The Butcher product is neither commutative nor associative. The pair $(u \circ v, v \circ u)$ plays a role in the symplecticity conditions. The Butcher product is also used in the algorithm for the grafting product defined in section 2.1.6.

2.1.2 Properties of Trees

The trees have several intrinsic properties, perhaps the most important of which is the *order* of a tree:

Definition 5. The **order** of a tree, $\tau \in T$, is the number of vertices in τ.
It is calculated recursively as $r(\tau) = |\tau| = 1 + \sum_{i=1}^{m} |\tau_i|$, with $r(\bullet) = 1$ as base case.

The order function partitions T into finite subsets, each consisting of trees of the same order. T^n denotes the set of all trees of order n. Consequently

$$T = \bigcup_{n \in \mathbb{N}} T^n. \tag{2.2}$$

Number of Trees of a Given Order

It is possible to calculate the number of trees of a given order, $|T^n|$, without counting them. The formula

$$|T^n| = a(n) = \begin{cases} n \text{ if } n < 2 \\ \frac{1}{n-1}\sum_{k=1}^{n-1} k \cdot a(k) \cdot s(n-1,k) \text{ else} \end{cases} \quad \text{with } s(n,k) = \sum_{j=1}^{n/k} a(n+1-j\cdot k) \tag{2.3}$$

is given in [22].

Cardinality of T

The partition of T in (2.2) is among other things useful for proving the following proposition.

Proposition 1. $|T| = \aleph_0$

Proof. The proposition is proved by showing that $|T| \geq |\mathbb{N}|$ and $|T| \leq |\mathbb{N}|$ separately.

1. Since the T^ns are pairwise disjoint and there is at least one tree of each order, for example the tall tree (defined later), the cardinality of T is no less that the cardinality of $\mathbb{N}$.

2. Since $|T^n| < \infty$, and the union of countably many countable sets is countable, T is countable.

Thus $|T| = |\mathbb{N}| = \aleph_0$. □

Proposition 1 implies the existence of a bijection between T and $\mathbb{N}$. Such a bijection is established by the total ordering discussed in the next section. The ordering is sometimes useful, both in theory (to define a representative for a free tree) and in the implementation on the computer.

Ordering

The following definition is taken from [16].

Definition 6. For $u, v \in T$, $u < v$ if one of the following is true:

1. $|u| < |v|$.

2. $|u| = |v|$ and the root of u has fewer children than that of v.

3. $|u| = |v|$ and the roots of u and v have equally many children. After sorting the child trees in ascending order according to this order relation, at the first position where the lists of child trees differ, the child tree of u is less than the child tree of v.

This is a strict total order relation, that is given any $u, v \in T$ such that $u \neq v$, one is considered smaller than the other.

In addition to being a strict total ordering, the ordering in definition 6 is a well-ordering and every element has a unique successor (this is in contrast to $\mathbb{Q}$ which is also 1-to-1 with the natural numbers).

Another definition of an ordering is given in definition 6 in [19]. It is based on the *standard decomposition* of trees and repeated almost verbatim in definition 7.

Definition 7. We say that the *standard decomposition* $\mathrm{dec}(u)$ of a tree u of order ≥ 2 is the pair $\mathrm{dec}(u) = (\mathrm{dec}_1(\tau), \mathrm{dec}_2(\tau)) \in T \times T$ such that $\mathrm{dec}_1(\tau) \circ \mathrm{dec}_2(\tau) = \tau$ and $\mathrm{dec}_2(\tau)$ is *maximal.*

Maximality is defined according to the following ordering:

Given $u, v \in T$ we say that $u < v$ if one of the following conditions is fulfilled:

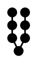
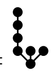
1. $|u| < |v|$,

2. $|u| = |v|$ and $\mathrm{dec}_1(u) < \mathrm{dec}_1(v)$,

3. $|u| = |v|$, $\mathrm{dec}_1(u) = \mathrm{dec}_1(v)$, and $\mathrm{dec}_2(u) < \mathrm{dec}_2(v)$.

Proposition 2. *The ordering relations in definition 6 and 7 are not equivalent.*

Proof by counterexample: Let

$u =$  and $v =$ 

According to definition 6.2 $u < v$, while according to definition 7.2 $v < u$.

An exhaustive search shows that these are the smallest trees for which the two definitions differ. □

In the following the ordering according to definition 6 is used unless otherwise specified.

The ordering can be used to define a bijection between T and $\mathbb{N}$. Analogously it can define a bijection between any subset A of T and the $|A|$ first natural numbers.

Definition 8. The *index* of $\tau \in T$ is defined recursively as $N(\tau) = N(P(\tau)) + 1$ with base case $N(\bullet) = 1$. $P(\tau)$ denotes the immediate predecessor of τ in the ordering in definition 6.

The ability to sort finite subsets of T turns out to be useful when implementing trees on a computer. In particular it enables the storing of coefficients in an array and performing linear algebra.

Other Properties

The following properties are used later in connection with B-series. The first two are both functions $T \to \mathbb{N}$.

Definition 9. The **symmetry coefficient** of $\tau \in T$, denoted $\sigma(\tau)$, is the number of ways in which child trees can be permuted without changing the drawing of the tree. For example $\sigma($ $) = 2$ and $\sigma($ $) = 3$. The symmetry coefficient can be calculated recursively as

$$\sigma(\tau) = \prod_{j=1}^{k} \sigma(\tau_j)^{\mu_j} \mu_j! \quad \text{with} \quad \sigma(\bullet) = 1. \tag{2.4}$$

Definition 10. The **density** of a tree is defined recursively as

$$\gamma(\tau) = |\tau| \cdot \prod_{i=1}^{k} \gamma(\tau_i)^{\mu_i} \quad \text{with} \quad \gamma(\bullet) = 1. \tag{2.5}$$

Note how the density is generally higher the taller the tree is (more levels of recursion).

Definition 11. The **elementary differential** corresponding to a tree, τ is a function from $\mathbb{R}^n$ to $\mathbb{R}^n$ denoted $F(\tau)$. It is defined recursively as $F(\tau) = f^{(m)}(F(\tau_1), \ldots, F(\tau_m))$, with base case $F(\bullet) = f$.

2.1.3 Special Trees

For some purposes it is useful to group trees by their shape. This subsection describes three kinds of trees that appear in connection with B-series.

A *bushy* tree is a tree where all the children of the root are leaves; that is trees of the form $\tau = [\bullet^n]$. At the other extreme are the *tall* trees, where every vertex, except for the last one, has exactly one child. There are exactly one tall and one bushy tree of any given order. Further more, by both definition 6 and 7, the tall tree is the smallest and the bushy tree the largest tree among the trees of a given order.

Another special kind of trees are the *binary* trees. They are the trees where each vertex has at most two children. The binary trees are connected to quadratic right hand sides of (1.1). This is explored further in section 2.4.7.

2.1.4 Free Trees

the concept of *free trees* is used in the analysis of certain properties of numerical methods.

Some rooted trees of the same order are equivalent in the sense that they can be transformed into each other by only changing which node is the root, for example and . This is the key to the use of free trees in the analysis of B-series methods.

Definition 12. A *free tree* is an equivalence class of rooted trees which only differ by which node is denoted root. The free tree corresponding to $\tau \in T$ is denote $\pi(\tau)$. The set of free trees is denoted FT and, analogously to rooted trees, the set of free trees of order n is denoted FT^n.

It is clear from the definition that the free tree $\pi(\tau)$ can be thought of as the undirected graph corresponding to τ.

The definition introduces the function $\pi : T^n \to FT^n$, taking a rooted tree to its corresponding free tree. Since this is a many-to-one relation, there is no true inverse. However, by abuse of notation $\pi^{-1}(t) = \{\tau \in T | \pi(\tau) = t\}$ is called the inverse of π.

Some of the tasks needed for free trees are to

1. Check if $\pi(\tau_1) = \pi(\tau_2)$ for $\tau_1, \tau_2 \in T$.

2. Find all the trees in the set $\pi^{-1}(t)$.

The implemented systematic approach to the two tasks above is based on the approach in [3], and depends on choosing a particular rooted tree as the representative of a free tree. The representative rooted tree of a free tree t, is the tree in $\pi^{-1}(t)$ where none of the children at the root contain more than half of the vertices in the tree. This leaves some ambiguity in the sense that some trees with even order can have two different representations fulfilling the above. The tie is broken by choosing the rooted tree whose largest child is larger according to definition 6. The clever thing about this choice of a representative is that it allows question 1 above to be answered by repeatedly shifting the root towards the biggest child tree in τ_1 and τ_2 and see if they have the same representative.

Free trees have one property that will be important to the analysis of B-series, namely superfluousness.

Definition 13. A free tree is said to be *superfluous* if it has one edge such that when that edge is removed, two identical trees remain.
Trees without such an edge are called *non-superfluous.*

It is obvious that all superfluous free trees have an even number of vertices. One should also note that, with the above definition of a rooted representative, the edge which must be removed in a superfluous tree is the edge between the root in the representative and its largest child tree.

2.1.5 Generalizations

When considering differential equations of slightly different forms from (1.1), it is in some cases possible to develop a theory of trees and series analogous to the current one. The two most important generalizations are briefly described below.

If the components of y are of different qualitative nature, the most typical case being position and momentum in mechanics, one might want to split the equation and use different numerical methods for the different kinds of components. This results in trees where each node has a label, usually called its 'color'.

The other generalization is to let the elements of y range over some other manifold than $\mathbb{R}^n$. Then the order of derivation is no longer immaterial, which corresponds to ordered trees, that is trees that are different if their child trees are listed in a different order.

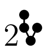
2.1.6 The Free Pre-Lie Algebra

The following is based mainly on [17]. Consider the real linear space with T as a Hamel basis, that is the elements are finite linear combinations of trees. This space is denoted $\mathcal{T}$. The trees of a given order make up subspaces $\mathcal{T}^n = \operatorname{span}(T^n)$ and

$$\mathcal{T} = \operatorname{span}(T) = \bigoplus_{n\in\mathbb{N}} \mathcal{T}^n. \tag{2.6}$$

On this space we introduce the *grafting product*. It is defined as a function $T \times T \to \mathcal{T}$ and extended to $\mathcal{T} \times \mathcal{T} \to \mathcal{T}$ by bilinearity.

Definition 14. For $\tau_1, \tau_2 \in T$ the **grafting product** $\tau_1 \curvearrowright \tau_2$ is the sum of all trees resulting from grafting the root of τ_1 to nodes of τ_2, multiplicities included.

"Multiplicities included" refers to that $\bullet \curvearrowright$ $= 2$ $+$, where the 2 is due to $\bullet$ being grafted to both the left and right branch of .

An important property of the grafting product between two trees is that when grafting τ_1 onto τ_2 all the trees in the result is of order $|\tau_1| + |\tau_2|$. As a consequence, when grafting two elements in $\mathcal{T}$ the coefficient in front of a given tree only depends on the coefficients of trees of smaller orders in the original elements. This observation is key when the need for grafting objects that are effectively infinite linear combinations of trees arises later.

A linear space endowed with a bilinear product, such as $(\mathcal{T}, \curvearrowright)$, is called an *algebra*. In this case the algebra product is neither associative nor commutative. It does however satisfy the pre-Lie property

$$x \curvearrowright (y \curvearrowright z) - (x \curvearrowright y) \curvearrowright z = y \curvearrowright (x \curvearrowright z) - (y \curvearrowright x) \curvearrowright z \quad \forall x, y, z \in \mathcal{T}, \tag{2.7}$$

which can be thought of as being "almost associative".

The importance of equation (2.7) is that it is the necessary and sufficient condition on the algebra product for the antisymmetrization to be a Lie bracket. The antisymmetrization is also known as the tree commutator.

Definition 15. The **tree commutator** is the bilinear and antisymmetric operation $[\cdot,\cdot] : \mathcal{T} \times \mathcal{T} \to \mathcal{T}$ calculated as $[u, v] = u \curvearrowright v - v \curvearrowright u$.

Note that the commutator would be zero for all arguments if and only if the grafting product was commutative. That the tree commutator is a Lie bracket means that, besides being bilinear and antisymmetric, it satisfies the Jacobi identity.

It was shown in [8] that $(\mathcal{T}, \curvearrowright)$ is the free pre-Lie algebra in one generator, namely $\bullet$. The important thing from the last sentence to the current thesis is the fact that repeated grafting of $\bullet$ onto itself generates all the trees. With the above observation about how the order of the product depends on the order of the factors, it is easy to convince oneself that an element of $\mathcal{T}$ with exactly the trees of order n can be constructed by

$$\bullet \curvearrowright (\bullet \curvearrowright (\ldots(\bullet \curvearrowright \bullet)\ldots)) \tag{2.8}$$

with a total of n repetitions of $\bullet$.

That the above is a way of generating all rooted trees can also bee deduced from the fact that grafting $\bullet$ onto some tree corresponds to taking the time derivative of its elementary differential:

$$\frac{\mathrm{d}F(\tau)}{\mathrm{d}t} = F(\bullet \curvearrowright \tau) \tag{2.9}$$

At this place it is worth mentioning that instead of going on to define forests and the Hopf algebra in the next sections, B-series could be defined with what has been done up till this point. This would entail the ad hoc introduction of the "empty tree", an inelegant lack of distinction between characters and infinitesimal characters, as well as a more confusing definition of the B-series of the composition of two methods.

2.2 Forests and the Symmetric Algebra

A mainstay of the remainder of this thesis will be the symmetric algebra on $\mathcal{T}$, denoted H. The elements of H are linear combinations of monomials of trees. The empty monomial is written $\mathbb{1}$ or $\emptyset$ and called 'the empty tree' or 'the empty forest'.

The elements of H are added, subtracted and multiplied by scalars as vectors. In addition they can be multiplied. The product, denoted μ, corresponds to multiplication of monomials and is thus commutative and associative. $\emptyset$ is the identity of this product.

2.2.1 Forests

The monomials of trees mentioned above are called forests since they are collections of trees. Since the multiplication is associative and commutative, forests are finite multisets of trees, just like trees themselves.

Definition 16. A **forest** is a finite multiset whose elements are trees. The set of all forests of trees is denoted F.

Definition 16 may seem very similar to the definition of trees. It is clear that every tree has a forest of child trees and that every forest is the forest of child trees for some tree. It is however useful to distinguish the two when dealing with H.

Contrary to what the bijection above might indicate, when a tree is interpreted as a forest, it is interpreted as the forest containing only that tree. This simplifies the notation for elements in H to e.g. 3 $+ \sqrt{2}$ ^{2}.

Definition 17. The *order* of a forest is the sum of the orders of its trees.

Note that the definition of the order of a forest results in the order being one less than the order of the tree having that forest as its forest of child trees.

2.2.2 Co- and Hopf Algebra Structures

Hopf algebra structures on H are important for the manipulation of B-series. This subsection introduces Hopf algebras through coalgebras and bialgebras. The current presentation is similar to that of appendix B of [2]. A comprehensive treatment of these structures can be found in [25].

A coalgebra is a vector space endowed with a linear *coproduct*, $\Delta : H \to H \otimes H$, and a linear *counit*, $\epsilon : H \to \mathbb{R}$. The coproduct must be *coassociativity*, namely that if two or more coproducts are performed after each other, the result is independent of whether the second coproduct is done on the left or right part of the first coproduct. That is

$$(\mathrm{Id} \otimes \Delta) \circ \Delta = (\Delta \otimes \mathrm{Id}) \circ \Delta. \tag{2.10}$$

The counit must satisfy the following *counital* property

$$(\mathrm{Id} \otimes \epsilon) \circ \Delta = \mathrm{Id} = (\epsilon \otimes \mathrm{Id}) \circ \Delta. \tag{2.11}$$

A *bialgebra* is a space with both an algebra product and a coalgebra structure where the algebra and coalgebra structures interact in the following way:

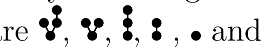
- $\Delta \circ \mu = (\mu \otimes \mu) \circ (\mathrm{Id} \otimes \tau \otimes \mathrm{Id}) \circ (\Delta \otimes \Delta)$ where τ denotes reversing the order of a pair.
- $\Delta \circ u = (u \otimes u)$ where u is the algebra unit $u : \mathbb{R} \to H$, which in general must satisfy $u(a)h = ah \ \forall \ a \in \mathbb{R}, h \in H$. In the present case $u : a \mapsto a \cdot \emptyset$.
- $\epsilon \circ \mu = \mu_{\mathbb{R}} \circ (\epsilon \otimes \epsilon)$ where $\mu_{\mathbb{R}}$ denotes multiplication of real numbers.
- $\epsilon \circ u = \mathrm{Id}$.

A Hopf algebra is a bialgebra with an addition operation, an antipode. An antipode S must have the property that $\mu \circ (S \otimes \mathrm{Id}) \circ \Delta = \mu \circ (\mathrm{Id}, \otimes S) \circ \Delta$, that is taking the coproduct, then the antipode of one of the sides in the result, before taking the product must be independent of which part of the coproduct the antipode was applied to. In additionthe result must be only the empty forest part of the original element in H.

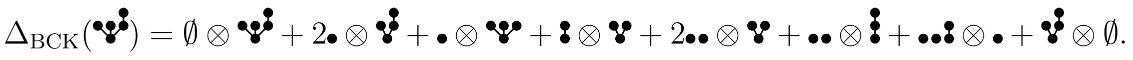
The algebra above can be made a Hopf algebra through two different coproducts, one named after Butcher, Connes and Kreimer (the 'BCK' Hopf algebra), and one named after Calaque, Ebrahim-Fard and Manchon (the 'CEM Hopf algebra').

The importance of the above structures to this thesis is the *convolution product*, $\mu \circ (a \otimes b) \circ \Delta$, between elements of the Hopf algebra's dual space.

2.2.3 The Hopf Algebra of Butcher, Connes and Kreimer

The coproduct in H_{BCK} is based on a concept called *ordered subtrees* of a tree. The subtrees of $\tau \in T$ are all possible trees that can be made by removing child trees from τ. Both the empty tree and τ are subtrees of τ. e.g. the subtrees of are  and $\emptyset$.

The coproduct of τ, $\Delta_{\mathrm{BCK}}(\tau)$, is the linear combination of tensor products with the forest of child trees that were cut off on the left and the corresponding subtree on the right. The coefficients in the linear combination account for the number of ways the subtree can be made. An example is


(2.12)

The Antipode

The antipode for one tree is given as

$$S(\tau) = -\tau - \sum_{\phi \otimes \theta \in \tilde{\Delta}_{\mathrm{BCK}}(\tau)} S(\phi) \cdot \theta \tag{2.13}$$

where $\tilde{\Delta}_{\mathrm{BCK}}$ is the coproduct except $\tau \otimes \emptyset$ and $\emptyset \otimes \tau$. For a forest $t = [\tau_1, \ldots, \tau_m]$

$$S(t) = \prod_{1 \le i \le m} S(\tau_i). \tag{2.14}$$

2.3 Numerical Methods

In this section the focus changes from trees to numerical methods before the connections between the two are explored in section 2.4.

All of the following focuses on numerical methods for autonomous first order ordinary differential equations, that is equations on the form

$$\dot{y} = f(y), \qquad y(0) = y_0, \qquad y : \mathbb{R} \to \mathbb{R}^n, \qquad f : \mathbb{R}^n \to \mathbb{R}^n. \tag{2.15}$$

This section starts by introducing the concept of flows and numerical flows used to discuss one-step methods. It will then introduce Hamiltonian equations and discuss possible properties of the exact and approximate solutions in terms of flows.

The goal of section 2.4 is then to show how some flows can be expressed and manipulated as B-series, and how the properties introduced in this section manifest themselves in these B-series.

2.3.1 Flows

When comparing a numerical method to the exact solution of an ODE on some manifold M, it is useful to consider both as maps $\mathbb{R} \times M \to M$. The map for the exact solution is usually denoted φ_t, and is the map such that $\varphi_t(y_0) = y(t)$, the solution to equation (2.15) at time t given the initial condition $y(0) = y_0$. The appealing thing about the above formulation is that any one step method represent such a map, typically denoted Φ_h. In this notation one step is performed as $y_{n+1} = \Phi_h(y_n)$.

The map for the exact solution has the property of being a flow.

Definition 18. From [11]: A *flow* is a one parameter mapping $\psi_t : M \to M$ such that:

1. $\psi_t \circ \psi_s = \psi_{t+s}$ and $\psi_0(y_0) = y_0$.

2. It is smooth as a mapping $\mathbb{R} \times M \to M$.

The mapping corresponding to a numerical method is not a flow, but is called a *numerical flow.* In particular, a numerical flow will not satisfy property 1 in the above definition. If it did the step size would not matter.

The analysis of a particular one-step method can be thought of as the study of similarities and differences between φ_h and that methods numerical flow, often denoted Φ_h. It is worth keeping in mind that numerical flows are often derived from idealized versions of methods. In particular, rounding errors are ignored and for implicit methods one assumes that the implicit equation is solved exactly.

Definition 19. Whenever it exists, the *inverse* of a method Φ_h is denoted Φ_h^{-1}. It is defined to be $\Phi_h^{-1}(y_{n+1}) = y_n$ if $y_{n+1} = \Phi_h(y_n)$.

In other words, the inverse answers the question: If I got to y by taking one-step with method Φ_h, where did I start out?

Definition 20. The *adjoint* of a method Φ_h is denoted Φ_h^* and is defined to be $\Phi_h^* = \Phi_{-h}^{-1}$.

One important property of the exact flow is *symmetry* or self-adjointness. Symmetric numerical methods are associated with *reversible* differential equations. Many differential equations describing mechanical systems have the property that reversing all velocity vectors of an initial condition results in the same solution trajectory with the time reversed. A precise and general definition of reversible equations is taken from [12]:

Definition 21. Let ρ ba an invertible linear transformation in the phase space of $\dot{y} = f(y)$. This differential equation and the vector field $f(y)$ are called ρ-reversible if $\rho f(y) = -f(\rho y)$ for all y.

A numerical method Φ_h is ρ-reversible if it is symmetric and $\rho \circ \Phi_h = \Phi_{-h} \circ \rho$. It turns out that the latter requirement is met by most methods, including all Runge-Kutta methods.

An interesting operation with numerical flows is to compose them, for two numerical methods Φ_h and Ψ_h this corresponds to taking one step of the first one followed by one step of the other: $(\Phi_{h_2} \circ \Psi_{h_1})(y_0) = \Phi_{h_2}(\Psi_{h_1}(y_0))$. This operation is associative and non-commutative. It is also worth noting that the step length of the composed method $(\Phi_{h_2} \circ \Psi_{h_1})$ is $h_1 + h_2$, making constructions like $\Phi_{\frac{h}{2}} \circ \Psi_{\frac{h}{2}}$ appropriate when constructing new methods by composition.

Since the adjoint satisfies $(\Phi^*)^* = \Phi$ and $(\Phi \circ \Psi)^* = \Psi^* \circ \Phi^*$, a simple way of making a symmetric method from a non-symmetric one is $\Phi_{\frac{h}{2}} \circ \Phi^*_{\frac{h}{2}}$.

2.3.2 Hamiltonian Systems

A large and important class of ODEs on the form (2.15) are the Hamiltonian systems. Some of the concepts and conditions in the following are only defined for Hamiltonian systems.

Hamiltonian systems were discovered as a particularly nice way of expressing the equations of motion for certain mechanical problems. In order to understand it, recall that the state of a mechanical system with n degrees of freedom is typically defined by specifying the position and speed for each of the degrees. This leads to $2n$ initial conditions. The Lagrangian formulation of mechanical problems describe the development through n second order differential equations, each needing two initial conditions. Hamiltonian mechanics describe the same problems through $2n$ first order differential equations, each demanding one initial condition.

Although Hamilton's equations can be written on the form of equation (2.15), when derived from mechanics the following canonical formulation is more common. Hamiltonian systems can also be defined in a more general setting, described in for example part II of [1].

Let q be an n-vector of position coordinates, p be an n-vector of *conjugate momenta* and $H(p, q)$ a function called the Hamiltonian of the system. The Hamiltonian is a constant of motion (first integral) often corresponding to the total energy[2]. The motions are described by the differential equations

$$\dot{p_k} = -\frac{\partial H(p,q)}{\partial q_k}, \dot{q_k} = \frac{\partial H(p,q)}{\partial p_k} \text{ with } 1 \leq k \leq n. \tag{2.16}$$

In the notation of equation (2.15) the system in equation (2.16) can be written

$$\dot{y} = J^{-1}\nabla H(y) \tag{2.17}$$

where

$$y = \begin{pmatrix} p \\ q \end{pmatrix} \in \mathbb{R}^{2n} \text{ and } J = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \in \mathbb{R}^{2n\times 2n}. \tag{2.18}$$

From a mathematical point of view, a system is considered Hamiltonian if it can be written on the form (2.17).

Symplecticity

As described in [12, Ch. VI.2], an important property of the flow of a Hamiltonian system is its *symplecticity.* For one degree of freedom symplecticity corresponds to conservation of area. That is for $B \subset \mathbb{R}^2$ the exact flow of a Hamiltonian system has the property that the area of $\varphi_t(B)$ is equal to the area of B.

For systems with more degrees of freedom, symplecticity corresponds to preservation of the sum of oriented areas, one area per degree of freedom in a mechanical system. For two $\mathbb{R}^{2n}$-vectors x and y, the sum of the areas of the two dimensional parallelograms spanned out by the vectors (x_i, x_{n+i}) and (y_i, y_{n+i}) is calculated as:

$$\omega(x, y) = \sum_i^n (x_i y_{n+i} - x_{n+i} y_i) = x^T J y. \tag{2.19}$$

The function $\omega : \mathbb{R}^{2n} \times \mathbb{R}^{2n} \to \mathbb{R}$ is an example of a symplectic bilinear form. That is, it is bilinear, alternating ($\omega(v, v) = 0 \;\forall\; v$) and nondegenerate (if $\omega(v, w) = 0 \;\forall\; w$ then $v = 0$). If equation (2.15) was on a more general manifold than $\mathbb{R}^{2n}$, symplecticity would be based on another symplectic bilinear form than (2.19).

A linear mapping $A \in R^{2n\times 2n}$ is symplectic if

$$AJA = J \Leftrightarrow \omega(Ax, Ay) = \omega(x, y). \tag{2.20}$$

A general differentiable mapping $g : \mathbb{R}^{2n} \to \mathbb{R}^{2n}$ is called symplectic if the linearization, the Jacobian g', is symplectic.

It is known that the flow, φ_t, of any Hamiltonian system with twice differentiable Hamiltonian function is symplectic [12, Thm. VI.2.4]. Conversely, if the exact flow of a differential equation $\dot{y} = f(y)$ is symplectic, the equation is at locally Hamiltonian [12, (Thm. VI.2.6].

[2]If H depends on time, this is not necessarily true. This possibility is not considered in this thesis.

The numerical flow of a method can also be symplectic. That implies, by the aforementioned theorem, that it is locally the solution of *some* Hamiltonian system and thus conserves the Hamiltonian in question. Since the numerical flow is an approximation of the exact flow, it is reasonable to consider the Hamiltonian it conserves as a perturbation of the Hamiltonian of the exact flow.

Another approach is to search for methods conserving the original Hamiltonian without involving symplecticity. These are called energy preserving methods.

2.3.3 Conjugacy

Some numercal methods are connected through an equivalence relation called *conjugacy.*

Definition 22. [12, p. 222]
Two numerical methods Φ_h and Ψ_h are mutually conjugate if there exists a method χ_h satisfying $\chi_h = y + \mathcal{O}(h)$ such that

$$\Phi_h = \chi_h \circ \Psi_h \circ \chi_h^{-1} \tag{2.21}$$

The importance of conjugacy is that since $\Phi_h^n = \chi_h \circ \Psi_h^n \circ \chi_h^{-1}$ the long time behaviors of the two methods are very similar. This again means that being conjugate to a method with a desirable property is enough to almost have that same property in the long run.

Conjugacy is interesting with respect to all the different desirable properties discussed in this thesis: order of convergence, symplecticity and energy conservation.

2.3.4 The Modified Equation

A useful approach when analyzing a numerical method is to ask the question:

Does it solve *some* differential equation exactly, and if so what?

Such an equation (or rather its right hand side in (2.15)) is called the *modified equation.* It is in general a perturbation depending on h of the exact equation. We will see that for B-series methods the modified equation can also be written as a B-series.

As one might expect, the modified equation is a useful tool for drawing conclusions about the qualities of the numerical solution. For example, the modified equation of the exact flow is f, while the modified equation of a method of order p is an $\mathcal{O}(h^p)$ perturbation of f [12, Thm. IX.1.2].

Another result is that the $\mathcal{O}(h^p)$ perturbation in the modified equation of the adjoint method is $(-1)^p$ times the corresponding term in the modified equation of the method [12, Thm. IX.2.1]. This leads to the fact that the modified equation of symmetric methods do not have perturbation terms of odd order [12, Thm. IX.2.2]. Consequently, all symmetric methods are of an even order.

The two previous paragraphs are examples of the tendency for properties that have quite intricate definitions for flows to have simpler definitions in the language of modified equations.

In a similar fashion as above, the modified equation of a symplectic method is again Hamiltonian provided the differential equation in question is Hamiltonian [12, Thm. IX.3.1].

An idea similar to that of modified equations is to ask what right hand side should have been fed into a particular method for the numerical solution to be the exact solution. This is called *modifying integrators* [9]. It is an easy way of utilizing knowledge of the derivative of f to make an improved method from an existing one. This approach is not pursued further in this thesis.

2.4 B-Series

This section discusses how the exact flow can be thought of as a series in trees, that is a B-series. We will also see that whenever the flow of a particular numerical method can be written as a B-series, many interesting properties can be deduced from it. Many, but not all, numerical methods have a B-series expansion. Those who do are called *B-series methods.*

We will then turn to the details of manipulating B-series and draw conclusions about the numerical method they represent. An important part of this will be to realize that the set of B-series should be split in two, corresponding to the characters and the infinitesimal characters of H.

The nature of B-series will be treated more thoroughly in 2.4.2. Until then, the following definition is sufficient.

Definition 23. The series

$$B_{hf}(a,y) = a(\emptyset)y + \sum_{\tau\in T} \frac{h^{|\tau|}}{\sigma(\tau)} a(\tau)F(\tau)(y)$$

is called a **B-series**.

A B-series is called *consistent* if $a(\emptyset) = 1$. This is not the same as saying that the numerical method is consistent, since the latter amounts to being at least of order 1.

2.4.1 The B-Series Expansion of the Exact Solution

The following outlines how φ_t, the flow of the exact solution, can be expressed as a B-series. It is taken almost verbatim from the authors specialization project, but is included for completeness.

The connection between equation (2.15) and trees is established by considering the Maclaurin series of $y(h)$. As pointed out in for example [12], the derivatives involved can be expressed in terms of f and its derivatives, rather than y and its derivatives. The first derivative of y with respect to t is obviously

$$\dot{y} = f(y). \tag{2.22}$$

The next two derivatives can be written

$$\ddot{y} = \frac{\mathrm{d}\dot{y}}{\mathrm{d}t} = f'(y)\dot{y} = f'(y)f(y) \tag{2.23}$$

$$\dddot{y} = \frac{\mathrm{d}\ddot{y}}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\left(f'(y)f(y)\right) = f''(y)\left(f(y), f(y)\right) + f'(y)f'(y)f(y). \tag{2.24}$$

Here f' is the Jacobian matrix and f'' is a bilinear map. In general the n^{th} derivative of f is a n-linear map. Even more insight is gained by writing equation (2.23) and (2.24) on component form. This is done in the equations below, where superscript denotes component and subscripts partial derivatives with respect to a y-component.

$$\ddot{y}^k = \frac{\mathrm{d}f^k}{\mathrm{d}t} = \sum_i f_i^k f^i$$

$$\dddot{y}^k = \frac{\mathrm{d}\ddot{y}^k}{\mathrm{d}t} = \sum_i \left(\frac{\mathrm{d}}{\mathrm{d}t}\left(f_i^k\right) f^i + f_i^k \frac{\mathrm{d}}{\mathrm{d}t}\left(f^i\right)\right) = \sum_i \left(\sum_j \left(f_{i,j}^k f^j\right) f^i + f_i^k \sum_j \left(f_j^i f^j\right)\right) = \sum_{i,j} f_{i,j}^k f^j f^i + f_i^k f_j^i f^j$$

From the above it is easy to convince oneself that the interaction of the product rule and the chain rule will cause an ever increasing number of terms of ever increasing complexity. To see the pattern more clearly, it is useful to consider the fourth derivative:

$$y^{(4)} = f'''\left(f,f,f\right) + 3f''\left(f'f,f\right) + f'f''\left(f,f\right) + f'f'f'f$$

First, one should note that the number 3 comes from the fact that the term following it is arrived at by three different ways from the third derivative. Secondly, and more importantly, note that each term starts with a derivative of f. The order of this derivative is always equal to the number of factors following, each of which corresponds to one of the terms in a previous derivative of y. This is exactly what leads to the recursive structure of the trees.

It is also important to note that since the order in which partial derivatives are taken is irrelevant (recall the assumption $f \in C^\infty$), the ordering of the arguments to a derivative of f is irrelevant. This is the reason that *unordered* trees are appropriate representatives for these derivatives.

If $\mathbb{R}^n$ in (2.15) is exchanged for a more general manifold this may no longer be true, leading to ordered trees. This path is not pursued further in this thesis.

As this suggests, the Maclaurin series can be expressed as a B-series. One obvious difference, that $h^{|\tau|}$ os divided by $\sigma(\tau)$ instead of $|\tau|!$ is a matter of convention. Since

$$\sigma(\tau) = \frac{|\tau|!}{\alpha(\tau)\cdot\gamma(\tau)} \tag{2.25}$$

the difference is just a factor, which is absorbed by $a(\tau)$. The meaning of $\alpha(\tau)$ is explained below.

The rule in this B-series is denoted e and happens to be

$$e(\tau) = \frac{1}{\gamma(\tau)}. \tag{2.26}$$

By comparing equation (2.26) to equation (2.25) one can see that $\alpha(\tau)$ will be the actual factor in front of $F(\tau)$ when the B-series is written out.

Special B-series

In addition to the B-series of the exact solution, at least two other B-series are worth mentioning here.

The first one if the B-series of the modified equation of the exact solution. It is denoted

$$\delta_\bullet(\tau) = \begin{cases} 1 \text{ if } \tau = \bullet \\ 0 \text{ otherwise} \end{cases}. \tag{2.27}$$

The other one is found for example in [20]. It is defined as

$$\mathbb{1}(\tau) = \delta_\emptyset(\tau) = \begin{cases} 1 \text{ if } \tau = \emptyset \\ 0 \text{ otherwise} \end{cases}, \tag{2.28}$$

and is used in the logarithm in section 2.4.5.

2.4.2 B-Series as Characters

Considering definition 23 with the previous subsection in mind, it is clear that what distinguishes the B-series of the exact solution from those of numerical methods are the coefficients $a(\tau)$. Since B-series are in general infinite, the coefficients must be specified as a function $a : T \to \mathbb{R}$ based on an algorithm that calculates the coefficient for any particular tree. Thus, ignoring $a(\emptyset)$ and the less interesting parameters h, f and y, the B-series stand in a one-to-one relationship with the dual space $\mathcal{T}^*$.

Regarding $a(\emptyset)$ only two values are of interest, namely 0 and 1. All B-series representing the flow of a numerical method has $a(\emptyset) = 1$, since $a(\emptyset) \neq 1$ corresponds to non-consistent methods. All the B-series with $a(\emptyset) = 0$ represent right hand sides of the differential equation, such as modified equations and modifying integrators.

It turns out that it is appropriate to extend the functions' domain from T to F, thus considering elements in the dual space of H instead of $\mathcal{T}^*$, see [17].

For B-series representing methods, that is $a(\emptyset) = 1$, this is done by demanding that if t is the forest containing the trees $\tau_1, \tau_2, \ldots, \tau_m$, then

$$a(t) = \prod_{i=1}^{m} a(\tau_i). \tag{2.29}$$

For B-series representing vector fields, that is $a(\emptyset) = 0$, the extension to forests is done by defining a to be zero for all forest except those containing exactly one tree.

These definitions make the flow and right hand sides the characters and infinitesimal characters of H respectively.

2.4.3 Composition

For B-series that represent numerical methods one can consider their composition, inverse and adjoint as described in subsection 2.3.1. In all three cases the resulting is again a B-series.

Starting with composition, consider two B-series methods $\Phi_h(y) = B_{hf}(a, y)$ and $\Psi_h(y) = B_{hf}(b, y)$ and their composition $\Psi_h \circ \Phi_h$. The B-series of the composition is $B_{hf}(b, B_{hf}(a, y))$. To see this, pick a y_0 and let $\tilde{y}_1 = \Phi_h(y_0)$. Then the result of the composite step is $y_1 = \Psi_h(\tilde{y}_1)$ and $B_{hf}(b, B_{hf}(a, y))$ is found by substitution.

Theorem III.1.10 in [12] shows that the composite method is a B-series method, denoted $B_{hf}(ab, y)$, and how the value of the new rule ab at any given tree can be calculated from the value of a on certain forests and the value of b on certain trees. Thus the product ab is consistent for a consistent a and any b.

The composition of a and b is actually a group operation on the characters of H. The resulting group is called the Butcher group. The value of ab can be calculated as:

$$ab = \mu_{\mathbb{R}} \circ (a \otimes b) \circ \Delta_{\text{BCK}}, \tag{2.30}$$

where $\mu_{\mathbb{R}}$ is ordinary multiplication in $\mathbb{R}$.

The coproduct in any Hopf algebra gives rise to such a group on its characters. The inverse of an element in such a group is known to be

$$a^{-1} = a \circ S_{\text{BCK}}, \tag{2.31}$$

where S is the antipode as described in 2.2.3. Since this is the inverse of composition of methods, $B_{hf}(a^{-1}, y)$ is the B-series of the inverse of the numerical flow corresponding to $B_{hf}(a, y)$ in the sense described in definition 19.

Note that equation (2.31) depends on a being a character operating on forests in accordance with equation (2.29).

Special Case

In the special case that b is

$$b(\tau) = \begin{cases} 1 & \text{if } \tau = \bullet \\ 0 & \text{else} \end{cases} \tag{2.32}$$

lemma 1.9 in [12, Ch. III] states that the composition can be calculated in a particularly simple way. In this case $ab(\emptyset) = 0$ and

$$ab(\tau) = \prod_{j=1}^{k} a(\tau_j)^{\mu_j}, \quad \tau = [\tau_1^{\mu_1}, \dots, \tau_k^{\mu_k}].$$

2.4.4 Adjoint and Symmetry

At this point, the only new operation needed to obtain the B-series of the adjoint method is to reverse the time step of a B-series. By inspecting definition 23, it is clear that all that is needed is to reverse the sign of $a(\tau)$ whenever $|\tau|$ is odd.

Since the adjoint method is the inverse of the time-reversed method, this allows the construction of the B-series of the adjoint method for any B-series method.

An important use of the adjoint method is to check for symmetry (self-adjointness). With the above functionality in place, it is trivial to check for symmetry up to any finite order by the pattern outlined in 2.5.

2.4.5 Modified Equations and the Lie Derivative

Much of the content of this subsection was explored in the authors specialization project. However, the text is rewritten to incorporate the concept of infinitesimal characters and the use of *log* and *exp*.

For a given B-series method with B-series $B_{hf}(a, y)$, its modified equation is the differential equation

$$h\dot{\tilde{y}} = B_{hf}(b, \tilde{y}) \tag{2.33}$$

such that the exact solution of (2.33) at time h is given by the method in question. The h on the left hand side of (2.33) corresponds to reducing the exponent of h in the terms by one (see definition 23). This is necessary since it is supposed to be a perturbation of the exact equation, f. From the requirement that the modified equation is a perturbation of the original differential equation, one can also establish that $b(\emptyset) = 0$ and $b(\bullet) = 1$. Note that unlike the B-series of the method, the B-series of the modified equation is of the type $b(\emptyset) = 0$, or in the words of 2.4.2, b is an infinitesimal character of H.

The rule for the B-series of the modified equation can be derived from that of the flow of the method in at least two different ways:

1. Through the Lie derivative.
2. By a construction similar to the Taylor series of $\log(x)$ at $x = 1$.

The rest of this subsection is devoted to explaining the two methods.

The Lie Derivative

Given two vector fields b and c on $\mathbb{R}^n$ it is possible to take the derivative of c with respect to b. That is, the directional derivative of c in the direction of b at every point. The result is again a vector field and is called the *Lie Derivative* of c with respect to b.

Given a B-series rule b such that $b(\emptyset) = 0$ and an arbitrary c, lemma IX.9.1 in [12] gives the equation

$$\partial_b c(\tau) = \begin{cases} 0 & \text{if } \tau = \emptyset \\ \sum_{\theta \in SP(\tau)} c(\theta) \cdot b(\tau \setminus \theta) & \text{else} \end{cases} \tag{2.34}$$

for the Lie derivative. The set $SP(\tau)$ is called the splittings of τ. They are defined to be all the trees found as what remains of τ after one child tree has been cut off. $\tau \setminus \theta$ denotes the child tree which has to be cut off to get θ.

Note that if b act on forests as an infinitesimal character 2.4.2, the summation can be taken over the coproduct since $SP(\tau)$ corresponds to exactly those terms of $\Delta_{\text{BCK}}(\tau)$ where the left element consists of a forest with one tree. That is

$$\sum_{\theta \in SP(\tau)} c(\theta) \cdot b(\tau \setminus \theta) = \sum_{\phi \otimes \theta \in \Delta_{\text{BCK}}(\tau)} c(\theta) \cdot b(\phi). \tag{2.35}$$

The Modified Equation by Lie Derivative

As Lemma IX.9.2 in [12] gives the formula for the B-series rule b for the modified equation given a B-series with rule a, where a satisfies $a(\emptyset) = 1$ and $a(\bullet) = 1$. As pointed out above $b(\emptyset) = 0$ and $b(\bullet) = 1$. For the remaining trees

$$b(\tau) = a(\tau) - \sum_{j=2}^{|\tau|} \frac{1}{j!} \partial_b^{j-1} b(\tau), \tag{2.36}$$

where ∂_b^k denotes k repeated Lie derivations.

At first glance equation (2.36) seems self referring. However, it does allow $b(\tau)$ to be calculated recursively.

Modified Equation by Logarithm

The definition of the logarithm of a B-series rule is modeled on the logarithm of a real number. The Taylor expansion at $x = 1$ of the logarithm of a real number is

$$\log(x) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(x-1)^n}{n} \tag{2.37}$$

The logarithm of an element $a \in H^*$ s.t. $a(\emptyset) = 1$ is defined in [21] to be

$$\log(a) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{(a - \mathbb{1})^{*n}}{n} \text{ and } \log(a)(\emptyset) = 0, \tag{2.38}$$

where $\mathbb{1} = \delta_\emptyset$ denotes the identity element of the composition product, and $(\cdot)^{*n}$ denotes composition n times.

The sum in equation (2.38) turns out to be finite with $|\tau|$ terms when evaluated on a concrete tree τ. This makes the definition suitable for actual calculations. It is well known, e.g. according to [9], that the logarithm of the rule of the B-series of a numerical method, is the rule of the modified equation.

Numerical Flow from a Modified Equation by Exponentiation

Just like the logarithm is defined by extending the definition of the Taylor series, the exponential function can be defined in an analogous manner by mimicking

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}. \tag{2.39}$$

According to [21], for any $\alpha \in H^*$ with $\alpha(\emptyset) = 0$ the exponential is defined as

$$\exp(\alpha) = \sum_{n=0}^{\infty} \frac{\alpha^{*n}}{n!} \text{ and } \exp(\alpha)(\emptyset) = 1. \tag{2.40}$$

This is the inverse of the logarithm and provides a way of finding the B-series of a numerical method from the B-series of its modified equation. It is worth noting that the exponential and logarithm are defined for any elements of the dual space satisfying the requirements on the value at $\emptyset$, not just the characters and infinitesimal characters. However, the logarithm of a character is an infinitesimal character, and the exponential of an infinitesimal character is a character.

2.4.6 Conjugacy

In section 2.3.3 the conjugate of a method $\Phi_h = B(a, y)$ with respect to some other given method $\chi_h = B(c, y)$ such that $\chi(\emptyset) = 1$ is defined to be

$$\Psi_h = \chi_h \circ \Phi_h \circ \chi_h^{-1} \tag{2.41}$$

Since it has been shown how B-series can be composed and inverted, it is in principle possible to find the rule for the conjugate in this way. However, the conjugate of a with respect to c can also be computed by a series of commutators. This is in principle the same commutator as for $\mathcal{T}$. However, two problems arises:

1. a and c must be corrected by σ since σ is factored out in definition 23.

2. B-series are infinite, so the coefficients of the resulting series must be computed in some systematic way on demand.

The conjugate of a with respect to c is computed in a way resembling the Taylor expansion

$$\sum_{n=0}^{\infty}(-1)^n \frac{x^n}{n!}, \tag{2.42}$$

but with "raising to the power n" replaced by n nestings of $[c, \cdot]$. Thus

$$\mathbf{conj}(a, c) = a - [c, a] + \frac{1}{2}[c, [c, a]] + \frac{1}{6}[c, [c, [c, a]]] - \dots \tag{2.43}$$

As above, this does not seem to give a finite algorithm, but it turns out that for trees of order m all terms with more than m nested commutators evaluates to 0.

Note that this approach depends on an a priori known χ, which is rarely the case. In most cases were conjugacy appears, the question is if Φ is conjugate to a method with some desirable property. Under such circumstances neither χ or Ψ are known nor sought. This calls for quite different approaches, one of which is discussed in section 2.5.4.

2.4.7 Simplifications from Additional Knowledge About the Right Hand Side

Up to this point the only assumption on f, the right hand side of equation (2.15), has been sufficient differentiability. However, certain properties of f results in B-series with additional properties. This can reduce the complexity of calculations with B-series, and, more importantly, result in certain methods having better properties than when they are applied to general problems.

The author is aware of two such properties, problems in only one dimension and quadratic right hand side. The former will not be discussed further in this thesis.

A right hand side is said to be quadratic when f is on the form

$$f(y) = Q(y) + By + c, \tag{2.44}$$

where c is a vector, B a square matrix and Q a quadratic form. The ith component of a quadratic form Q is on the form $Q_i(y) = y^T A_i y$ where A_i is a symmetric matrix.

For quadratic vector fields all elementary differentials corresponding to non-binary trees disappear. This means that both the B-series of the flow and the B-series of the modified equation contain terms with a proper subset of T. However, this is a result of the elementary differentials being zero due to the particular ODE under consideration, and not due to the B-series rule, which is derived form the method (The method may apply to more general ODEs). A consequence is that the property must be taken explicitly into account during the transition to modified equation.

2.5 Checking Properties

While the last section focused on manipulating B-series, this section concerns investigating the properties of numerical methods through their B-series. The properties of interest are order of convergence, symplecticity and energy preservation as well as being conjugate to these. As already mentioned, many properties can be investigated more conveniently by considering the modified equation instead of the flow.

While demonstrating that a method is sympletic or energy preserving can be done in the language of B-series, the proofs depends on novel arguments involving an understanding of the B-series rule and no attempt has been made to automate it. However, properties can hold and be verified up to a finite order, and this can be done with a finite effort. The approach to checking properties will be to do tests on trees of increasing order until the test fails or a predetermined maximal order is reached.

2.5.1 Order of Convergence

The most basic property of a numerical method is its order of convergence. It is also the simplest property to check. At the end of this subsection we will argue that the investigation of other properties can benefit significantly from knowing the order of convergence.

Definition 24. [12, p. II.1.2] A numerical method Φ_h has *order p* if p is the largest integer such that

$$y(h) - \Phi_h(y(0)) = \mathcal{O}(h^{p+1}) \text{ as } h \to 0. \tag{2.45}$$

The term $y(h) - \Phi_h(y(0))$ is called the *local error.* The *global error* is $y(nh) - \phi_h^n(y(0))$. For a numerical one-step method of order p the global error is $\mathcal{O}(h^p)$ for $nh = const.$

The above implies that for a numerical one-step method of order p, a halving of the step size will result in the global error shrinking by a factor of approximately 2^p.

The following theorem reveals how the order of a method manifests itself in its B-series and the B-series of the modified equation.

Proposition 3. *Given a B-setiies method of order p with B-series $B_{hf}(a, y)$ and modified equation $h\tilde{y} = B(b, \tilde{y})$, then p is the larges integer such that*

$$a(\tau) = e(\tau)\ \forall\ \tau \in \{\tau \in T : |\tau| \le p\}. \tag{2.46}$$

p is also the largest integer such that

$$b(\tau) = 0\ \forall\ \tau \in \{\tau \in T : 1 < |\tau| \le p\}. \tag{2.47}$$

The first part of the preceding theorem is a well known fact about B-series, a proof can be found in [12, Ch. III.1]. The second part is a consequence of the fact that the modified equation of a p-th order method is $\mathcal{O}(h^{p+1})$ close to f and can for example be found in [9, Ch. 4.2].

A useful consequence of equation (2.46) is that it is often sufficient to check favorable properties for trees of higher order than the methods order of convergence. This follows directly from the fact that the B-series is identical to the exact solution up to its order of convergence, and thus all properties of the exact solution holds for these coefficients. Since comparing the coefficients of a B-series to the B-series of the exact solution is simpler than the other tests that follows, this leads to a useful simplification of the test and an increase in computation speed.

2.5.2 Symplecticity

Most of the following theorem is from [12, Thm. VI.7.6]. The statement about the modified equation can be found in [12, Thm. IX.9.3].

Proposition 4. *Consider a B-series method $\Phi_h(y) = B_{hf}(a, y)$for equation* (2.15). *The following statements are equivalent:*

- *the coefficients $a(\tau)$satisfies*

$$a(u \circ v) + a(v \circ u) = a(u) \cdot a(v)\ \forall\ u, v \in T. \tag{2.48}$$

- *the coefficients $b(\tau)$ of the B-series of the modified equation satisfies*

$$b(u \circ v) + b(v \circ u) = 0\ \ \forall\ \ u, v \in T. \tag{2.49}$$

- *quadratic first integrals of the form $Q(y) = y^T C y$, where C is symmetric, are exactly conserved.*
- *The method is symplectic if the differential equation is Hamiltonian.*

Since equation (2.48) and (2.49) amounts countably infinitely many conditions, they cannot be check exhaustively. This is however not necessary in many cases, since they often do not hold for all $\tau \in T$. This results in a new concept, namely pseudo-symplecticness.

The following theorem is a result of [12, Thm. IX.7.6] and [12, Thm. I.9.3].

Proposition 5. *A B-series method is of pseudo-symplectic order q* iff *q is the larges integer such that*

$$a(u \circ v) + a(v \circ u) = a(u) \cdot a(v) \;\; \forall \;\; u, v \in T \;\; s.t. \;\; |u| + |v| \leq q \tag{2.50}$$

holds. For modified equations the equivalent condition is

$$a(u \circ v) + a(v \circ u) = 0 \;\; \forall \;\; u, v \in T \;\; s.t. \;\; |u| + |v| \leq q. \tag{2.51}$$

From the above it is clear that one can think of a symplectic method as a method of pseudo-symplectic order ∞. Theorem 5 suggests an algorithm for finding the pseudo-symplectic order of an arbitrary B-series method by checking (2.50) or (2.51) for all pairs such that $|u| + |v| = n$ before moving to the next larger integer. Implementations along these lines are described in 4.9.3.

2.5.3 Properties as Subspaces

The article [5] describes how the modified equations of symplectic and energy preserving B-series methods form linear subspaces of the B-series. It also describes how a basis for each of these spaces can be formed such that each basis vector is a linear combination of trees of the same order.

The subspace of $\mathcal{T}^n$ in which the nth order part of any energy preserving B-series lies is denoted $\mathcal{T}^n_H$. Analogously $\mathcal{T}^n_\Omega$ is the subspace of $\mathcal{T}^n$ in which the nth order part of any Hamiltonian B-series lies.

The above allows for an algorithm where looping over u, v-pairs described above is replaced by determining whether or not the vector of all coefficients of trees of a given order can be expressed as a linear combination of certain other vectors. Obviously, if this can be done for all trees of order less than or equal n in a given B-series, the B-series has the property up to and including order n.

Note that this approach with linear subspaces is only possible in the domain of modified equations, not the B-series of methods directly. Also worth mentioning is that in [5] the factor in front of trees is taken to be the B-series rule divided by $\sigma(\tau)$ when checking for membership of $\mathcal{T}^n_H$ or $\mathcal{T}^n_\Omega$. For practical reasons the following presentation accounts for the symmetry when setting up the vectors, making the comparison with a given B-series rule simpler. In case of Hamiltonian B-series this results in σ not appearing at all, while in the case of energy preserving B-series the result is that σ appears although it does not appear in equation (4) of [5].

The Hamiltonian Property as a Subspace

Theorem 1 in [5] states that a basis for $\mathcal{T}^n_\Omega$ can be indexed by the non-superfluous free trees of order n and each basis vector is a linear combination of the rooted trees corresponding to the same free tree. The basis vector corresponding to a free tree is a linear combination of all its rooted representative. The coefficients are determined by choosing one of the rooted trees and set its coefficient in the linear combination to 1. The coefficients for the other rooted trees are -1 for trees whose root is an odd number of edges away from the root of the chosen representative, and 1 for trees whose root is an even number of edges away from the root of the chosen representative.

Energy Preservation as a Subspace

Theorem 3 in [5] describes how to construct a basis for $\mathcal{T}^n_H$.

Some trees are basis vectors by them self. Thus no particular investigation of their coefficients in B-series is necessary. These are trees such that the free tree resulting from grafting it onto a new node, $\pi(\tau \curvearrowright \bullet)$, is superfluous.

Another group of trees are trees that does not appear in the B-series of the modified equation of any energy preserving method. These are the trees such that $\pi(\tau \curvearrowright \bullet)$ is symmetric and non-superfluous.

The reminding trees in $\mathcal{T}^n$ are part of basis vectors with two trees. Theorem 2 in [5] describes these pairs. Given a tree, τ_1, there is some freedom to choose another tree with witch a basis vector for $\mathcal{T}_H^n$ can be made. In particular, the other tree, τ_2, must satisfy $\pi(\tau_1 \curvearrowright \bullet) = \pi(\tau_2 \curvearrowright \bullet)$. The basis vector is then $\sigma(\tau_1)\tau_1 \pm \sigma(\tau_2)\tau_2$ where the sign is determined by the number of edges between the root of $\tau_1 \curvearrowright \bullet$ and that of $\tau_2 \curvearrowright \bullet$. If this number is even, the sign is negative, if it is odd, the sign is positive.

In order to bring forth a complete basis of $\mathcal{T}_H^n$ one can collect all the trees τ mapping to the same free tree $\pi(\tau \curvearrowright \bullet)$. Then an arbitrary of these nth order trees is picked and pair with each of the other trees to form basis vectors for $\mathcal{T}_H^n$ as described above.

2.5.4 Conjugate to Symplectic

As mentioned in section 2.4.6, problems concerning conjugacy are often regarding whether or not a given method is conjugate to another one with a desirable property, without seeking to construct the other one. The paper [13, chapter 3] describes conditions for a method of order p to be conjugate to a symplectic method up to order $2p$. The approach is to derive a number of conditions on the change of variables series, denoted χ_h in section 2.4.6, and check whether or not they can be satisfied simultaneously.

The conditions uses a subset of the splittings of a tree, $SP(\tau)$, namely those resulting from removing one leaf. This subset is denoted $SP_*(\tau)$.

In addition, for any B-series rule, a, the notation $a(u, v)$ with $u, v \in T$ is taken to mean $a(u \circ v) + a(v \circ u)$. Note that the function is symmetric in the sense that the order of the two arguments is irrelevant.

Theorem 3.6 in [13] states that necessary and sufficient conditions for a method of order p, whose modified equation has B-series rule α, to be conjugate to symplectic up to order $p + r \leq 2p$, is that there exists a symmetric function c on two trees satisfying

$$\alpha(u, v) = -\sum_{\hat{v}} c(u, \hat{v}) - \sum_{\hat{u}} c(\hat{u}, v) \tag{2.52}$$

for all unordered tree pairs u, v such that $p < |u| + |v| \leq p + r$. Furthermore, if the method is symmetric, the conditions are always satisfied for even $|u| + |v|$.

The function c is connected to a change of variables series, but that is irrelevant to the use of the theorem.

If it is known that the conditions are satisfied for $r - 1$, the new conditions that have to be checked for r are unordered pairs such that $|u| + |v| = p + r$. In this case equation (2.52) turns into a linear system of equations for the values of $c(u, v)$ for all pairs such that $|u| + |v| = p + r - 1$.

2.6 Some B-series Methods

This section describes some well known B-series methods. They are included since they have been used to verify the correctness of the implementation.

2.6.1 Runge-Kutta Methods

The following is mainly from the authors specialization project.

RK methods can be thought of as an attempt to improve on the explicit Euler method by approximating the average of f by a weighted average of s approximations, k_i ($i = 1, 2, \ldots, s$), at different points in the interval $[t_k, t_{k+1}]$. The approximations, called the stages, are computed as

$$k_i = f(y_k + h \sum_{j=1}^{s} a_{ij} k_j), \tag{2.53}$$

and the final approximation as

$$y_{k+1} = y_k + h \sum_{i=1}^{s} b_i k_i. \tag{2.54}$$

The coefficients a_{ij} and b_i are typically written

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array}$$

where A is the s-by-s matrix of a_{ij} coefficients, b is the vector of b_i coefficients, and c is a vector such that $c_i = \sum_{j=1}^{s} a_{ij}$ represents the point in the interval $[t_k, t_{k+1}]$ where stage i is evaluated. c does not play any role for autonomous equations.

[20, Thm 2.5], attributed to Butcher, states that given any finite number of B-series coefficients, an RK method can be constructed so that its B-series has the prescribed coefficients.

Since every RK method is a B-series method (proof: For any RK method the corresponding B-series can be constructed by the method below), the above makes the RK methods dense in the B-series under some suitable metric, for example $d(a, b) = \frac{1}{n}$ where a and b are B-series (characters) and n is the largest integer such that a and b are equal for all trees of order less than n.

Considering the above, it is not surprising that many B-series methods are RK methods. Given a Butcher tableau, the B-series rule of an RK method can be calculated as follows [12]:

The rule in the B-series corresponding to one step of a given RK method, μ, is typically denoted ϕ_μ and its values are called elementary weights. The derivation of ϕ_μ is most elegantly done by applying the composition rule in section 2.4.3 to (2.53) and (2.54). Here we will make do with citing the formulas used in the implementation.

In the following A and b are the coefficient matrix and weight vector respectively of the RK method μ. Following the notation in [12] $\boldsymbol{g}$ and $\boldsymbol{u}$ are vector functions on trees (also depending on μ). The $\prod$-sign denotes Hadamard products.

$$\phi_\mu(\tau) = \sum_i b_i \boldsymbol{g}_i(\tau) = b^T \boldsymbol{g}(\tau) \tag{2.55}$$

$$\boldsymbol{g}(\tau) = \prod_{i=1}^{m} \boldsymbol{u}(\tau_i) \tag{2.56}$$

$$\boldsymbol{u}(\tau) = A\boldsymbol{g}(\tau) \tag{2.57}$$

2.6.2 Kahan's Method

Kahan's method is a numerical method whose properties depend on f being quadratic [7]. When applied to a quadratic problem it is an RK method with B-series rule

$$a(\tau) = \frac{1}{2^{|\tau|-1}\sigma(\tau)} \tag{2.58}$$

for tall trees and 0 otherwise.

When applied to quadratic problems it is a second order method, conjugate to a symplectic method up to order 4 [7, prop 2].

2.6.3 The Average Vector Field Method

The Average Vector Field Method ('AVF method') is an example of a B-series method that is not an RK method. It is defined as

$$\frac{y_{n+1} - y_n}{h} = \int_0^1 f\left((1-\xi)y_n + \xi y_{n+1}\right) d\xi. \tag{2.59}$$

For a tree $\tau = [\tau_1, \tau_2, \ldots, \tau_m]$ its B-series is given by

$$a(\tau) = \frac{1}{m+1} \Pi_{i=1}^{m} a(\tau_i). \tag{2.60}$$

The above formulas as well as the following facts can be found in [6]. The AVF method is energy preserving, symmetric and of order 2. It is conjugate to symplectic up to order 4.

Chapter 3

Method

This chapter describes some of the non-mathematical sides of the implementation.

3.1 Programming Language

The implementation is written in the programing language Python. This is a high level language, that is it allows a fair amount of abstraction at the cost of slightly longer running times than lower level languages such as C or Fortran. This trade-off is well suited for the task at hand, where implementing a large number of different functionality is more important than optimizing the running time. This assertion is based on the assumption that the alternative to doing it on the computer is to perform the calculations with pen and paper. It is obviously better to have a computer implementation of one more thing than to halve the running time of something that is already implemented. Another important reason for choosing Python is the current authors knowledge of and experience with the language, saving the time needed to acquaint oneself with a new language.

Other alternatives included Maple, Mathematica and Sage. Attempts to use Sage were undertaken both during the project in the fall and during the work on this master's thesis. Sage is an open-source computer algebra system which aims to become an alternative to the commercial software packages widely used for mathematical computations today. Sage organizes mathematics with the help of categories (in the mathematical sense of the word). This has the advantage of a mathematically very correct representation of elements and relations between different concepts, as well as standardization. Sage also has a Notebook for web browsers which can print objects in a way more similar to ordinary mathematical notation.

The potential advantages of implementing trees and B-series in Sage are obvious from the above. The attempts to gain sufficient proficiency with Sage stranded on the current authors lack of knowledge of category theory, as well as the fact that no one with a deep knowledge of Sage could be found at NTNU. Another potential disadvantage with Sage is that the advantages are only available as long as things are done the way they are meant to. An example of this being a potential problem was the task of representing forests as monomials of trees. It turned out that, to the best of out knowledge, monomials in Sage are either in a finite a-priori give set of variables, or at the most in variables automatically indexed by the natural numbers $(x_1, x_2, x_3, \dots)$. This would leave the choice between writing a fully fledged implementation of monomials in arbitrary variables or to make some lash-up that does not really fit in with the rest of Sage.

In the end, the lack of support from an experienced Sage programmer as well as the fear to run into other concepts that would have to be programmed very carefully to comply with Sage's category theoretical framework, Sage was abandoned.

3.2 Programming Strategy

Trees and B-series were unknown to the author before embarking on the project. New knowledge has been implemented continuously, and the correctness, of both understanding and implementation, has been tested by automated tests.

This approach allowed a stepwise refinement of the implementation. The use of tests is crucial to that end, as tests provide some assurance that the implementation is correct by comparing it to well known or hand calculated results when it is first written. However, a just as important result of this is the accumulation of a large number of tests that can be run in seconds at the click of a button. Frequent testing allows unintended side-effects of new code to be noticed and corrected early. The existence of tests covering most aspects of the code also allows changes to old code without tracing all consequences manually.

3.3 Focus of the Implementation

The implementation is based on taking a function, supplied by the user, that calculates the coefficient for any given tree. The properties of the B-series implied by that function is then investigated. Although there is a way of producing the B-series rule for an arbitrary Butcher tableau, the Butcher tableau is never used explicitly to investigate the properties of a method. This was omitted, even though certain properties can be showed to hold exactly by performing a finite number of checks on the Butcher tableau, in order to focus on B-series.

Analogously no attempt was made to allow the user to tell the implementation that a method has a certain property, e.g. symplecticity, and exploit this knowledge when checking other properties, or track it through composition (e.g. the composition of two symplectic methods is again symplectic).

Another thing that was omitted was to allow methods to contain free parameters and calculate conditions they must satisfy in order to achieve a certain order, a certain pseudo-symplectic order or something else. This limits the scope of the package to investigating already known methods, not searching for new promising ones.

3.4 Odds and Ends of the Implementaiton

3.4.1 Exact Arithmetic

Most of the implementation is based on exact arithmetic. With some methods being highly recursive, they might introduce significant rounding errors if floating point numbers were used, or at least make it non-trivial to choose a cut off. On the other hand side, it might pose a problem for methods whose B-series coefficients are irrational.

3.4.2 Memoization

From chapter 2 it is clear that many functions are defined recursively on the unordered rooted trees. Recursive algorithms are often elegant, intuitive and easier to implement that the alternatives. However, they are known for causing significant overhead, in terms of both memory and computation, compared to iterative algorithms for the same problem[1]. Another issue with recursive algorithms is that the same sub problem might appear many times over.

A solution, which does not really affect the apparent structure of the calculations, is to evaluate a function only the first time it is called with a particular set of arguments, and store the result along with the argument. The next time the function is called with equal arguments, it is not evaluated, but instead

[1]Many, but not all problems that can be solved recursively can also be solved iteratively.

the stored result is returned. This is called memoization. Of course the expected or average reduction in computational effort hinges upon the number of times the same problem appears.

Considering that the number of trees of a given order grows exponentially, and that every tree is a forests of smaller trees, it is clear that memoization is very well suited for formulas that depend recursively on a tree's child trees.

3.4.3 Other Libraries

Some tasks are generic and common, and for some of these existing solutions have been used. Some calculations are done by calls numpy and scipy [14], in particular the few linear algebra calculations that are done.

Memoization is realized with a Python construct called a *decorator*. Decorators is a convenient way of altering the behavior of a function in a specific way. The memoization decorator used in the implementation is taken from `https://wiki.python.org/moin/PythonDecoratorLibrary#Memoize`.

For drawing trees graphically, the packages Planarforest [18] and tikz2svg[2] were used.

[2]Found at `https://gist.github.com/jbenet/9449155`.

Chapter 4

Implementation

This chapter describes the main result of the thesis work, the implementation of the mathematical structures introduced in chapter 2. The focus of this presentation is on how the mathematical objects are realized, how they interact and which algorithms are used for the calculations. Details on the syntax is deferred to the overview described in section 1.4.2. The tests also provide many examples of actual use.

Any implementation of B-series depends on a sound implementation of trees that are unordered, rooted and unlabeled. Although various tree structures are abundant in computer science, unordered trees are not. This calls for the construction of a data structure carefully designed to facilitate the various calculations needed in a reasonably efficient way. Since the cardinality of T^n grows rapidly with n, memory efficiency is also a potential issue. Everything related to trees, including free trees and how to loop over all trees, is treated in section 4.1.

Section 4.2 deals with algebraic structures on trees, in particular everything connected to $\mathcal{T}$, $\mathcal{F}$ and the Hopf algebra. This is the section with the largest deviations from theory. Only what's needed for analyzing B-series is implemented.

Section 4.3 deals with the *manipulation* of characters and infinitesimal characters of H, and section 4.9 describes how properties of B-series are investigated.

Thus the structure of this chapter is similar to that of chapter 2.

4.1 Trees

There are several ways of representing trees. The most important consideration when implementing unordered rooted trees on a computer is how the unorderedness of the children is handled. An important requirement in that respect is a reasonable algorithm for deciding whether or not two trees are equal.

The implementation of trees is motivated by definition 2. It utilizes the implementation of multisets described next.

4.1.1 Multisets

Concepts similar to mathematical multisets are known and used in computer science, often called "bags" or "multisets". However, the current author has not found implementations in Python of the mathematical concept of multisets. Admittedly Python has a built-in class named `collections.Counter` with behavior similar to a multiset. It is however not suitable to represent a mathematical multiset as it allows negative and non-integer multiplicities and does not offer typical multiset operations.

Instead of using a `Counter`, a multiset-class was created. It is based around a Pyhton `dict`-object[1]. The elements in the multiset are stored as keys and the multiplicities as the accompanying values. An advantage of using a hash table is that, from a users point of view, the elements are unordered. Another advantage is that equality of multisets is reduced to the built-in equality tests for dictionaries.

[1] `dict` is the built-in hash table in Python.

A technical point is that since trees are nested multisets, the multiset objects must be valid dictionary keys. That is being immutable, supporting hashing and comparing for equality (the `==` operator). The problem in this respect is that Python's dictionaries are mutable and consequently does not have a hash-function. On the other hand mutable multisets are necessary, or at least useful, when constructing new multisets from old ones.

An obvious solution is to make two multiset classes, one mutable and one immutable, and cast explicitly back and forth. The implemented solution, picked up from the cloning protocol in [23], is to make a class with a boolean variable to indicate whether an object is mutable or not. This allows initiating a mutable object, manipulate it and then mark it as immutable.

There are more operations that can be performed on multisets than on ordinary sets. Only those needed for the manipulation of trees are described here, the rest are described in the documentation (described in section 1.4.2). The functionality needed from multisets by PyBS is quite limited. The most used operations are:

Initiate A new multiset can be made from an old one (copying), another mapping[2], or a list or tuple of elements.

Add an element If `m` is a multiset and `elem` some element `m[elem] +=1` and `m[elem] = a` increases the multiplicity and sets it to `a` respectively. This works independently of whether `elem` is already in the multiset. This is the operation that underlies the Butcher product.

Remove a tree `m[elem] -=1` and `del m[elem]` respectively decreases the multiplicity by one and removes an element. In the first case, if the multiplicity reaches 0, the element is removed from the multiset altogether.

Iteration A multiset, `m`, can be iterated over in three different ways:

1. `m.iterkeys()` will iterate over the distinct elements of `m`.
2. `m.iteritems()` will iterate over element-multiplicity-pairs.
3. `m.elements()` will iterate over the elements, each repeated according to its multiplicity.

The two first are well known from Python's dictionaries, the third is based on the second.

Multiset sum of two multisets A and B is the multiset where the multiplicity of each element is the sum of its multiplicities in A and B. It is calculated by looping through the elements of B and increasing the corresponding multiplicities in A accordingly. This corresponds to the product between forests.

The hash function is not used explicitly, but is used when a multiset is stored in another multiset or dictionary. The hash value of a `ClonableMultiset` is calculated as

```
result = 0
for pair in self._ms.iteritems():
    result ^= hash(pair)
return result
```

The variable `pair` will be a tuple containing an element and its multiplicity. Note how the hashes of these pairs are combined into the hash of the multiset as a whole by xor-ing them (the ^=-symbol). The choice of operator is somewhat arbitrary, but it must be commutative and associative to ensure that equal multisets have the same hash, independent of the ordering produced by `iteritems()`.

Any good hash function returns values evenly distributed in the range and such that similar objects have wildly different hashes. The above algorithm meets this standard, assuming that the built-in hash function for tuples meets it.

[2]Python's `collections.Mapping`

4.1.2 Rooted Trees

With an implementation of multisets in place, the implementation of the unordered rooted trees is straight forward. The class `UnorderedTree` inherits from `ClonableMultiset` and adds some new methods. Among the added features are initiating from and printing a string representation of a tree, and code to produce graph representations of trees.

An important method on `UnorderedTree`-objects is the Butcher product, `u.butcher_product(v)`, which returns a new tree object corresponding to $u \circ v$. Furthermore, the ordering in definition 6 is implemented as the ordinary comparison operators `<` and `>`.

The basic properties of trees defined in section 2.1.2 are also implemented. The implementations follow the definitions, but exploits multiplicities wherever possible. In the following brief descriptions `t` $= \tau$ is an `UnorderedTree`, k is the number of different child trees of `t`, and μ denotes multiplicities:

Order is calculated as $|\tau| = 1 + \sum_{i=1}^{k} \mu_i |\tau_i|$ by the method `t.order()`.

Density is calculated as $\gamma(\tau) = |\tau| \cdot \prod_{i=1}^{k} \gamma(\tau_i)^{\mu_i}$ by the method `t.density()`.

Symmetry is calculated as $\sigma(\tau) = \prod_{i=1}^{k} \sigma(\tau_i)^{\mu_i} \mu_i!$by the method `t.symmetry()`.

Alpha is calculated as $\alpha(\tau) = \frac{|\tau|!}{\gamma(\tau)\cdot\sigma(\tau)}$ by the method `t.alpha()`.

Elementary differentials are returned as a string by `t.F()`. For example if `t` is , `t.F()` returns `f"(f,f)`.

The number of different children, k, is found by a direct call to the underlying dictionary, it is returned by `t.no_uniques()`.

The total number of children, m, is calculated as $\sum_{i=1}^{k} \mu_i$ by `t.number_of_children()`.

One might expect the base cases $|\bullet| = \gamma(\bullet) = \sigma(\bullet) = 1$ need particular attention, but the properties of the empty sum and the empty product ensures this is actually not necessary. The implementations are carefully made to mimic this and thus avoid testing for special cases.

The methods `t.is_tall()`, `t.is_bushy()` and `t.is_binary()` can be used to tell whether the tree `t` has the properties described in 2.1.3. The first and last check that the number of children is allowed (less than two and less than three respectively) and that the children recursively satisfy the same criterium. Bushyness is tested by verifying that the list of child trees of the root, multiplicities not included, is a list containing only the tree with one node (or that it is empty, since $\bullet$ is considered to be bushy).

In the following $\bullet$ is denoted `leaf` in code excerpts and examples. It has no children, and is thus represented by an empty multiset. More complicated trees can be derived in different ways, the simplest being to specify the forest of child trees.

The following code example demonstrates how trees can be made from simpler trees.

```
t = UnorderedTree()
forest = Forest([t, t])
t2 = UnorderedTree(forest)
```

The above results in `t2` = .

4.1.3 Free Trees

Free trees are represented by objects the `FreeTree`-class. These objects contain the rooted tree representative of the free tree as it is defined in 2.1.4. They also contain a dictionary to hold the rooted trees for which it is the free tree. The value at each rooted tree is 1 or -1 depending on whether its root is an even or odd number of vertices away from the root of the representative. Last they contain boolean variables to indicate whether the dictionary is complete, and whether or not the free tree is superfluous or symmetric.

The comparison operators compare free trees by applying definition 6 to their representatives. In addition `t.order()` returns the order of the free tree. The other properties of rooted trees are not defined.

Finding the free tree corresponding to a given rooted tree is implemented as the method `get_free_tree()` on `UnorderedTree`-elements. The algorithm works as one would expect from the description in 2.1.4. It is worth noting that this approach will not identify the entire set $\pi^{-1}(\pi(\tau))$. Since all uses of this set in PyBS needs the partition of an entire T^n into sets of trees corresponding to the same free tree at once, completing the above mentioned dictionary for π^{-1} is done for all trees of the same order at once. This is described in the next subsection.

4.1.4 The Tree Pool

Explicit use of trees is mostly done by the part of PyBS concerned with checking properties. These functions typically needs all the trees in order of increasing order, or they need all the trees of a given order, either sorted in a repeatable order or in an arbitrary order.

Since this is a reoccuring pattern, a class for creating, caching and returning trees was created. It is not mandatory in order to use `UnorderedTrees`, but is the default way of providing trees to B-series related parts of the library.

The pool of trees is one instance called `the_trees`,created at startup, of a class called `Trees`. `the_trees` contains a dicitonary of `TreeOrder`-objects. The `TreeOrder`-object for order n is accessed as `the_trees[n]` and contain all the trees of a given order. Furthermore, the first time the trees of that order are required sorted by the rule in definition 6, it does so and changes it internal collection of the trees from a `set` to a `tuple`. Thus the sorting is only performed once.

Another important feature of the `TreeOrder` class is the method `free_trees()`. The first time it is run, it loops through all the rooted trees to partition them according to their corresponding free tree. Note that this is the only implemented way of making sure the dictionary of rooted trees in the `FreeTree` objects are complete.

Furthermore, non superfluous free trees are provided in a similar fashion. The first time the method `non_superfluous_trees()` is called, it loops through the free trees and sorts out the non-superfluus ones.

Since some of the code is dealing with vectors whose elements correspond to coefficients in front of the trees of a given order, the following methods are provided by `TreeOrder` objects: `index(tree)` where `tree` is either an `UnorderedTree` or a `FreeTree`. The result is the trees index in a zero-indexed list of the trees or free trees of that order. `non_superfluous_index(tree)` does the analogous thing for `FreeTree` objects that are non-superfluous.

The methods `tree_with_index(i)`, `free_tree_with_index(i)` and `non_superfluous_tree_with_index(i)` are the inverses of the above (given an integer they return the appropriate tree object).

As one might imagine, the `TreeOrder`-class contains a fair amount of internal logic to generate, store, and sort various trees. This is mostly book-keeping to make sure that all the rooted trees are known before the free trees are generated, that all the free trees are known before the non-superfluous ones are filtered out, and to keep track of whether or not some collection of trees have already been sorted.

The only process to be discussed in detail is the generation of rooted trees. The operation of producing all unordered rooted trees with n nodes is based on the way the one vertex tree, $\bullet$, generates T by the method shown in equation (2.8). Note that any tree of order n can be obtained by grafting $\bullet$ onto some tree of order $n-1$. Thus if `previous_trees` is the set of trees of order $n-1$ the following code will generate all trees of order n:

```
new_trees = set()
for tree in previous_trees:
    new_trees.update(_graft_leaf(tree))
```

The `update` method on Python's built-in `set`-objects will add any elements in its argument that are not already in `new_trees` to `new_trees`. This is important since most trees can be made by grafting • to different trees of order $n-1$.

The `_graft_leaf()` function above does not perform grafting as it is defined in definition 14, but rather a simplified version returning a set instead of a linear combination with coefficients counting in how many ways any given tree was constructed. The code for `_graft_leaf()` is

```
def _graft_leaf(tree):
    result = set()
    result.add(tree.butcher_product(leaf))
    for subtree in tree.keys():
        amputated_tree = tree.sub(subtree)
        replacements = _graft_leaf(subtree)
        for replacement in replacements:
            with amputated_tree.clone() as tmp:
                tmp.inplace_add(replacement)
            result.add(tmp)
    return result
```

The ordinary grafting product is discussed in section 4.2.2.

The function `number_of_trees_of_order(n)` returns the corresponding integer, while `number_of_trees_up_to_order(n)` returns the number of trees up to and including order n. These two functions are based on the algorithm in equation 2.3.

4.2 Combinations and Forests

The previous section focuses on implementing representations and ways of constructing trees, along with some functions involving one tree at a time. This section on the other hand focuses on the linear spaces involving trees, that is mostly $\mathcal{T}$ and H, along with some functions and operations associated with these. The unifying theme is the need for the class `LinearCombination`.

The following sections are laid out as follows: Linear combinations are discussed in 4.2.1. Then forests are introduced in 4.2.4. Grafting and the commutator is treated in 4.2.2 and 4.2.3, before forests, splittings, the coproduct and the antipode are explained in 4.2.4 to 4.2.7.

Of these only the coproduct and the antipode was not treated in [24]. The user should be aware that several of the concepts discussed in this section are only implemented to cope with what is needed for B-series later.

4.2.1 Linear Combinations

The elements of $\mathcal{T}$ and H are finite linear combinations of trees and forests respectively. However, unlike $\mathbb{R}^n$ the sets of basis vectors are (countably) infinite. This makes it impractical to represent vectors as arrays where each index of the array is taken to be the coefficient in front of some basis vector.

The solution to this in the implementation is the class `LinearCombination`. Just like the multiset it uses a dictionary. The basis vectors are used as keys and the coefficients are stored as the corresponding values. This ensures each basis vector is only stored once (duplicate keys are forbidden).

`LinearCombination` shares the multisets ability to delete keys once their coefficient is set to 0, and to return 0 as the coefficient of keys not present. Furthermore the operators `+`, `+=`, `-`, `-=` and multiplication by scalars are all implemented. The method `dimensions()` returns the number of distinct trees.

It is important to note that `LinearCombination` objects do not care about the type of the keys/basis vectors they are given, not even that they are all the same type. Thus the same class is used to hold the result of grafting as the result of the coproduct, and there is also nothing stopping the used from adding two such objects together.

An example of how to use the `LinerarConbination` class is given below. `tree1` and `tree2` are two different rooted trees.

```
theSum = LinearCombination()
theSum[tree1] += 2
theSum[tree2] = 3.14
```

The above would result in a linear combination of two trees, $2 \cdot \tau_1 + 3.14 \cdot \tau_2$. And

```
 theSum * 2.5
```

would return $5 \cdot \tau_1 + 7.85 \cdot \tau_2$ as a new `LinearCombination` object.

4.2.2 Grafting

The function `graft(other, base)` returns the sum of all trees resulting from grafting the tree `other` onto the tree `base`. The following code demonstrates how the grafting product is performed recursively. The approach is to remove one child tree from the base and replace it with the trees resulting from grafting the other tree onto the child tree. The factor in front of such a tree in the final result is the factor in front of the tree added in as a “replacement” for the child tree that was cut off in the recursive grafting. In addition, in order to save time, the above procedure is prerformed once for each distinct child tree of the root, and the result is multiplied by the child trees multiplicity in `base`.

```
def graft(other, base):
    result = LinearCombination()
    if base == empty_tree:
        result += other
    elif other == empty_tree:
        result += base
    else:
        result += base.butcher_product(other)
        for subtree, multiplicity1 in base.items():
            amputated_tree = base.sub(subtree)
            replacements = graft(other, subtree)
            for replacement, multiplicity2 in replacements.items():
                new_tree = amputated_tree.add(replacement)
                result[new_tree] += multiplicity1 * multiplicity2
    return result
```

4.2.3 The Commutator

With the above implementations of `LinearCombination` and `graft()`, the implementation of the tree commutator is simply:

```
def TreeCommutator(op1, op2):
    return graft(op1, op2) - graft(op2, op1)
```

The implementation of the commutator for linear combinations of trees is

```
def linCombCommutator(op1, op2, max_order=None):
    ... # Casting op1, op2 to LinearCombination here if necessary.
    result = LinearCombination()
    for tree1, factor1 in op1.items():
        for tree2, factor2 in op2.items():
            if (not max_order) or order(tree1) + order(tree2) <= max_order:
                result += (factor1 * factor2) * TreeCommutator(tree1, tree2)
    return result
```

The option `max_order` truncates the result. This may be useful if the arguments are truncated B-series.
The commutator for entire B-series can also be calculated. This is described in 4.6.

4.2.4 Forests

The implementation of forests is straight forward, and quite similar to that of trees. The class `Forest` inherits `ClonableMultiset`. A `Forest`-object can be initiated from a list of trees and it provides some basic methods to investigate and manipulate it. These include getting the order (as in definition 17), getting the number of trees (both different trees and total), comparing for equality, and multiplying two forest (by multiset sum). In addition a simple string representation based on the string representation of trees is provided.

The forest with no trees, $\emptyset$, often needs to be treated as a special case in the code. Whenever it occurs, it is denoted `empty_tree` due to its role as the first term in B-series representing numerical flows.

4.2.5 Splittings

The implementation of the splitting of a tree described in 2.4.5 returns a `LinearCombination` of pairs. The pairs are represented by Python tuples. The first element is the tree cut off, and the second one is the subtree.

The implementation uses two functions, `split()` and `_split()`, in order to get the multiplicity of the edge case $(\tau \otimes \emptyset)$, where τ is the tree whose splitting is calculated, right. The majority of the work is done by the `_split()`-function. Again, the approach is to call itself recursively for each different child tree. The details of the algorithm is clear from the following code excerpts:

```
def split(tree, truncate=False):
    result = _split(tree)
    if not truncate:
        result[(tree, empty_tree)] = 1
    return result

def _split(tree):
    result = LinearCombination()
    for childtree, multiplicity in tree.items():
        amputated_tree = tree.sub(childtree)
        result[(childtree, amputated_tree)] = multiplicity
        childSplits = _split(childtree)
        for pair, multiplicity2 in childSplits.items():
            new_tree = amputated_tree.add(pair[1])
            new_pair = (pair[0], new_tree)
            result[new_pair] = multiplicity * multiplicity2
    return result
```

The boolean variable `truncated` is explained in connection with the function for the modified equation in 4.8.

4.2.6 Coproduct

Similar to the alogrithms for grafitng and splitting, the implemented algorithm for finding coproducts is reursive. However, the additional machinery needed on each level of the recursion is more elaborate than for splittings and grafting.

The approach is to find the coproduct of all child trees by recursive calls to `subtrees()`. Then one tensor product from each of the resulting linear combinations is chosen in all possible ways by `itertools.product()`. The subtrees are then grafted together and the forests of cuttings collected to a new forest. In addition the multiplicities are multiplied to find the multiplicity of the new tensor product which is added to the final result. This procedure is carried out by the code snippet below.

```
def subtrees(tree):
    result = LinearCombination()
        ... # Code dealing with forests go here. See next code excerpt.
    result[(Forest((tree,)), empty_tree)] = 1
    if tree == leaf:
        result[(empty_tree, tree)] = 1
        return result
    tmp = [subtrees(child_tree) for child_tree in tree.elements()]
    tmp = [elem.items() for elem in tmp]
    for item in product(*tmp):  # iterator over all combinations.
        tensorproducts, factors = zip(*item)
        multiplicity = 1
        for factor in factors:
            multiplicity *= factor
        cuttings, to_be_grafted = zip(*tensorproducts)
        with Forest().clone() as forest_of_cuttings:
            for forest in cuttings:
                forest_of_cuttings.inplace_multiset_sum(forest)
        result[(forest_of_cuttings, UnorderedTree(to_be_grafted))] += \
            multiplicity
    return result
```

It is interesting to note that the above algorithm is completely devoid of procedures and arguments explicitly using ordered trees as is common in the definition of this coproduct.

The code omitted in the above code excerpt deals with extending the coproduct to forests. The approach is to recursively find the copropduct of one of the trees and the rest of the forest, and combine the results accordingly:

```
    if tree == empty_tree:
        result += (empty_tree, empty_tree)
        return result
    elif isinstance(tree, Forest):
        if tree.number_of_trees() == 1:
            for elem in tree:
                return subtrees(elem)
        else:
            for elem in tree:
                amputated_forest = tree.sub(elem)
                break
            for pair1, multiplicity1 in subtrees(elem).items():
                for pair2, multiplicity2 in subtrees(amputated_forest).items():
                    if isinstance(pair1[1], UnorderedTree):
                        pair1_1 = Forest((pair1[1],))
                    else:
                        pair1_1 = pair1[1]
                    if isinstance(pair2[1], UnorderedTree):
```

```
                    pair2_1 = Forest((pair2[1],))
                else:
                    pair2_1 = pair2[1]
                pair = (pair1[0] * pair2[0], pair1_1 * pair2_1)
                result[pair] += multiplicity1 * multiplicity2
        return result
```

4.2.7 Antipode

The antipode is implemented as suggested by equation 2.13 and 2.14.

```
def antipode_ck(tree):
    result = LinearCombination()
    if tree == empty_tree:
        result[empty_tree] = 1
        return result
    elif isinstance(tree, Forest):
        result[empty_tree] = 1
        for tree1, multiplicity in tree.items():
            for i in range(multiplicity):
                tmp = LinearCombination()
                for forest1, multiplicity1 in antipode_ck(tree1).items():
                    for forest2, multiplicity2 in result.items():
                        tmp[forest1 * forest2] += multiplicity1 * multiplicity2
                result = tmp
        return result
    result[Forest((tree,))] -= 1
    for (forest, subtree), multiplicity in _subtrees_for_antipode(tree).items():
        for forest2, coefficient in antipode_ck(forest).items():
            result[forest2.add(subtree)] -= coefficient * multiplicity
    return result
```

The above relies on the function `_subtrees_for_antipode()`, which corresponds to $\tilde{\Delta}_{\mathrm{BCK}}$, that is the coproduct except $\tau \otimes \emptyset$ and $\emptyset \otimes \tau$.

4.3 B-Series

The implementation does not treat B-series directly. Instead objects representing elements of the dual space of H, H^*, are used. This might be impractical, but it has the advantage of separating the manipulation of characters and infinitesimal characters from the consequences of particular properties of the right hand side of the ODE, such as being Hamiltonian, quadratic, and so forth. For instance, whether f is Hamiltonian or not does not affect the calculations

The implementation uses three classes `BseriesRule`, `VectorfieldRule` and `ForestRule` whose instances are callable. That means the objects can be used as functions, in this case with elements of H as input.

The two classes `BseriesRule` and `VectorfieldRule` can be initialized with a function $T \to \mathbb{R}$. Instances of `BseriesRule` extends the domain of the given function to H in accordance with the rules for characters (equation (2.29)). This means the resulting object will accept `UnorderedTrees`, `Forests` and `LinearConbinations` of trees and forests as input.

Alternatively `BseriesRule` and `VectorfieldRule` can be initiated with a `LinearCombination` object containing trees. The coefficient of a particular tree in the linear combination is then treated as value of the rule at that tree. The extension to forests and linear combinations is still the same.

Last, but not least, if `BseriesRule` or `VectorfieldRule` are initiated with no argument, the result is the rule returning zero for any input.

A trivial example of the above is

```
tree = UnorderedTree()
theRule = BseriesRule()
coefficient_of_tree = theRule(tree)
```

where `coefficient_of_tree` will be 0.

The class `ForestRule` works much like the two others, but the supplied function must accept forests. This kind of rule is necessary in the internals of the exp and log functions 2.4.5.

4.3.1 Some B-Series

For convenience and testing purposes, some B-series rules have been implemented. In addition, the next subsection describes how the implementation derives the B-series of an arbitrary RK method from its Butcher tableau.

In addition to the zero-series one gets by not providing anything to the constructor of the above classes, the special B-series the implementation knows are:

- The B-series of the exact solution, $a(\tau) = \frac{1}{\gamma(\tau)}$, as `exponential`. This is the unit element of the Butcher group.
- The B-series $a(\tau) = \begin{cases} 1 & \text{if } \tau = \emptyset \\ 0 & \text{else} \end{cases}$ as `'unit'`.
- The B-series $a(\tau) = \begin{cases} 1 & \text{if } \tau = \bullet \\ 0 & \text{else} \end{cases}$ as `unit_field`. This is the modified equation of the exact solution.
- The B-series of the average vector field method described in 2.6.3 as `AVF`.

4.4 The B-Series of a RK Method

The class `RK_method` is included in the implementation in order to be able to work with RK methods. Its instances stores a Butcher tableau as a matrix `A` and a vector `b`. The method phi, whose implementation is cited below, returns a BseriesRule-object based on the formulas in equation (2.55).

The implementation is a fairly straight forward application of the formulas, except for the explicit use of multiplicities to reduce the number of computations. The implementation uses Numpy [14], abbreviated to `np`, both for the matrix format and for some of the operations.

```
def phi(self):
    def rule(tree):
        if tree == empty_tree:
            return 1
        return np.dot(self.b, self.g_vector(tree))[0]
    return BseriesRule(rule)

def g_vector(self, tree):
    g_vector = np.ones((self._s, 1), dtype=object)
  def u_vector((subtree, multiplicity)):
        return np.dot(self.A,
                        self.g_vector(subtree)) ** multiplicity
    return reduce(operator.__mul__, map(u_vector, tree.items()), g_vector)
```

4.5 The Butcher Group

This and the following section describe some operations on B-series. They are implemented as functions with one or more B-series rules as arguments and they return a new B-series rule. When the new B-series rule is evaluated on a tree, the result is found by calls to the underlying rule(s). This has the advantage that one must not decide in advance for what trees the new rule shall be evaluated. It also means that the construction of the new rule, e.g. the composition, is fast, while the evaluation on trees of moderate size can be computationally expensive.

4.5.1 Composition

The implementation of composition of two B-series rules, a and b, is based on equation (2.30). Since the coproduct is already described, the implementation is straightforward.

```
def composition(a, b):
    def new\_rule(arg):
        result = 0
        for pair, multiplicity in subtrees(arg).items():
            result += a(pair[0]) * b(pair[1]) * multiplicity
        return result
    if a(empty\_tree) != 1:
        return ForestRule(new\_rule)
    else:
        return BseriesRule(new\_rule)
```

The if-else statement is the result of the fact that the composition of non-consistent methods is necessary in the exponential and logarithm described later.

Special Case

The especially simple formula for the case $b(\tau) = \begin{cases} 1 & \text{if } \tau = \bullet \\ 0 & \text{else} \end{cases}$ is available as

```
def hf_composition(a):
    if a(empty_tree) != 1:
        raise ValueError('...')
    def new_rule(tree):
        if tree == empty_tree:
            return 0
        else:
            result = 1
            for subtree, multiplicity in tree.items():
                result *= a(subtree) ** multiplicity
            return result
    return BseriesRule(new_rule)
```

4.5.2 Inverse

The implementation of the inverse of a consistent B-series rule is based on equation (2.31). The implementation is a straightforward application of the antipode:

```
def inverse(a):
    def new_rule(tree):
        return a(antipode_ck(tree))
    return BseriesRule(new_rule)
```

4.5.3 Step Size Adjustment

Although the step size h is not specified, it is sometimes necessary to change it. This is typically needed when several methods are composed to one new method. If nothing is done, the step size of the new method will be a multiple of the nominal step size.

By inspecting definition 23 it is easy to see that multiplying h by A corresponds to substituting $a(\tau)$ by $A^{|\tau|}a(\tau)$. The following function does that.

```
def stepsize_adjustment(a, A):
    base_rule = a._call
    def new_rule(tree):
        return A**tree.order() * base_rule(tree)
    return BseriesRule(new_rule)
```

When combining two methods to form a new one, say explicit and implicit Euler to get the implicit midpoint rule, it is more efficient to halve the step size after the composition than doing it twice before. The following function combines composition and step size halving for convenience.

```
def composition_ssa(a, b):
    if a(empty_tree) != 1:
        raise ValueError('...')
    def new_rule(tree):
        sub_trees = subtrees(tree)
        result = 0
        for pair, multiplicity in sub_trees.items():
            result += a(pair[0]) * b(pair[1]) * multiplicity
        return Fraction(result, 2**tree.order())
    return BseriesRule(new_rule)
```

4.5.4 Adjoint

As described in definition 20 the adjoint is the inverse with reversed time step. By looking at definition 23 again, and let $h \to -h$ it is clear that time reversal corresponds to reversing the sign of B-series coefficients for trees of odd order. This is is what the following code excerpt does.

```
def adjoint(a):
    def new_rule(tree):
        return (-1)**tree.order() * inverse(a)(tree)
    return BseriesRule(new_rule)
```

4.5.5 Conjugate

With composition and inversion of B-series, the conjugate $c^{-1}ac$ is easily implemented as:

```
def conjugate(a, c):
    return composition(inverse(c), composition(a, c))
```

4.6 Series Commutator

As described in section 2.4.6 the the commutator for elements in $\mathcal{T}$ can be extended to B-series. The clue is to observe that the coefficient in front of a given tree in the result of the commutator only depends on the coefficients in front of trees of lower in the arguments. In particular, to find the coefficients of the trees of order n one only needs to consider the grafting of trees such that the sum of the orders of the arguments is n. On the other hand side, it is impossible to choose trees such that when they are grafted a given tree is in the result. The solution is to calculate the coefficients for all trees of a given order at once and store the result. This is the purpose of the `new_rule.order` and `new_rule.storage` variables in the code below.

```
def series_commutator(a, b):
    def new_rule(tree):
        order = tree.order()
        if order in new_rule.orders:
            return new_rule.storage[tree] * tree.symmetry()
        else:
            result = LinearCombination()
            for tree1, tree2 in tree_tuples_of_order(order):
                result += \
                    Fraction(a(tree1) * b(tree2),
                             tree1.symmetry() * tree2.symmetry()) * \
                    tree_commutator(tree1, tree2)
            new_rule.orders.add(order)
            new_rule.storage += result
            return new_rule.storage[tree] * tree.symmetry()
    new_rule.storage = LinearCombination()
    new_rule.orders = set((0,))
    return BseriesRule(new_rule)
```

In the above code excerpt the for-loop loops over the generator[3] `tree_tuples_of_order(order)`. It loops through all the possible (ordered) tuples of trees such that the sum of their orders is a given number, as shown below. It should not be confused with the function `tree_pairs_of_order()`, which loops over all possible *unordered* pairs of trees whose order sum up to a given number.

```
def tree_tuples_of_order(order):
    for order1 in range(1, order):
        order2 = order - order1
        for tree1 in trees_of_order(order1):
            for tree2 in trees_of_order(order2):
                yield (tree1, tree2)
```

4.6.1 The Conjugate by Series Commutator

As described in section 2.4.6 the conjugate $c^{-1}ac$ can be evaluated for a given tree by a finite series of commutators. This is implemented in the below code excerpt:

[3]In Python a *generator* is like a list that can be iterated through, but whose elements are not actually stored. Instead they are created when they are needed.

```
def conjugate_by_commutator(a, c):
    def new_rule(tree):
        if tree == empty_tree:
            return 0
        tmp = a
        result = 0
        for n in range(tree.order() + 1):
            result += Fraction((-1)**n, factorial(n)) * tmp(tree)
            tmp = series_commutator(c, tmp)
        return result
    return BseriesRule(new_rule)
```

4.7 The Lie Derivative

Using the `split()`-function described in section 4.2.5, the implementation of a function returning the rule corresponding to equation (2.34) is:

```
def lie_derivative(c, b, truncate=False):
    if b(empty_tree) != 0:
        raise ValueError('...')
    def new_rule(tree):
        result = 0
        if tree == empty_tree:
            return result
        pairs = split(tree, truncate)
        for pair, multiplicity in pairs.items():
            result += multiplicity * b(pair[0]) * c(pair[1])
        return result
    return VectorfieldRule(new_rule)
```

The boolean argument `truncate` is necessary when the Lie derivative is used for the modified equation and is explained in the next section.

4.8 The Modified Equation

As pointed out in the theory chapter, the equation for the B-series of the modified equation,

$$b(\tau) = a(\tau) - \sum_{j=2}^{|\tau|} \frac{1}{j!} \partial_b^{j-1} b(\tau), \tag{4.1}$$

seems to be self referring, but does allow a recursive evaluation. It is however very close to self referring, and a naive implementation will run into infinite recursion. The problem arises when $b(\emptyset) \cdot b(\tau)$ is evaluated by the Lie derivative. The fact that $b(\emptyset) = 0$ resolves the problem as it causes the above product to always be zero. In the implementation reproduced below the issue is addressed by passing `truncate=True` to the `split()`-function which causes it to skip the problematic pair $\emptyset \otimes \tau$.

```
def modified_equation(a, quadratic_vectorfield=False):
    if a(empty_tree) != 1 or a(leaf) != 1:
        raise ValueError('...')
    def new_rule(tree):
        if tree == empty_tree:
            return 0
        result = a(tree)
        c = new_rule
        for j in range(2, tree.order() + 1):
            c = lie_derivative(c, new_rule, True)
            result -= Fraction(c(tree), factorial(j))
        return result
    result = VectorfieldRule(new_rule)
    if quadratic_vectorfield:
        result = remove_non_binary(result)
    return result
```

The if-statement at the end is necessary to keep the property that only terms with binary trees are non-zero when the right hand side of the ODE is quadratic. The function `remove_non_binary` returns a new rule where the coefficients of all non-binary trees are set to zero.

4.8.1 The Logarithm and Exponential Maps

As pointed out in the theory chapter, the modified equation can also be calculated with the logarithm map. The code excerpt below implements equation (2.38) in a straightforward way. This method of finding the modified equation runs much slower than the one presented above.

```
def log(a, quadratic_vectorfield=False):
    if a(empty_tree) != 1:
        raise ValueError('...')
    def new_rule(tree):
        if tree == empty_tree:
            return 0
        a_2 = remove_empty_tree(a)
        result = a_2(tree)
        b = a_2
        for n in range(2, tree.order() + 1):
            b = composition(b, a_2)
            result += ((-1)**(n+1)) * Fraction(b(tree), n)
        return result
    result = VectorfieldRule(new_rule)
    if quadratic_vectorfield:
        result = remove_non_binary(result)
    return result
```

The `remove_empty_tree`-function returns a rule with 0 as the coefficient of the empty tree.

The exponential function, the inverse of the logarithm, is implemented in line with equation (2.40) in the theory chapter.

```
def exp(a, quadratic_vectorfield=False):
    if a(empty_tree) != 0:
        raise ValueError('...')
    def new_rule(tree):
        if tree == empty_tree:
            return 1
        result = a(tree)
        b = a
        for n in range(2, tree.order() + 1):
            b = composition(b, a)
            result += Fraction(b(tree), factorial(n))
        return result
    result = BseriesRule(new_rule)
    if quadratic_vectorfield:
        result = remove_non_binary(result)
    return result
```

The if-statements at the end serve the same purpose as above.

4.9 Investigating Properties of B-series

The tests in this section all follow a general pattern of checking whether some property holds for the coefficients in a B-series for trees of increasing order, one order at at time. The goal is to identify the first order for which the property does not hold. In many cases an integer variable `max_order` can be given to ensure that the test terminates in a reasonable amount of time even if the property holds for all trees or to some very high order.

4.9.1 Order of Convergence

Finding the order of convergence is the simplest test since it amounts to comparing the B-series of the numerical flow to that of the exact solution, or in Python code:

```
def convergence_order(a):
    return equal_up_to_order(a, exponential)
```

where `equal_up_to_order` is what one would expect it to be:

```
def equal_up_to_order(a, b, max_order=None):
    if not a(empty_tree) == b(empty_tree):
        return None
    for tree in tree_generator():
        if max_order and tree.order() > max_order:
            return max_order
        elif not a(tree) == b(tree):
            return tree.order() - 1
```

It would also be possible to test the convergence order by comparing the modified equation to the B-series called `unit_field` (from section 4.3.1). However, one usually starts with the B-series of the numerical flow of a method, and finding the modified equation is more work than computing $\gamma(\tau)$.

4.9.2 Symmetry

With the implementation of the adjoint from section 4.5.4, symmetry is checked in much the same way as order of convergence:

```
def symmetric_up_to_order(a, max_order=None):
    b = adjoint(a)
    return equal_up_to_order(a, b, max_order)
```

4.9.3 Symplectic and Hamiltonian

Determining to which order a numerical method is symplectic or, equivalently, to which order its vector field is Hamiltonian, is more involved. Both tests involve checking a condition for all unordered pairs of trees τ_1, τ_2 such that $|\tau_1| + |\tau_2|$ equals the order to which one wants to check whether the property is satisfied. This is done in the following way.

```
def _satisfied_for_tree_pairs_of_order(condition, order):
    max_check_order = order / 2  # Intentional truncation in division.
    for order1 in range(1, max_check_order + 1):
        order2 = order - order1
        for tree1 in trees_of_order(order1):
            for tree2 in trees_of_order(order2):
                if not condition(tree1, tree2):
                    return False
    return True
```

With this in place checking whether a flow is symplectic is done by

```
def symplectic_up_to_order(a, max_order=None):
    if a(empty_tree) != 1:
        return None
    orders = count(start=2)
    if max_order:
        orders = islice(orders, max_order - 1)
    _symp_cond = partial(_symplecticity_condition, a)
    for order in orders:
        if not _satisfied_for_tree_pairs_of_order(_symp_cond, order):
            return order - 1
    return max_order
```

where `_symplecticity_condition()` is

```
def _symplecticity_condition(a, tree1, tree2):
    return a(tree1.butcher_product(tree2)) + a(tree2.butcher_product(tree1)) \
        == a(tree1) * a(tree2)
```

The implementation for determining whether a vector field is Hamiltonian is analogous:

```
def hamiltonian_up_to_order(a, max_order=None):
    if a(empty_tree) != 0 or a(leaf) == 0:
        return None
    orders = count(start=2)
    if max_order:
        orders = islice(orders, max_order - 1)
    _ham_cond = partial(_hamilton_condition, a)
    for order in orders:
        if not _satisfied_for_tree_pairs_of_order(_ham_cond, order):
            return order - 1
    return max_order
```

with

```
def _hamilton_condition(a, tree1, tree2):
    return a(tree1.butcher_product(tree2)) + \
        a(tree2.butcher_product(tree1)) == 0
```

4.9.4 Properties as Subspaces

The remaining properties are tested for by asserting whether or not the B-series of the modified equation is in a certain subspace, as described in 2.5.3. The tests are also performed order by order. For each order an ordering of the trees is chosen in order to represent a part of the B-series as an array. The choice is arbitrary, but the implementation uses definition 6.

Then a matrix A, whose columns form a basis of the subspace in question, is constructed. Since the subspace is necessarily a subspace of T^n, A is taller than it is wide. The coefficients of the B-series under consideration are stored in a vector b and the problem

$$Ax = b \tag{4.2}$$

is solved with least squares solver `lsqr()` from `scipy.sparse.linalg`.

The following code excerpt shows how the result of the least square algorithm is used to deduce whether or not b is in the colspan of A.

```python
def not_in_colspan(A, b):
    try:
        result = lsqr(A, b)
    except ZeroDivisionError:
        return False  # Happens if iteration hits the exact solution.
    return result[1] != 1 and (result[1] == 2 or result[3] > 10.0**(-10))
```

The return value from `lsqr()` is a tuple. Its value at 1 is an integer revealing why the algorithm terminated, while the value at 3 is the norm of the residual vector. More important is the fact that this algorithm is done in floating point arithmetic, while all other computations in the library are done in exact arithmetic.

The next subsections goes through how this is done when determining to what order the modified equation is Hamiltonian and energy preserving. The important difference between the two is the matrix A.

4.9.5 Hamiltonian as Subspace

The basis for the Hamiltonian sub space of a given order is constructed using free trees as described in section 2.5.3. The rather straightforward algorithm is placed in an own function:

```python
def hamiltonian_matrix(order):
    nsft = the_trees[order].non_superfluous_trees(sort=True)
    m = number_of_trees_of_order(order)
    n = len(nsft)
    result = sparse.lil_matrix((m, n), dtype=np.int8)
    for free_tree in nsft:
        j = the_trees[order].non_superfluous_index(free_tree)
        for tree, sign in free_tree._rooted_trees.items():
            i = the_trees[order].index(tree)
            result[i, j] = sign
    return result
```

The rest of the implementation is given below. There are a few shortcuts which are discussed below the code excerpt.

```python
def subspace_hamiltonian_up_to_order(a, max_order=None):
    if a(empty_tree) != 0 or a(leaf) == 0:
        return None
    orders = count(start=2)
    if max_order:
        orders = islice(orders, max_order - 1)
    for order in orders:
        b = np.asarray(map(a, trees_of_order(order, sort=True)),
                            dtype=np.float64)
        if not np.any(b):
            continue
        if order == 2:
            return 1
        A = hamiltonian_matrix(order)
        if not_in_colspan(A, b):
            return order - 1
    return max_order
```

The two first if-clauses in the for-loop test for some simple cases that does not need to be resolved by `not_in_colspan()`. The first one checks if all the coefficients of that order are zero, and if they are the rest of that loop-iteration is skipped. This can be done since the zero vector is in every subspace.

The second if-clause uses the fact that the tree of order two never appears in a Hamiltonian B-series.

4.9.6 Energy Preservation

Energy preservation is checked in a slightly different way. The trees that can have any coefficient is simply ignored. Before any intricate computations, the coefficient in front of the trees that never appears in energy preserving B-series are checked. If this check is passed, the function goes on to check independently for each of the free trees of order $n+1$. This last part has the advantage of solving several small least squares problems instead of one large problem.

For each order the implementation does the following:

```
def _is_energy_preserving_of_order(a, order):
    forbidden_trees, interesting_trees = _get_tree_sets(order)
    for tree in forbidden_trees:
        if a(tree) != 0:
            return False
    for free_tree, collection in interesting_trees.items():
        collection = sorted(collection)
        A = get_energy_matrix(free_tree, collection)
        b = np.asarray(map(a, collection), dtype=np.float64)
        if not_in_colspan(A, b):
            return False
    return True
```

where `_get_tree_sets` is

```
def _get_tree_sets(order):
    forbidden_trees = set()  # Never found in energy preserving series.
    interesting_trees = dict()  # included in a non trivial basis vector.
    for tree in the_trees[order].trees():
        free_tree = leaf.butcher_product(tree).get_free_tree()
        if free_tree.superfluous:
            pass  # Don't store them
        elif free_tree.is_symmetric():
            forbidden_trees.add(tree)
        elif free_tree in interesting_trees:
            interesting_trees[free_tree].add(tree)
        else:
            interesting_trees[free_tree] = set((tree,))
    return forbidden_trees, interesting_trees
```

For each free tree the matrix is constructed as suggested by the last paragraph of the section on energy conservation in the theory chapter:

```
def get_energy_matrix(free_tree, collection):
    le = len(collection)
    A = sparse.lil_matrix((le, le-1), dtype=np.int64)
    A[0, :] = Fraction(-collection[0].symmetry(),
                       free_tree._rooted_trees[
                       leaf.butcher_product(collection[0])])
    for tree in collection[1:]:
        i = collection.index(tree)
        A[i, i-1] = Fraction(tree.symmetry(),
                             free_tree._rooted_trees[
                             leaf.butcher_product(tree)])
    return A
```

Note how the height of the matrix is only the number of rooted trees corresponding to the free tree, resulting in a relatively small least squares problem.

The function tying the above together is similar to `subspace_hamiltonian_up_to_order()`:

```
def energy_preserving_upto_order(a, max_order=None):
    if a(empty_tree) != 0 or a(leaf) != 1:
        return None
    orders = count(start=2)
    if max_order:
        orders = islice(orders, max_order - 1)
    for order in orders:
        if not _is_energy_preserving_of_order(a, order):
            return order - 1
    return max_order
```

4.9.7 Conjugate to Symplectic

The implementation to check whether a method is conjugate to symplectic up to at most twice its convergence order uses the conditions in section 2.5.4. The system of linear equations is set up as a rectangular matrix by the function `_conjugate_symplecticity_matrix`.

```
def _conjugate_symplecticity_matrix(order):
    A = []
    list_of_pairs1 = tree_pairs_of_order(order, sort=True)
    list_of_pairs2 = tree_pairs_of_order(order-1, sort=True)
    for pair in list_of_pairs1:
        tmp = [0] * m(order-1)
        for tree, multiplicity in symp_split(pair[0]).items():
            try:
                the_number = list_of_pairs2.index((tree, pair[1]))
            except ValueError:
                the_number = list_of_pairs2.index((pair[1], tree))
            tmp[the_number] += multiplicity

        for tree, multiplicity in symp_split(pair[1]).items():
            try:
                the_number = list_of_pairs2.index((pair[0], tree))
                except ValueError:
                    the_number = list_of_pairs2.index((tree, pair[0]))
                tmp[the_number] += multiplicity
            A.append(tmp)
    A = np.asarray(A)
    return A
```

The function `tree_pairs_of_order()` returns a list of tuples of trees, but in such a way that each unordered pair only appears once.

The following function iterates through orders and perfoms the check. It takes care to utilize symmetric methods whenever possible, and to stop at $p + r$.

```
def conjugate_to_symplectic(a, max_order=float("inf"),
                            quadratic_vectorfield=False):
    conv_order = convergence_order(a)
    first_order_checked = conv_order + 1 + (conv_order == 2)
    max_order = min(max_order, 2*conv_order)
    orders = xrange(first_order_checked, max_order+1)
    _alpha = modified_equation(a, quadratic_vectorfield)
    def alpha(u, v):
        return _alpha(u.butcher_product(v)) - _alpha(v.butcher_product(u))
    for order in orders:
        if symmetric_up_to_order(a, order) == order and order % 2 == 0:
            continue
        A = _conjugate_symplecticity_matrix(order)
        b = np.asarray(
            [alpha(u, v) for u, v in tree_pairs_of_order(order, sort=True)],
            dtype=np.float64)
        if not_in_colspan(A, b):
            return order - 1
    return max_order
```

4.10 Quadratic Vector Fields

As pointed out in section 2.4.7, quadratic right hand sides of equation (2.15) ensures that certain trees are absent in the B-series. Since this is a result of the elementary differentials disappearing, rather than the coefficients of the method, the property is not automatically respected when going back and forth between `BseriesRules` and `VectorfieldRules`. Instead `modified_equation()`, `log()` and `exp()` takes an optional boolean arguments, `quadratic`, and ensures the coefficients of all non-binary trees are set to zero with the help of the following code when necessary.

```
def remove_non_binary(a):
    base_rule = a._call
    def new_rule(tree):
        if tree == empty_tree or tree.is_binary():
            return base_rule(tree)
        else:
            return 0
    return type(a)(new_rule)
```

4.11 Printing Trees

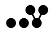
This section does not concern mathematics, but the issue of printing trees in a human readable way. This includes both a reasonable string representation that can be output on the command line, as well as producing drawings of trees in a graphical user interface.

4.11.1 String Representation

Although the intended output from the library is information on to what extent a given method has certain properties, being able to print trees as string is invaluable in any attempt to track the actions of the program for debugging purposes. To this end a simple scheme for representing trees, forests and linear combinations as strings has been implemented.

- `UnorderedTree`s are printed as nested multisets using square brackets. For example, the one node tree, •, is `'[]'` and is represented as `'[[],[]]'`. The child trees are printed in no particular order. In addition the constructor of `UnorderedTree` will accept and parse strings on this format.

- A free trees is printed as its rooted tree representative.

- A forest is printed as comma separated list of its trees, with multiplicities as exponentials. For example could  be printed as `'([]^2, [[],[]]^1)'`.

- A linear combination is printed with the factor in front of the term, then a '`*`', then the term itself (a tree or a forest). The terms are separated by `+`-signs. For example would $2 \cdot$ $+$ • be printed `'2*[[]] + 1*[]'`.

The implementations of all the above are straightforward. The string representation of a Python object, say a tree called `t`, is obtained by calling the `str(t)`, or it can be printed directly with `print t`.

`BseriesRules`, `VectorfieldRules` and `ForestRules` do not have any meaningful string representation. The main reason for this is that they are initiated from an arbitrary Python-function supposed to return real numbers for any tree, and there is no obvious way of representing this as a string. A possibility would have been to allow the initialization to accept an explanatory string, and make sure composition, the logarithm, the exponential, the Lie derivative and all the simpler operations on B-series rules created appropriate strings. However this would have been a considerable amount of work.

4.11.2 Drawings in IPython

For `UnorderedTree`-objects, and only them, a method `_repr_svg_()` has been implemented. It allows IPython to print trees as drawings, much like the way trees are drawn in this report. The method uses the planarforest [18] package to cretate a tikz-representation of the tree, then tikz2svg is used to convert it to an svg-image that can be displayed by the IPyhton notebook.

Chapter 5

Conclusion

This chapter reviews some of the decisions made and their effect on the product, the PyBS library. In doing this it is natural to compare the current implementation to the ones previously mentioned, as well as the library's aptitude for investigating B-series methods. Suggestions for further work is included at the end.

Programming Language and Strategy

The principal choice in regards of what programming language to use was between Python as a general purpose programming language and a computer algebra system, Sage in particular. Other alternatives include Maple and Mathematica. The choice in favor of Python was made mainly due to the lack of experience (or support from someone with experience) with implementing new algebraic structures in Sage.

As the work progressed it became clear that a CAS's ability to deal with common mathematical concepts could have saved some work and ensured a sound and rigorous implementation. An example is the class `LinearCombination`.

That being said, the use of Python did not cause any major problems. The practice of writing and running tests allowed for timely discovery of bugs. It also gave the opportunity to make changes to existing code and with high probability discover all unfavorable consequences.

Tree Structure

The way trees are represented is one of the choices with the most far-reaching consequences. It is thus interesting to note how PyBS and the three other implementations mentioned in section 1.3 does this.

Nodepy represents trees as strings in a format not unlike the way PyBS prints trees (see section 4.11.1). The main difference is that curly brackets are used and that if a vertex has leaves, they are written as T^n at the beginning of the list of child trees. The advantage of this is that it is immediately readable to a human. The disadvantage is that any function or operation on trees must do a fair amount of parsing.

The Maple script provided by Owren stores trees as nested lists of lists. This necessitates the use of an arbitrary but consistent sorting of these lists each time a tree is created or modified. This approach has the disadvantage that multiplicities must be recovered by counting.

The Mathematica scripts provided by Murua stores trees as the standard decomposition described in definition 7. This is memory efficient, and since the standard decomposition is unique, comparing for equality is simple. The main objection to this approach is probably that parsing the trees results in much deeper recursions than the other methods described here. This might have an impact on running times, since the overhead of recursive function calls are known to affect running times.

One notable advantage of the approach used in PyBS over the above is the way it facilitates the use of multiplicities explicitly in the calculations. This encourages faster algorithms looping over distinct child trees instead of all child trees with repetitions. On the other hand side, the use of dictionaries (hash tables) adds an unnecessary layer of abstraction. In retrospect the current author believes a modified edition of Owren's approach might be simpler than the one used in PyBS. A tree could store a sorted list of tuples of its distinct child trees and their multiplicity. If the list is sorted based on the distinct child trees ordering according to some definition, lookups can be done with efficient binary searches.

B-series as Functions

Since B-series are infinite, it is impossible to calculate all coefficients and store them in memory. In PyBS this is addressed by representing the B-series rules as functions and let operations on series produce new functions depending on the original series whenever they are evaluated.

Another approach, used in the implementations by Owren and Murua, is to decide the size of the largest tree needed before any calculations and analyze what is effectively truncated series. At least in principle relieving the user of this responsibility has some advantage and few disadvantages. However, in practice the user will often know, or be able to make an educated guess of, the larges tree needed.

Suggestions for Further Work

The PyBS library is focused on functionality needed to analyze B-series methods. Due to this focus, some desirable operations on trees have been ignored. This includes the Grossmann-Larson product and grafting a forest onto a tree. The latter is somewhat involved since once a tree has been grafted onto the target, the remaining trees in the forest may not be grafted on top of it.

Another area in which much work can be done is the use of simplifying assumptions such as quadratic vector fields and one dimensional ODEs. The current practice of forcing the coefficients of non-binary trees to be zero is not optimal. A better solution might be to modify the set of trees returned by the iterators supplying trees to the functions testing for properties. An idea to create instances only supplying e.g. binary trees was some of the motivation for creating the `Trees`-class (in addition to handle free trees).

Other possible expansions are to include checks for conjugate to energy preserving and effective order, to implement the Hopf algebra by Calaque, Ebrahim-Fard and Manchon, or calculating modifying integrators as B-series. It is also perceivable to implement colored and ordered trees, but that would require much of the current implementation to be modified.

Bibliography

[1] V. I. Arnold. *Mathematical Methods of Classical Mechanics.* Graduate texts in mathematics. Springer-Verlag New York Inc., 1989.

[2] Geir Bogfjellmo. "Algebraic structure of aromatic B-series." Version 1. 2015. URL: `http://arxiv.org/pdf/1505.01973.pdf`.

[3] Marthe Bonamy. "A Small Report on Graph and Tree Isomorphism." Lecture note. 2010. URL: `http://perso.ens-lyon.fr/eric.thierry/Graphes2010/marthe-bonamy.pdf`.

[4] John Butcher. *Numerical Methods for Ordinary Differential Equations; 2nd ed.* Chichester: Wiley, 2008.

[5] Elena Celledoni, Robert Ian McLachlan, Brynjulf Owren, and G.R.W. Quispel. "Energy-Preserving Integrators and the Structure of B-series." In: *Foundations of Computational Mathematics* 10 (2010), pp. 673–693.

[6] Elena Celledoni, Robert Ian McLachlan, David I. McLaren, Brynjulf Owren, G. Reinout W. Quispel, and William M. Wright. "Energy-Preserving Runge-Kutta Methods." In: *ESAIM: Mathematical Modelling and Numerical Analysis* 43 (2009), pp. 645–649.

[7] Elena Celledoni, Robert Ian McLachlan, Brynjulf Owren, and G. Reinout W. Quispel. "Geometric properties of Kahan's method." In: *Journal of Physics A: Mathematical and Theoretical* 46 (2013).

[8] Frédéric Chapoton and Muriel Livernet. "Pre-Lie Algebras and the Rooted Trees Operad." In: *International Mathematics Research Notices* 8 (2001), pp. 395–408.

[9] Philippe Chartier, Ernst Hairer, and Gilles Vilmart. "Algebraic structures of B-series." In: *Foundations of Computational Mathematics* 10 (2010), pp. 407–427.

[10] Alain Connes and Dirk Kreimer. "Hopf Algebras, Renormalization and Noncommutative Geometry." In: *Communications in Mathematical Physics* 199 (1998), pp. 203–242.

[11] *Crash Course on Flows.* Notes. URL: `http://www.math.toronto.edu/karshon/grad/2010-11/flows.pdf`.

[12] Ernst Hairer, Christian Lubich, and Gerhard Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations.* Vol. 31. Springer Series in Computational Mathematics. Berlin: Springer, 2006.

[13] Ernst Hairer and Christophe J. Zbinden. "On conjugate symplecticity of B-series integrators." In: *IMA Journal of Numerical Analysis* 33 (2013), pp. 57–79.

[14] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python.* 2001–. URL: `http://www.scipy.org/`.

[15] David Ketcheson. *NodePy: Numerical ODEs in Python.* `http://numerics.kaust.edu.sa/nodepy/`. 2014.

[16] Pierre Leone. "Symplecticity and symmetry of general integration methods." PhD thesis. University of Geneva, 2000.

[17] Alexander Lundervold and Hans Z. Munthe-Kaas. "On algebraic structures of numerical integration on vector spaces and manifolds." 2011.

[18] Håkon Marthinsen. *Planarforest*. `http://hmarthinsen.github.io/planarforest/`. 2014.

[19] Ander Murua. “Formal series and numerical integrators, Part I: Systems of ODEs and symplectic integrators.” In: *Applied Numerical Mathematics* 29 (1999), pp. 221–251.

[20] Ander Murua. “From Runge-Kutta methods to Hopf algebras of rooted trees.” Note prepared for COCO2010. 2012.

[21] Ander Murua. “The Hopf algebra of rooted trees, free Lie algebras, and Lie series.” In: *Foundations of Computational Mathematics* 6 (2006), pp. 387–426.

[22] Neil James Alexander Sloane. *The On-Line Encyclopedia of Integer Sequences, sequence A000081*. [Referred algorithm apparently due to Joe Riel]. 2014. URL: `https://oeis.org/A000081`.

[23] William Arthur Stein et al. *Sage Mathematics Software (Version 6.4.1)*. `http://www.sagemath.org`. The Sage Development Team. 2014.

[24] Henrik Sperre Sundklakk. *A Library for Computing with Trees and B-Series*. Specialization Project, TMA4500. 2014.

[25] Moss Eisenberg Sweedler. *Hopf Algebras*. W. A. Benjamin, inc., 1969.

[26] Wikipedia. *Arborescence (graph theory) — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-Apr.-2015]. 2015. URL: `http://en.wikipedia.org/wiki/Arborescence_(graph_theory)`.