Fall 2019

Computational Analysis of Antipode Algorithms for the Output Feedback Hopf Algebra

Lance Berlin
Old Dominion University, lberl001@odu.edu

COMPUTATIONAL ANALYSIS OF ANTIPODE ALGORITHMS FOR THE OUTPUT FEEDBACK HOPF ALGEBRA

by

Lance Berlin
Bachelor's of Science in Engineering December 2013, Old Dominion University

A Thesis Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

ELECTRICAL AND COMPUTER ENGINEERING

OLD DOMINION UNIVERSITY
December 2019

Approved by:

W. Steven Gray (Director)

Luis A. Duffaut Espinosa (Member)

Oscar R. González (Member)

Dimitrie C. Popescu (Member)

ABSTRACT

COMPUTATIONAL ANALYSIS OF ANTIPODE ALGORITHMS FOR THE OUTPUT FEEDBACK HOPF ALGEBRA

Lance Berlin
Old Dominion University, 2019
Director: Dr. W. Steven Gray

The feedback interconnection of two systems written in terms of Chen-Fliess series can be described explicitly in terms of the antipode of the output feedback Hopf algebra. At present, there are three known computational approaches to calculating this antipode: the left coproduct method, the right coproduct method, and the derivation method. Each of these algorithms is defined recursively, and thus becomes computationally expensive quite quickly. This motivates the need for a more complete understanding of the algorithmic complexity of these methods, as well as the development of new approaches for determining the Hopf algebra antipode. The main goals of this thesis are to create an implementation in code of the derivation method and compare the computational performance against existing code for the two coproduct methods in *Mathematica*. Both temporal and spatial complexity are examined empirically, and the main conclusion is that the derivation method yields the best performance.

Dedicated to my family.
You were there through the most difficult time.
I hope one day I can have as great an impact on others as you have had on me.

ACKNOWLEDGEMENTS

The author would first like to thank Dr. W. Steven Gray for his guidance and assistance throughout both the research and writing processes. Without his mentoring, this achievement would not have been possible. Dr. Gray was supported by the National Science Foundation under grant CMMI 1839378.

Next the author recognizes the members of the thesis committee, Dr. Luis Duffaut Espinosa (Department of Electrical and Biomedical Engineering, The University of Vermont), Dr. Oscar González (Department of Electrical and Computer Engineering, Old Dominion University) and Dr. Dimitrie Popescu (Department of Electrical and Computer Engineering, Old Dominion University). Their time and input on this body of work were essential contributions. Dr. Duffaut Espinosa by the National Science Foundation under grant CMMI 1839387.

A special thanks is given to Dr. Kurusch Ebrahimi-Fard (Department of Mathematical Sciences, Norwegian University of Science and Technology, Norway) and the Instituto de Ciencias Matemáticas, Consejo Superior de Investigaciones Científicas, in Madrid for hosting the author during a series of visits. These occasions enabled much of the research appearing in this thesis. Funding for these visits was graciously provided by the BBVA Foundation.

Another important figure in this work is Dr. J. William Helton (University of California San Diego). Besides being an author of the NCAlgebra package that serves as a foundation for the NCFPS software utilized in this thesis, he also made notable code contributions to NCFPS itself. The work of Dr. Helton and his group was supported by National Science Foundation grants DMS 0757212 and DMS 0700758.

The author would also like to acknowledge the support rendered by the staff of the Electrical and Computer Engineering Department of Old Dominion University, as well as the Department's funding provided towards the author's Master of Science degree.

Finally, the support and reinforcement of the author's family cannot go unmentioned. Their presence during times of hardship enabled this amongst many other achievements, and they cannot be thanked enough.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

(c, η)	The coefficient of η in the series c
@	Feedback product
$\emptyset$	Empty word
$\|\eta\|$	Length of the word η
$\mathbb{R}^{\ell}\langle\langle X\rangle\rangle$	Set of formal power series with coefficients in $\mathbb{R}^{\ell}$ over X
$\tilde{\circ}$	Modified composition product
$\sqcup\!\sqcup$	Shuffle product
a_{η}^{j}	The j^{th} coordinate function indexed by η
E_{η}	Iterated integral indexed by the word η
F_c	Fliess operator with generating series c
Sa_{η}^{j}	The antipode of a_{η}^{j}
X^*	The set of all words composed of letters in X

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

There are a variety of ways to mathematically represent a nonlinear input-output system. A particularly versatile class of representations is the set of functional series expansions, which includes Volterra, Weiner, and Chen-Fliess series [1–11]. The latter set, also known as Fliess operators, has shown to be particularly useful in control theory [1, 6, 12]. Simply put, a Chen-Fliess series is a weighted sum of iterated integrals taken over the vector of input functions. Such a series can be written uniquely in terms of a formal power series in noncommutative variables. This allows for direct algebraic manipulation. Such a series is effectively a noncommutative Taylor series representation of the input-output map of a system [6]. Chen-Fliess series are normally written as infinite sums indexed by words.

1.1.1 FORMAL POWER SERIES

A set of *letters*, $X = \{x_0, x_1, \ldots, x_m\}$, is referred to as an *alphabet*. From the elements of an alphabet, one can form finite sequences of letters known as *words* over X such as $\eta = x_{i_1} \cdots x_{i_k}$. The letters in X are noncommutative with respect to multiplication, thus words containing the same letters in different orders are considered distinct. The length of η is defined by $|\eta|$, which is simply the number of letters in the word. Let $|\eta|_{x_i}$ indicate the number of times the letter x_i occurs in the word η. The set X^* contains all possible words over X, including the empty word $\emptyset$.

Catenation on X^* is the binary operation $X^* \times X^* \to X^* : (\eta_1, \eta_2) \mapsto \eta_1\eta_2$. The empty word acts as a unit, i.e., $\emptyset\eta = \eta\emptyset = \eta \ \forall \ \eta \in X^*$. Catenation is associative, thus X^* is a monoid under catenation. Mappings from X^* onto $\mathbb{R}^\ell$ are known as *formal power series*. They are often written in terms of a formal sum $c = \sum_{\eta \in X^*} (c, \eta)\eta$, where $(c, \eta) \in \mathbb{R}^\ell$ denotes the coefficient of η. The $\mathbb{R}$-vector space of all such series is written $\mathbb{R}^\ell\langle\langle X \rangle\rangle$. Here addition is defined as

$$c + d = \sum_{\eta \in X^*} ((c, \eta) + (d, \eta))\eta,$$

and scalar multiplication is defined as

$$\alpha c = \sum_{\eta \in X^*} \alpha(c, \eta)\eta.$$

This space also forms an associative $\mathbb{R}$-algebra, where multiplication is the *Cauchy product*

$$cd \mapsto \sum_{\eta_1, \eta_2 \in X^*} (c, \eta_1)(d, \eta_2)\eta_1\eta_2$$

and the product on $\mathbb{R}^\ell\langle\langle X \rangle\rangle$ is defined componentwise. In addition, the *shuffle product* of two words, $\sqcup\!\sqcup$, can be defined recursively by

$$(x_i\eta) \sqcup\!\sqcup (x_j\xi) = x_i(\eta \sqcup\!\sqcup (x_j\xi)) + x_j((x_i\eta) \sqcup\!\sqcup \xi),$$

where $x_i, x_j \in X^*$, $\eta, \xi \in X^*$, and $\eta \sqcup\!\sqcup \emptyset = \emptyset \sqcup\!\sqcup \eta = \eta$ [4]. This definition can be extended linearly to the space $\mathbb{R}^\ell\langle\langle X \rangle\rangle$ to form an associative and commutative $\mathbb{R}$-algebra, the so called *shuffle algebra*.

Example 1.1.1. Given $c = 1 + x_0x_1$ and $d = x_2x_0$, observe

$$\begin{aligned}
c \sqcup\!\sqcup d &= (1 + x_0x_1) \sqcup\!\sqcup x_2x_0 \\
&= 1 \sqcup\!\sqcup x_2x_0 + x_0x_1 \sqcup\!\sqcup x_2x_0 \\
&= x_2x_0 + x_0(x_1 \sqcup\!\sqcup x_2x_0) + x_2(x_0x_1 \sqcup\!\sqcup x_0) \\
&= x_2x_0 + x_0[x_1(1 \sqcup\!\sqcup x_2x_0) + x_2(x_1 \sqcup\!\sqcup x_0)] + x_2[x_0(x_1 \sqcup\!\sqcup x_0) + x_0(x_0x_1 \sqcup\!\sqcup 1)] \\
&= x_2x_0 + x_0[x_1x_2x_0 + x_2x_1x_0] + x_2[x_0x_1x_0 + x_0x_0x_1] \\
&= x_2x_0 + x_0x_1x_2x_0 + x_0x_2x_1x_0 + x_2x_0x_1x_0 + x_2x_0x_0x_1.
\end{aligned}$$

1.1.2 FLIESS OPERATORS

Any series $c \in \mathbb{R}^\ell\langle\langle X \rangle\rangle$ can be associated with a causal m-input, ℓ-output operator, F_c. Let $\mathfrak{p} \geq 1$ and $t_0 < t_1$ be fixed. Given a Lebesgue measurable function $u : [t_0, t_1] \to \mathbb{R}^m$, define it's $\mathfrak{p}$-norm as $||u||_\mathfrak{p} = \max\{||u_i||_\mathfrak{p} : 1 \leq i \leq m\}$ with $||u_i||_\mathfrak{p}$ denoting the usual $L_\mathfrak{p}$-norm for a measurable, real-valued function, u_i, defined on $[t_0, t_1]$. Let $L^m_\mathfrak{p}[t_0, t_1]$ represent the set of all measurable functions defined on $[t_0, t_1]$ which have finite $||\cdot||_\mathfrak{p}$ norm. The closed ball of radius R at the origin in $L^m_\mathfrak{p}[t_0, t_1]$ is denoted by

$$B^m_\mathfrak{p}(R)[t_0, t_1] := \{u \in L^m_\mathfrak{p}[t_0, t_1] : ||u||_\mathfrak{p} \leq R\}.$$

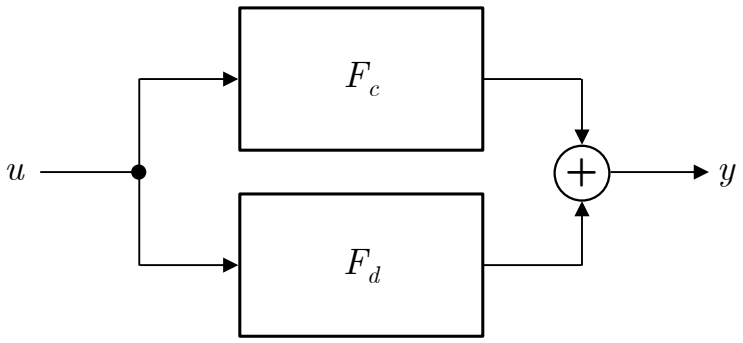


Fig. 1: Parallel sum interconnection.

$C[t_0, t_1]$ is the subset of continuous functions in $L_1^m[t_0, t_1]$. For every $\eta \in X^*$ it is possible to inductively define the mapping $E_\eta : L_1^m[t_0, t_1] \to C[t_0, t_1]$ as

$$E_{x_i\bar{\eta}}[u](t, t_0) = \int_{t_0}^{t} u_i(\tau) E_{\bar{\eta}}[u](\tau, t_0) d\tau$$

with $x_i \in X$, $\bar{\eta} \in X^*$, $u_0 = 1$, and $E_\emptyset[u] = 1$. There then exists a corresponding input-output operator for c, namely the *Fliess operator*

$$F_c[u](t) = \sum_{\eta \in X^*} (c, \eta) E_\eta[u](t, t_0)$$

[4]. One refers to c as the *generating series* of F_c.

Define $|z| := \max_{1 \le i \le \ell} |z_i|$ for $z \in \mathbb{R}^\ell$. If there exists constants $K, M > 0$ such that

$$|(c, \eta)| \le K M^{|\eta|} (|\eta|!)^s, \forall \eta \in X^*,$$

then F_c is said to have *Gevery order* $s \in \mathbb{R}$ [13]. When $s = 1$, F_c is a well-defined mapping from $B_{\mathfrak{p}}^m(R)[t_0, t_0 + T]$ into $B_{\mathfrak{q}}^\ell(S)[t_0, t_0 + T]$ when $R, T > 0$ are sufficiently small and $\mathfrak{p}, \mathfrak{q} \in [1, \infty]$ are conjugate exponents [14]. The set $\mathbb{R}_{LC}^\ell \langle\langle X \rangle\rangle$ is comprised of all such *locally convergent* series.

1.1.3 OPERATIONS ON FLIESS OPERATORS

Given a pair of Fliess operators, F_c and F_d with $c, d \in \mathbb{R}_{LC}^\ell \langle\langle X \rangle\rangle$, the parallel sum connection satisfies $F_c + F_d = F_{c+d}$, and the parallel product connection satisfies $F_c F_d = F_{c \sqcup\!\sqcup d}$, as illustrated in Figures 1 and 2, respectively [4]. If the operators are connected in series as shown in Figure 3, where now $c \in \mathbb{R}^\ell \langle\langle X \rangle\rangle$ and $d \in \mathbb{R}^m \langle\langle X \rangle\rangle$, then the resulting

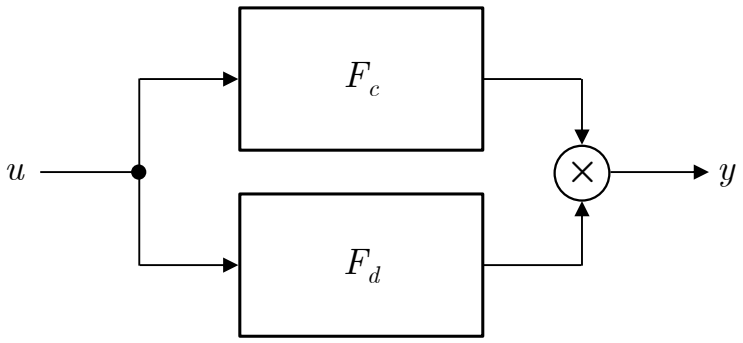


Fig. 2: Parallel product interconnection.

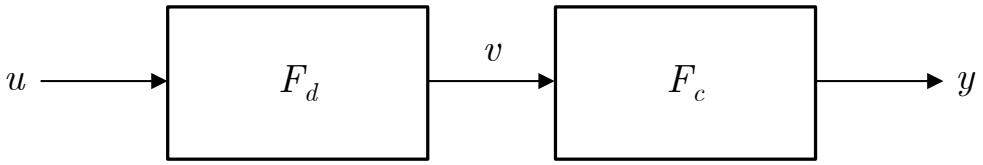


Fig. 3: Series interconnection.

composition $F_c \circ F_d$ always produces another Fliess operator represented by $F_{c \circ d}$. The *composition product* between c and d is defined as

$$c \circ d = \sum_{\eta \in X^*} (c, \eta) \psi_d(\eta)(1)$$

[15–17]. The mapping ψ_d is the ultrametric continuous algebra homomorphism from $\mathbb{R}\langle\langle X \rangle\rangle$ to the set of vector space endomorphisms on $\mathbb{R}\langle\langle X \rangle\rangle$, $\mathrm{End}(\mathbb{R}\langle\langle X \rangle\rangle)$, given by

$$\psi_d(x_i \eta) = \psi_d(x_i) \circ \psi_d(\eta)$$

$$\psi_d(x_i)(e) = x_0(d_i \sqcup\!\sqcup e)$$

for $i = 0, 1, \ldots, m$, $e \in \mathbb{R}\langle\langle X \rangle\rangle$, where d_i is the i-th component series of d ($d_0 := 1$). Further, $\psi_d(\emptyset)$ is the identity map on $\mathbb{R}\langle\langle X \rangle\rangle$. The composition product is both associative and $\mathbb{R}$-linear in its left argument.

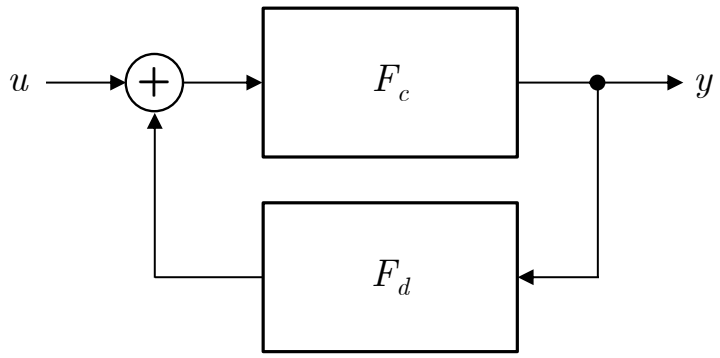


Fig. 4: Feedback interconnection.

Example 1.1.2. Given $c = 1 + x_0x_1$ and $d = x_1x_0$

$$\begin{aligned} c \circ d &= \psi_d(\emptyset)(1) + \psi_d(x_0x_1)(1) \\ &= 1 + x_0(1 \sqcup\!\sqcup x_0[x_1x_0 \sqcup\!\sqcup 1]) \\ &= 1 + x_0x_0x_1x_0. \end{aligned}$$

Consider a Fliess operator F_c whose input is fed by an operator F_d via a feedback connection as shown in Figure 4. The closed-loop system is known to always have a Fliess operator representation [15, 16]. The generating series in this case is given by $c@d$, the *feedback product* of c and d. This product can be computed utilizing the Hopf algebra of coordinate functions associated with the underlying *output feedback group* [18–20]. If I is taken to be the identity map on $L^m_{\mathfrak{p}}[t_0, t_1]$, then define the set of unital Fliess operators $\mathscr{F}_\delta = \{I + F_c : c \in \mathbb{R}\langle\langle X\rangle\rangle\}$. Formally define a generalized series δ such that $F_\delta := I$. In which case, $I + F_c := F_{\delta+c} = F_{c_\delta}$ and $c_\delta = c + \delta$. All such generating series for $\mathscr{F}_\delta$ will be denoted by $\mathbb{R}\langle\langle X_\delta\rangle\rangle$. It can be shown that $(\mathscr{F}_\delta, \circ, I)$ forms a group under the composition

$$F_{c_\delta} \circ F_{d_\delta} = (I + F_c) \circ (I + F_d) = F_{c_\delta \circ d_\delta},$$

where $c_\delta \circ d_\delta := \delta + d + c \,\tilde{\circ}\, d$ for $c, d \in \mathbb{R}\langle\langle X\rangle\rangle$. The *modified composition product*, $\tilde{\circ}$, is defined similarly to the composition product with $c \,\tilde{\circ}\, d = \sum_{\eta \in X^*}(c, \eta)\rho_d(\eta)(1)$. In this case, $\rho_d(x_i\eta) = \rho_d(x_i) \,\tilde{\circ}\, \rho_d(\eta)$ and $\rho_d(x_i)(e) = x_ie + x_0(d_i \sqcup\!\sqcup e)$ for $e \in \mathbb{R}\langle\langle X\rangle\rangle$ [12]. The output feedback Hopf algebra, H, is comprised of the set of coordinate maps of the form

$$\begin{aligned} a^i_\eta : \mathbb{R}^m\langle\langle X\rangle\rangle &\to \mathbb{R} \\ : c &\mapsto (c_i, \eta), \end{aligned}$$

where $\eta \in X^*$ and $i = 1, 2, \ldots, m$. The commutative product is defined as

$$\mu : a^i_\eta \otimes a^j_\xi \mapsto a^i_\eta a^j_\xi.$$

Here the unit $\mathbf{1}$ is defined to map any $c \in \mathbb{R}^m\langle\langle X\rangle\rangle$ to zero. Define the *degree* of a^i_η as $\deg(a^i_\eta) = 2|\eta|_{x_0} + \sum_{j=1}^m |\eta|_{x_j} + 1$, which then renders H graded and connected. Specifically, $H = \bigoplus_{n\geq 0} H_n$, where H_n represents the set of all elements of degree n and $H_0 = \mathbb{R}\mathbf{1}$. The coproduct Δ is defined such that the formal power series product $c \circledcirc d := d + c \,\tilde{\circ}\, d$ satisfies

$$\Delta a^i_\eta(c, d) = a^i_\eta(c \circledcirc d) = (c_i \circledcirc d, \eta).$$

The essential fact is that the group inverse $c^{-1}_\delta := \delta + c^{-1}$ is computed using the antipode of the output feedback Hopf algebra.

Lemma 1.1.1. *[20] The Hopf algebra (H, μ, Δ) has an antipode S satisfying $a^i_\eta(c^{-1}) = (Sa^i_\eta)(c)$ for all $\eta \in X^*$ and $c \in \mathbb{R}^m\langle\langle X\rangle\rangle$.*

The first few anitpode terms, ordered by degree, are:

$$\begin{aligned}
&H_1 : Sa^i_\emptyset = -a^i_\emptyset \\
&H_2 : Sa^i_{x_j} = -a^i_{x_j} \\
&H_3 : Sa^i_{x_0} = -a^i_{x_0} + a^i_{x_\ell} a^\ell_\emptyset \\
&H_3 : Sa^i_{x_j x_k} = -a^i_{x_j x_k} \\
&H_4 : Sa^i_{x_0 x_j} = -a^i_{x_0 x_j} + a^i_{x_\ell} a^\ell_{x_j} + a^i_{x_\ell x_j} a^\ell_\emptyset \\
&H_4 : Sa^i_{x_j x_0} = -a^i_{x_j x_0} + a^i_{x_j x_\ell} a^\ell_\emptyset \\
&H_4 : Sa^i_{x_j x_k x_l} = -a^i_{x_j x_k x_l} \\
&H_5 : Sa^i_{x_0^2} = -a^i_{x_0^2} + a^i_{x_\ell} a^\ell_{x_0} + a^i_{x_\ell x_0} a^\ell_\emptyset + a^i_{x_0 x_\ell} a^\ell_\emptyset - a^i_{x_\nu} a^\nu_{x_\ell} a^\ell_\emptyset - a^i_{x_\nu x_\ell} a^\nu_\emptyset a^\ell_\emptyset,
\end{aligned}$$

for $i, j, k, l = 1, 2, \ldots m$.[1]

The antipode, S, described in Lemma 1.1.1, provides a tool for calculating the feedback product, @. The theorem that follows describes this method.

Theorem 1.1.1. *[20] For any $c, d \in \mathbb{R}^m\langle\langle X\rangle\rangle$ it follows that $c@d = c \,\tilde{\circ}\, (-d \circ c)^{-1} = c \circ (\delta - d \circ c)^{-1}$.*

[1] The Einstein summation notation is used throughout to indicate summations from either 0 or 1 to m, e.g., $\sum_{i=1}^m a_i b^i = a_i b^i$. It will be clear from the context which lower bound is applicable.

The next result utilizes this concept for system inversion of single-input, single-output (SISO) Fliess operators whose generating series have a well defined relative degree in a certain sense [21]. Here the *natural part* of any $c \in \mathbb{R}_{LC}\langle\langle X\rangle\rangle$ is given by $c_N = \sum_{k\geq 0}(c, x_0^k)x_0^k$ so that $c_N \in \mathbb{R}_{LC}[[X_0]]$, where $X_0 := \{x_0\}$.

Theorem 1.1.2. *[21] Suppose $c \in \mathbb{R}_{LC}\langle\langle X\rangle\rangle$ has relative degree r. Let y be analytic at $t = 0$ with generating series $c_y \in \mathbb{R}_{LC}[[X_0]]$ satisfying $(c_y, x_0^k) = (c, x_0^k), k = 0, \ldots, r-1$. Then the input*

$$u(t) = \sum_{k=0}^{\infty}(c_u, x_0^k)\frac{t^k}{k!},$$

where

$$c_u = \left(\left(\frac{(x_0^r)^{-1}(c - c_y)}{(x_0^{r-1}x_1)^{-1}c}\right)^{-1}\right)_N$$

is the unique analytic solution to $F_c[u] = y$ on $[0, T]$ for some $T > 0$.

In both applications, the composition inverse is computed via the antipode S. Since H is a graded connected Hopf algebra, two standard recursions are known for computing them in terms of the coproduct Δ [22]. Recently in [23], a third recursion was identified using so called *derivations*. Little is known at present about the computational efficiency of each of these methods.

1.2 PROBLEM STATEMENT

The goals of this thesis are to:

1. Develop a software implementation in *Mathematica* of the output feedback Hopf algebra antipode using the recent method of derivations.

2. Compare the computational complexity of such an implementation against existing left and right coproduct techniques, and determine empirically which method provides a more efficient approach. To achieve a more robust comparison, a multivariable version of the coproduct code from [22] will be run against the code developed for the method of derivations. Both time and spatial performance will be measured and compared. The intuitive hypothesis is that the method of derivations should provide appreciable performance gains over the two coproduct methods.

1.3 OUTLINE

This thesis is organized into five chapters. Chapter 2 presents the mathematical framework needed to describe the techniques being developed in the sections that follow. These topics include formal languages, operations on words of these languages, power series representations of systems, and Hopf algebra concepts.

Chapter 3 describes the three existing techniques for computing the antipode of the output feedback Hopf algebra: the left coproduct recursion, the right coproduct recursion, and the method of derivations.

In Chapter 4, a software implementation of the derivation method is presented. It is written using the Wolfram language and the full code is listed and detailed. The code makes up a small subset of the *NonCommutative Formal Power Series* (*NCFPS*) package documented in Appendix A. Following the code, a comparison of antipode techniques' performance is made based on the code listed in Appendix B and Appendix C.

Chapter 5 summarizes the conclusions and provides some directions for future research.

CHAPTER 2

MATHEMATICAL PRELIMINARIES

2.1 FORMAL LANGUAGE

In a *formal language*, an *alphabet*, X, is a set of *letters* such as $X = \{x_0, x_1, \ldots, x_m\}$. A finite sequence of the letters from X such as $\eta = x_{i_1} \cdots x_{i_k}$, is a word over X. As in natural languages, permuting the letters changes the word. For example, $\eta_1 = x_0 x_1$ is distinct from $\eta_2 = x_1 x_0$. While distinct, these two words do have the same length, which is denoted by $|\eta|$. Specifically, $|\eta|$ is the number of letters that a word contains. There is also the need to count how many times a given letter in X occurs within a word η. Let $|\eta|_{x_i}$ denote the number of times the letter x_i is present in η. In the case of η_1 as given above, $|\eta_1|_{x_1} = 1$. The set of all possible words using the letters in a given alphabet X is written as X^*. This set includes the empty word, $\emptyset$, which contains no letters, and thus has the property $|\emptyset| = 0$. Any subset $X' \subseteq X^*$ is referred to as a *language*. The following operation on X^* is essential.

Definition 2.1.1. The catenation product is the binary operation

$$\begin{aligned} \mathcal{C} : X^* \times X^* &\to X^* \\ : (\eta_1, \eta_2) &\mapsto \eta_1 \eta_2, \end{aligned}$$

where $\eta_1, \eta_2 \in X^*$.

Normally $\mathcal{C}(\eta_1, \eta_2)$ is written as $\eta_1 \eta_2$. This operation is associative

$$(\eta_1 \eta_2)\eta_3 = \eta_1(\eta_2 \eta_3), \ \forall \eta_i \in X^*,$$

but not commutative since

$$\eta_1 \ \eta_2 \neq \eta_2 \ \eta_1.$$

The empty word acts as the unit, i.e.,

$$\eta\emptyset = \emptyset\eta = \eta, \ \forall \eta \in X^*.$$

Thus, $(X^*, \mathcal{C}, \emptyset)$ is a monoid.

2.2 FORMAL POWER SERIES

A *formal power series* is any mapping of the form $X^* \mapsto \mathbb{R}^\ell$, where ℓ is any positive integer. It is common to represent such a series as a formal sum over words

$$c = \sum_{\eta \in X^*} (c, \eta)\eta,$$

where $(c, \eta) \in \mathbb{R}^\ell$ is used to denote the coefficient of the word η in c. The set $\mathbb{R}^\ell\langle\langle X \rangle\rangle$ contains all possible series which can be formed over the alphabet X. $\mathbb{R}^\ell\langle\langle X \rangle\rangle$ is an $\mathbb{R}$-vector space where addition is defined as

$$c + d = \sum_{\eta \in X^*} ((c, \eta) + (d, \eta))\eta$$

and scalar multiplication by $\alpha \in \mathbb{R}$ is defined as

$$\alpha c = \sum_{\eta \in X^*} \alpha(c, \eta)\eta.$$

2.2.1 OPERATIONS

In this section, a number of important operations on $\mathbb{R}^\ell\langle\langle X \rangle\rangle$ are defined.

Definition 2.2.1. The catenation product of Definition 2.1.1 can be extended over $\mathbb{R}^\ell\langle\langle X \rangle\rangle$ as

$$\begin{aligned} \mathcal{C} : \mathbb{R}^\ell\langle\langle X \rangle\rangle \times \mathbb{R}^\ell\langle\langle X \rangle\rangle &\to \mathbb{R}^\ell\langle\langle X \rangle\rangle \\ : (c, d) &\mapsto \sum_{\eta, \xi \in X^*} (c, \eta)(d, \xi)\eta\xi, \end{aligned}$$

where the product on $\mathbb{R}^\ell \times \mathbb{R}^\ell$ is defined componentwise.

The catenation product on $\mathbb{R}^\ell\langle\langle X \rangle\rangle$ forms an associative $\mathbb{R}$-algebra.

Definition 2.2.2. [4] The shuffle product between two words, $x_i\eta, x_j\xi \in X^*$, is defined recursively as

$$(x_i\eta) \sqcup\!\sqcup (x_j\xi) = x_i(\eta \sqcup\!\sqcup (x_j\xi)) + x_j((x_i\eta) + \xi).$$

This recursion terminates in the case where one or both arguments are the empty word, $\emptyset$, so that

$$\eta \sqcup\!\sqcup \emptyset = \emptyset \sqcup\!\sqcup \eta = \eta.$$

This operation is extended linearly to series in $\mathbb{R}^{\ell}\langle\langle X\rangle\rangle$ as

$$c \sqcup\!\sqcup d = \sum_{\eta,\xi \in X^*} (c,\eta)(d,\xi)\eta \sqcup\!\sqcup \xi.$$

Under the shuffle product, $\mathbb{R}^{\ell}\langle\langle X\rangle\rangle$ is an associative and commutative $\mathbb{R}$-algebra. Two types of series composition are defined next.

Definition 2.2.3. [15,16] For $X_1 = \{x_0, x_1, \ldots, x_m\}$, $X_2 = \{x_0, x_1, \ldots, x_n\}$, and two series $c \in \mathbb{R}^{\ell}\langle\langle X_1\rangle\rangle$ and $d \in \mathbb{R}^{m}\langle\langle X_2\rangle\rangle$, the composition product is

$$c \circ d = \sum_{\eta \in X^*} (c,\eta)\psi_d(\eta)(1),$$

where $c \circ d \in \mathbb{R}^{\ell}\langle\langle X_2\rangle\rangle$ and the mapping ψ_d is given as

$$\begin{aligned} \psi_d : X &\to \mathrm{End}(\mathbb{R}\langle\langle X\rangle\rangle) \\ : x_i &\mapsto x_0(d_i \sqcup\!\sqcup \cdot) \end{aligned}$$

with $\psi_d(x_i\eta) = \psi_d(x_i) \circ \psi_d(\eta)$, d_i is i-th component series of d, and $d_0 := 1$. On $\mathbb{R}\langle\langle X\rangle\rangle$, $\psi_d(\emptyset)$ acts as the identity map.

The product described in Definition 2.2.3 has the property of being associative as well as left $\mathbb{R}$-linear.

Definition 2.2.4. [12] For $X_1 = \{x_0, x_1, \ldots, x_m\}$, $X_2 = \{x_0, x_1, \ldots, x_n\}$, and two series $c \in \mathbb{R}^{\ell}\langle\langle X_1\rangle\rangle$ and $d \in \mathbb{R}^{m}\langle\langle X_2\rangle\rangle$

$$c \tilde{\circ} d = \sum_{\eta \in X^*} (c,\eta)\rho_d(\eta)(1),$$

where $c \tilde{\circ} d \in \mathbb{R}^{\ell}\langle\langle X_2\rangle\rangle$ is referred to as the *modified composition product*. The mapping ρ_d is given as

$$\begin{aligned} \rho_d : X &\to \mathrm{End}(\mathbb{R}\langle\langle X\rangle\rangle) \\ : x_i &\mapsto x_i \cdot + x_0(d_i \sqcup\!\sqcup \cdot), \end{aligned}$$

where $\rho_d(x_i\eta) = \rho_d(x_i) \circ \rho_d(\eta)$, d_i is the i-th component series of d, and $d_0 := 1$.

2.3 SYSTEM REPRESENTATIONS USING FLIESS OPERATORS

With any series c one can associate an input-output map. First, the input space is defined.

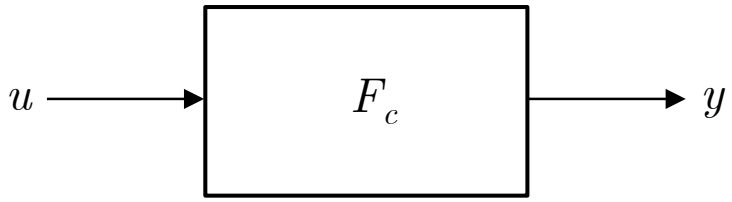


Fig. 5: Input-output system with Fliess operator representation.

Definition 2.3.1. For any Lebesgue measurable function $u : [t_0, t_1] \to \mathbb{R}^m$, define $||u||_{\mathfrak{p}} = \max\{||u_i||_{\mathfrak{p}} : 1 \leq i \leq m\}$ with $\mathfrak{p} \geq 1$ and $||u_i||_{\mathfrak{p}}$ denoting the usual $L_{\mathfrak{p}}$-norm for a measurable, real-valued function, u_i, defined on $[t_0, t_1]$. The set of all such u having a finite $||\cdot||_{\mathfrak{p}}$ norm is written as $L^m_{\mathfrak{p}}[t_0, t_1]$.

The ball of radius R in $L^m_{\mathfrak{p}}[t_0, t_1]$ centered at the origin is denoted by

$$B^m_{\mathfrak{p}}(R)[t_0, t_1] := \{u \in L^m_{\mathfrak{p}}[t_0, t_1] : ||u||_{\mathfrak{p}} \leq R\}.$$

The subset of all continuous functions in $L^m_1[t_0, t_1]$ is denoted $C[t_0, t_1]$. For any $\eta \in X^*$ there is an associated iterated integral, $E_\eta : \ L^m_1[t_0, t_1] \to C[t_0, t_1]$, defined inductively by

$$E_{x_i\bar{\eta}}[u](t, t_0) = \int_{t_0}^{t} u_i(\tau) E_{\bar{\eta}}[u](\tau, t_0) d\tau,$$

where $x_i \in X$, $\bar{\eta} \in X^*$, $u_0 = 1$, and $E_\emptyset[u] = 1$. In the context of nonlinear system modeling, the letter x_0 represents an internal input from the system itself (a sort of zero-input response) where as the remaining letters of an m-input system, $x_1, \ldots, x_m$, represent the external inputs.

Definition 2.3.2. [4] For a given generating series $c \in \mathbb{R}^\ell\langle\langle X\rangle\rangle$, the corresponding *Fliess operator* is

$$F_c[u](t) = \sum_{\eta \in X^*} (c, \eta) E_\eta[u](t, t_0).$$

The summation above is only a formal expression unless convergence can be proven.

Definition 2.3.3. [13] Let $|z| := \max_{1 \leq i \leq \ell} |z_i|$ for $z \in \mathbb{R}^\ell$. A series $c \in \mathbb{R}^\ell\langle\langle X\rangle\rangle$ is said to have Gevery order $s \in \mathbb{R}$ if there exist constants $K, M > 0$ such that

$$|(c, \eta)| \leq K M^{|\eta|} (|\eta|!)^s, \ \forall \eta \in X^*.$$

For the case where $s = 1$, it is known that the sum defining $F_c[u]$ converges uniformly and absolutely for any $u \in B^m_{\mathfrak{p}}(R)[t_0, t_1]$ provided that $R, T > 0$ are sufficiently small [14]. Therefore, F_c maps $B^m_{\mathfrak{p}}(R)[t_0, t_0 + T]$ into $B^\ell_{\mathfrak{q}}(S)[t_0, t_0 + T]$ when $\mathfrak{p}, \mathfrak{q} \in [1, \infty]$ are conjugate exponents, i.e. $1/\mathfrak{p} + 1/\mathfrak{q} = 1$. All such locally convergent series are denoted by $\mathbb{R}^\ell_{LC}\langle\langle X \rangle\rangle$.

2.3.1 SYSTEM INTERCONNECTIONS

For input-output systems which have locally convergent Fliess operator representations, it is known that all the standard system interconnections in control theory also have Fliess operator representations which are locally convergent.

Theorem 2.3.1. *[4] Given two Fliess operators F_c and F_d with $c, d \in \mathbb{R}^\ell_{LC}\langle\langle X \rangle\rangle$, the parallel sum connection shown in Figure 6 is equivalent to*

$$F_c + F_d = F_{c+d},$$

where $c + d \in \mathbb{R}^\ell_{LC}\langle\langle X \rangle\rangle$.

Example 2.3.1. Let $F_c[u](t) = 1 + \int_{t_0}^t u_1(\tau) d\tau$ and $F_d[u](t) = 2 \int_{t_0}^t \int_{t_0}^{\tau_1} u_1(\tau_2) \int_{t_0}^{\tau_2} d\tau_3 d\tau_2 d\tau_1$. Then F_c and F_d have generating series $c = 1 + x_1$ and $d = 2x_0x_1x_0$ respectively, and $F_c + F_d$ has the generating series

$$c + d = 1 + x_1 + 2x_0x_1x_0.$$

Thus,

$$F_{c+d}[u](t) = 1 + \int_{t_0}^t u_1(\tau) d\tau + 2 \int_{t_0}^t \int_{t_0}^{\tau_1} u_1(\tau_2) \int_{t_0}^{\tau_2} d\tau_3 d\tau_2 d\tau_1.$$

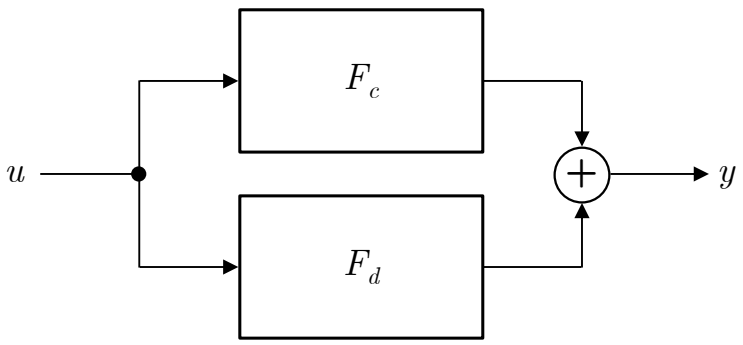


Fig. 6: Block diagram of F_{c+d}.

Theorem 2.3.2. *[4] Given two Fliess operators F_c and F_d with $c, d \in \mathbb{R}^{\ell}_{LC}\langle\langle X\rangle\rangle$, the parallel product connection shown in Figure 7 is equivalent to*

$$F_c F_d = F_{c \sqcup\!\sqcup d},$$

where $c \sqcup\!\sqcup d \in \mathbb{R}^{\ell}_{LC}\langle\langle X\rangle\rangle$.

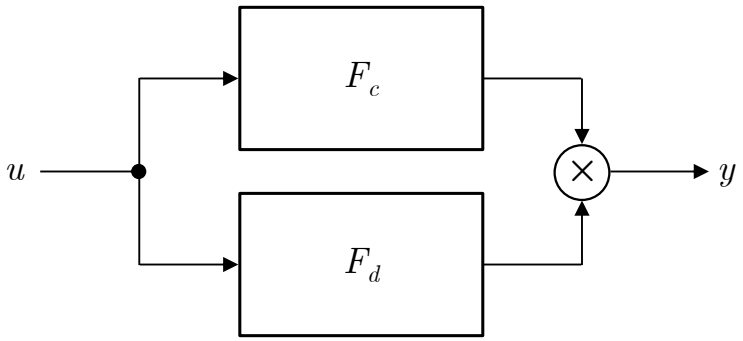


Fig. 7: Block diagram of $F_{c \sqcup\!\sqcup d}$.

Example 2.3.2. Assume as in Example 2.3.1, that F_c and F_d have generating series $c = 1 + x_1$ and $d = 2x_0x_1x_0$. The generating series for F_cF_d is given by

$$\begin{aligned} c \sqcup\!\sqcup d =& (1 + x_1) \sqcup\!\sqcup 2x_0x_1x_0 \\ =& 2x_0x_1x_0 + x_1 \sqcup\!\sqcup 2x_0x_1x_0 \\ =& 2x_0x_1x_0 + 2x_1x_0x_1x_0 + 4x_0x_1x_1x_0 + 2x_0x_1x_0x_1. \end{aligned}$$

Theorem 2.3.3. *[15, 16] Given two Fliess operators F_c and F_d with $c \in \mathbb{R}^{\ell}_{LC}\langle\langle X_1\rangle\rangle$, $d \in \mathbb{R}^{m}_{LC}\langle\langle X_2\rangle\rangle$, $\ell, m \in \mathbb{Z}^+$, and $|X_1| = m + 1$, the series connection shown in Figure 8 is equivalent to*

$$F_c \circ F_d = F_{c \circ d}$$

where $c \circ d \in \mathbb{R}^{\ell}_{LC}\langle\langle X_2\rangle\rangle$ as defined in Definition 2.2.3.

Example 2.3.3. For $X = \{x_0, x_1\}$ and generating series $c = 1 + x_1$ and $d = 2x_0x_1x_0$, the generating series for $F_c \circ F_d$ is given by

$$\begin{aligned} c \circ d =& (1 + x_1) \circ 2x_0x_1x_0 \\ =& \psi_d(\emptyset)(1) + 2\psi_d(x_1)(1) \\ =& 1 + 2x_0(x_0x_1x_0 \sqcup\!\sqcup 1) \\ =& 1 + 2x_0x_0x_1x_0. \end{aligned}$$

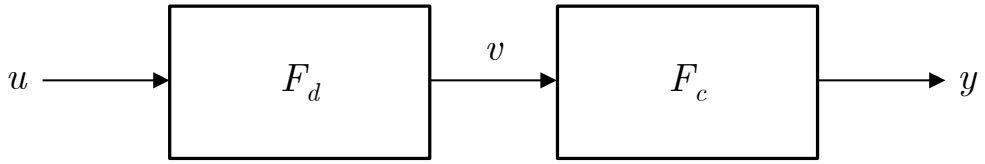


Fig. 8: Block diagram of $F_{c \circ d}$.

Theorem 2.3.4. *[19, 20] Given two Fliess operators F_c and F_d with $c \in \mathbb{R}^{\ell}_{LC}\langle\langle X_1 \rangle\rangle$, $d \in \mathbb{R}^{m}_{LC}\langle\langle X_2 \rangle\rangle$, $\ell, m \in \mathbb{Z}^{+}$, $|X_1| = m + 1$, and $|X_2| = \ell + 1$, the feedback connection shown in Figure 9 is equivalent to $F_{c@d}$, where*

$$c@d = c \,\tilde{\circ}\, (-d \circ c)^{-1}, \tag{1}$$

where @ is called the feedback product, and c^{-1} is the composition inverse of series c.

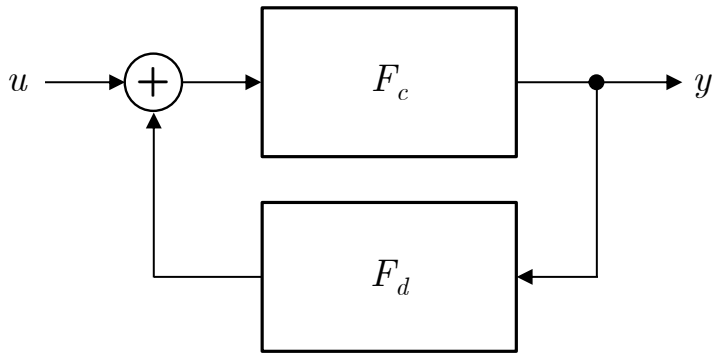


Fig. 9: Block diagram of $F_{c@d}$.

As indicated in the last theorem, there is a need to compute the composition inverse of a series. The mathematics behind this concept are presented next.

2.4 COMPOSITION INVERSE

The feedback product in Equation (1) is related to the *output feedback group* [19, 20]. Define the set of unital Fliess operators to be

$$\mathscr{F}_{\delta} = \{I + F_c : c \in \mathbb{R}\langle\langle X \rangle\rangle\},$$

where I is the identity map. A fictitious series, δ, is utilized here as a placeholder to represent the generating series of the identity map, I, i.e. $F_{\delta} := I$. In which case $I + F_c := F_{\delta + c} = F_{c_{\delta}}$,

where $c_\delta = c+\delta$. The set $\mathbb{R}\langle\langle X_\delta \rangle\rangle$ contains all generating series for $\mathscr{F}_\delta$. Composing operators from this set gives a group $(\mathscr{F}_\delta, \circ, I)$, where

$$F_{c_\delta} \circ F_{d_\delta} = (I + F_c) \circ (I + F_d) = F_{c_\delta \circ d_\delta}$$

with

$$c_\delta \circ d_\delta := \delta + d + c \,\tilde{\circ}\, d. \tag{2}$$

A unital associative $\mathbb{R}$-algebra (A, μ, σ) with $\mathbb{R}$-vector space A can be described in terms of two $\mathbb{R}$-linear maps

$$\mu : A \otimes A \to A$$

and

$$\sigma : \mathbb{R} \to A.$$

The product μ is associative satisfying $(ab)c = a(bc),\ a, b, c \in A$ where $ab := \mu(a \otimes b)$. The map σ satisfies $\mathbf{1}a = a = a\mathbf{1},\ a \in A$, where $\mathbf{1} := \sigma(1)$ is the unit of A.

A dual of (A, μ, σ) called a counital coassociative *coalgebra* can be constructed using two $\mathbb{R}$-linear maps

$$\Delta : A \to A \otimes A$$

and

$$\epsilon : A \to \mathbb{R}$$

that satisfy the coassociative and counital properties, respectively. That is, $(\mathrm{id} \otimes \Delta) \circ \Delta = (\Delta \otimes \mathrm{id}) \circ \Delta$, where id is the identity map on A, and $(\epsilon \otimes \mathrm{id}) \circ \Delta = \mathrm{id} = (\mathrm{id} \otimes \epsilon) \circ \Delta$. The triple (A, Δ, ϵ) is the coalgebra.

For two $\mathbb{R}$-algebras (A_1, μ_1, σ_1) and (A_2, μ_2, σ_2), a $\mathbb{R}$-linear map $\psi : A_1 \to A_2$ where

$$\psi \circ \mu_1 = \mu_2 \circ (\psi \otimes \psi)$$
$$\psi \circ \sigma = \sigma_2$$

is known as an $\mathbb{R}$-*algebra homomorphism*. A similar mapping can be defined for coalgebras. If Δ and ϵ are both $\mathbb{R}$-algebra homomorphisms, the five-tuple $(A, \mu, \sigma, \Delta, \epsilon)$ is called an $\mathbb{R}$-*bialgebra*.

Consider the *Hopf convolution product* defined as

$$f \star g := \mu \circ (f \otimes g) \circ \Delta,$$

where f and g are elements of $\mathrm{End}(A)$, the set of all endomorphisms on A. Along with the unit $\vartheta = \sigma \circ \epsilon$, the triple $(\mathrm{End}(A), \star, \vartheta)$ forms an associative $\mathbb{R}$-algebra.

The *antipode* of the bialgebra, $S \in \mathrm{End}(A)$, satisfies $S \star \mathrm{id} = \mathrm{id} \star S = \vartheta$. When it exists, the antipode is unique. The convolution inverse of id is S. That is

$$S = \mathrm{id}^{\star -1} = (\vartheta - (\vartheta - \mathrm{id}))^{\star -1} = \vartheta + \sum_{k=1}^{\infty} (\vartheta - \mathrm{id})^{\star k}.$$

Definition 2.4.1. [24] A *Hopf algebra* is a bialgebra $(H, \mu, \sigma, \Delta, \epsilon)$ with an antipode $S \in \mathrm{End}(H)$.

There is a connected graded Hopf algebra corresponding to the group $(\mathscr{F}_\delta, \circ, I)$ denoted by H [25–27]. It contains *coordinate maps* of the form

$$\begin{aligned} a_\eta^i : \mathbb{R}^m \langle\langle X \rangle\rangle &\to \mathbb{R} \\ : c &\mapsto (c_i, \eta), \end{aligned}$$

where $\eta \in X^*$ and $i = 1, 2, \ldots, m$. The commutative product on H is defined as

$$\mu : a_\eta^i \otimes a_\xi^j \mapsto a_\eta^i a_\xi^j.$$

The unit $\mathbf{1}$ will map any $c \in \mathbb{R}\langle\langle X \rangle\rangle$ to zero. Each coordinate function $a_\eta^i \in H$ has a *degree* given by

$$\deg(a_\eta^i) = 2|\eta|_{x_0} + \sum_{j=1}^{m} |\eta|_{x_j} + 1.$$

The degree allows for H to be graded and connected with $H = \bigoplus_{n \geq 0} H_n$, where H_n is the set containing all coordinate functions of degree n and $H_0 = \mathbb{R}\mathbf{1}$. The *coproduct* on H, $\Delta : H \to H \otimes H$, is defined as

$$\Delta a_\eta^i(c, d) = a_\eta^i(c \circledcirc d) = (c_i \circledcirc d, \eta),$$

where the formal series product, $\circledcirc$, is given by Equation (2). Namely,

$$c \circledcirc d := d + c \,\tilde{\circ}\, d.$$

The key point is that the inverse in $\mathscr{F}_\delta$ can be computed using the antipode in H. Suppose $F_{c_\delta}^{-1} = F_{c_\delta^{-1}}$, where $c_\delta^{-1} = \delta + c^{-1}$. Theorem 2.4.1 describes how to compute c^{-1}.

Theorem 2.4.1. *[20] The Hopf algebra (H, μ, Δ) has an antipode S satisfying $a_\eta^i(c^{-1}) = (S a_\eta^i)(c)$ for all $\eta \in X^*$, $i = 1, 2, \ldots, m$, and $c \in \mathbb{R}^m \langle\langle X \rangle\rangle$.*

As an example, the first few coordinate function antipodes are listed below.

$$
\begin{aligned}
&H_1 : Sa^i_{\emptyset} = -a^i_{\emptyset} \\
&H_2 : Sa^i_{x_j} = -a^i_{x_j} \\
&H_3 : Sa^i_{x_0} = -a^i_{x_0} + a^i_{x_\ell} a^\ell_{\emptyset} \\
&H_3 : Sa^i_{x_j x_k} = -a^i_{x_j x_k} \\
&H_4 : Sa^i_{x_0 x_j} = -a^i_{x_0 x_j} + a^i_{x_\ell} a^\ell_{x_j} + a^i_{x_\ell x_j} a^\ell_{\emptyset} \\
&H_4 : Sa^i_{x_j x_0} = -a^i_{x_j x_0} + a^i_{x_j x_\ell} a^\ell_{\emptyset} \\
&H_4 : Sa^i_{x_j x_k x_l} = -a^i_{x_j x_k x_l} \\
&H_5 : Sa^i_{x_0^2} = -a^i_{x_0^2} + a^i_{x_\ell} a^\ell_{x_0} + a^i_{x_\ell x_0} a^\ell_{\emptyset} + a^i_{x_0 x_\ell} a^\ell_{\emptyset} - a^i_{x_\nu} a^\nu_{x_\ell} a^\ell_{\emptyset} - a^i_{x_\nu x_\ell} a^\nu_{\emptyset} a^\ell_{\emptyset},
\end{aligned}
\tag{3}
$$

for $i, j, k, \ell = 1, 2, \ldots m$.[1]

Given that series inversion is an integral step in computing the output of feedback-connected Fliess operators, the next chapter focuses on known algorithms that compute the antipode of H [18–20, 23].

[1]The Einstein summation notation is used throughout to indicate summations from either 0 or 1 to m, e.g., $\sum_{i=1}^m a_i b^i = a_i b^i$. It will be clear from the context which lower bound is applicable.

CHAPTER 3

ANTIPODE ALGORITHMS

The three existing methods for calculating the antipode of the output feedback Hopf algebra (H, μ, Δ) are described in this chapter. Described first are two techniques which utilize the coproduct Δ of H as well as *left augmentation operators* [18–20]. The third approach is centered around *right augmentation operators* which act as derivations on the product μ [23].

3.1 COPRODUCT METHODS

Consider the $\mathbb{R}$-vector space of coordinate functions with positive degree, $V_+ \subset H$.

Definition 3.1.1. For any $x_i \in X$, $\eta \in X^*$, and a^j_η in H, the *left augmentation operator* is defined as

$$\theta_{x_i}(a^j_\eta) := a^j_{x_i\eta}$$

with $\theta_{x_i}(\mathbf{1}) := \mathbf{0}$. For any words $\eta, \xi \in X^*$ with $\eta := x_{i_1}x_{i_2}\cdots x_{i_k}$ let

$$\theta_\eta(a^j_\xi) := \theta_{x_{i_k}} \circ \cdots \circ \theta_{x_{i_2}} \circ \theta_{x_{i_1}}(a^j_\xi).$$

The left augmentation operator acts as an endomorphism on V_+.

Definition 3.1.2. The *deshuffling coproduct*, $\Delta^j_{\sqcup\!\sqcup}(V_+) \subset V_+ \otimes V_+$, is defined on V_+ such that

$$\begin{aligned} \Delta^j_{\sqcup\!\sqcup}\, a^i_\emptyset &= a^i_\emptyset \otimes a^j_\emptyset \\ \Delta^j_{\sqcup\!\sqcup} \circ \theta_{x_k} &= (\theta_{x_k} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_k}) \circ \Delta^j_{\sqcup\!\sqcup}\,, \end{aligned}$$

where id is the identity map on H.

Example 3.1.1. Consider

$$\begin{aligned} \Delta^1_{\sqcup\!\sqcup}\, a^2_{x_0x_1} &= \Delta^1_{\sqcup\!\sqcup} \circ \theta_{x_0} \circ \theta_{x_1}(a^2_\emptyset) \\ &= (\theta_{x_0} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_0}) \circ \Delta^1_{\sqcup\!\sqcup} \circ \theta_{x_1}(a^2_\emptyset) \\ &= (\theta_{x_0} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_0}) \circ (\theta_{x_1} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_1}) \circ \Delta^1_{\sqcup\!\sqcup}\,(a^2_\emptyset) \\ &= (\theta_{x_0} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_0}) \circ (\theta_{x_1} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_1}) \circ a^2_\emptyset \otimes a^1_\emptyset \\ &= (\theta_{x_0} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_0}) \circ (a^2_{x_1} \otimes a^1_\emptyset + a^2_\emptyset \otimes a^1_{x_1}) \\ &= a^2_{x_0x_1} \otimes a^1_\emptyset + a^2_{x_1} \otimes a^1_{x_0} + a^2_{x_0} \otimes a^1_{x_1} + a^2_\emptyset \otimes a^1_{x_0x_1}. \end{aligned}$$

The next theorem states that the coproduct $\tilde{\Delta} := \Delta - \mathbf{1} \otimes \mathrm{id}$ can be computed by the following recursion.

Theorem 3.1.1. *[20] The following identities hold:*

(1) $\tilde{\Delta} a^i_\emptyset = a^i_\emptyset \otimes \mathbf{1}$

(2) $\tilde{\Delta} \circ \theta_{x_i} = (\theta_{x_i} \otimes \mathrm{id}) \circ \tilde{\Delta}$

(3) $\tilde{\Delta} \circ \theta_{x_0} = (\theta_{x_0} \otimes \mathrm{id}) \circ \tilde{\Delta} + (\theta_{x_i} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ \Delta^i{}_{\sqcup\!\sqcup}$

for $i = 1, 2, \ldots, m$.

Example 3.1.2. Given $X = \{x_0, x_1, x_2\}$ consider

$$\begin{aligned}
\tilde{\Delta} a^2_{x_1 x_0} &= \tilde{\Delta} \circ \theta_{x_1} \circ \theta_{x_0}(a^2_\emptyset) \\
&= (\theta_{x_1} \otimes \mathrm{id}) \circ [(\theta_{x_0} \otimes \mathrm{id}) \circ \tilde{\Delta} + (\theta_{x_1} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ \Delta^1{}_{\sqcup\!\sqcup} \\
&\quad + (\theta_{x_2} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ \Delta^2{}_{\sqcup\!\sqcup}](a^2_\emptyset) \\
&= (\theta_{x_1} \otimes \mathrm{id}) \circ [(\theta_{x_0} \otimes \mathrm{id}) \circ (a^2_\emptyset \otimes \mathbf{1}) + (\theta_{x_1} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ (a^2_\emptyset \otimes a^1_\emptyset) \\
&\quad + (\theta_{x_2} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ (a^2_\emptyset \otimes a^2_\emptyset)] \\
&= (\theta_{x_1} \otimes \mathrm{id}) \circ [(a^2_{x_0} \otimes \mathbf{1}) + (\theta_{x_1} \otimes \mu) \circ (a^2_\emptyset \otimes \mathbf{1} \otimes a^1_\emptyset) \\
&\quad + (\theta_{x_2} \otimes \mu) \circ (a^2_\emptyset \otimes \mathbf{1} \otimes a^2_\emptyset)] \\
&= (\theta_{x_1} \otimes \mathrm{id}) \circ [a^2_{x_0} \otimes \mathbf{1} + a^2_{x_1} \otimes a^1_\emptyset + a^2_{x_2} \otimes a^2_\emptyset] \\
&= a^2_{x_1 x_0} \otimes \mathbf{1} + a^2_{x_1^2} \otimes a^1_\emptyset + a^2_{x_2 x_1} \otimes a^2_\emptyset.
\end{aligned}$$

The next result is a classical theorem from the theory of Hopf algebras.

Theorem 3.1.2. *[28] The antipode, S, of any graded connected Hopf algebra (H, μ, Δ) can be computed for any $a \in H_k$, $k \geq 1$ by*

$$Sa = -a - \sum (Sa'_{(1)}) a'_{(2)} = -a - \sum a'_{(1)} Sa'_{(2)},$$

where the reduced coproduct is $\Delta' a = \Delta a - a \otimes \mathbf{1} - \mathbf{1} \otimes a = \sum a'_{(1)} a'_{(2)}$ (using the notation of Sweedler).

From Theorem 3.1.2 there are two possibilities for computing the antipode, S. Apply S to the left side of the coproduct (*left coproduct recursion*) or apply S to the right side of the coproduct (*right coproduct recursion*). The only potential difference between the two

methods is in regards to the efficiency of the computation. In [22] it was shown in the SISO case that the left coproduct recursion produces terms which cancel out in the final result. The right coproduct recursion was shown to be more efficient in [18] as it was proven that this recursion is free of cancellations.

The following theorem provides a fully recursive algorithm to compute the antipode of the output feedback Hopf algebra.

Theorem 3.1.3. *[20] The antipode, S, of any $a^i_\eta \in V_+$ in the output feedback Hopf algebra can be computed by the following algorithm:*

i. *Recursively compute $\Delta^j_{\sqcup\!\sqcup}$ via Definition 3.1.2.*

ii. *Recursively compute $\tilde{\Delta}$ via Theorem 3.1.1.*

iii. *Recursively compute S via Theorem 3.1.2 with $\Delta' a^i_\eta = \tilde{\Delta} a^i_\eta - a^i_\eta \otimes \mathbf{1}$.*

Example 3.1.3. Let $X = \{x_0, x_1, x_2\}$ and $\eta = x_0x_1$. To calculate $Sa^2_{x_0x_1}$ by either coproduct method, it is first necessary to compute the reduced coproduct $\Delta' a^2_{x_0x_1}$. Observe

$$\begin{aligned}
\Delta' a^2_{x_0x_1} &= \tilde{\Delta} a^2_{x_0x_1} - a^2_{x_0x_1} \otimes \mathbf{1} \\
&= \tilde{\Delta} \circ \theta_{x_0} \circ \theta_{x_1}(a^2_\emptyset) - a^2_{x_0x_1} \otimes \mathbf{1} \\
&= [(\theta_{x_0} \otimes \mathrm{id}) \circ \tilde{\Delta} + (\theta_{x_1} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ \Delta^1_{\sqcup\!\sqcup} \\
&\quad + (\theta_{x_2} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ \Delta^2_{\sqcup\!\sqcup}\,] \circ \theta_{x_1} \circ a^2_\emptyset - a^2_{x_0x_1} \otimes \mathbf{1} \\
&= (\theta_{x_0} \otimes \mathrm{id}) \circ (\theta_{x_1} \otimes \mathrm{id}) \circ a^2_\emptyset \otimes \mathbf{1} \\
&\quad + (\theta_{x_1} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ (\theta_{x_1} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_1}) \circ a^2_\emptyset \otimes a^1_\emptyset \\
&\quad + (\theta_{x_2} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ (\theta_{x_1} \otimes \mathrm{id} + \mathrm{id} \otimes \theta_{x_1}) \circ a^2_\emptyset \otimes a^2_\emptyset - a^2_{x_0x_1} \otimes \mathbf{1} \\
&= a^2_{x_0x_1} \otimes \mathbf{1} + (\theta_{x_1} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ (a^2_{x_1} \otimes a^1_\emptyset + a^2_\emptyset \otimes a^1_{x_1}) \\
&\quad + (\theta_{x_2} \otimes \mu) \circ (\tilde{\Delta} \otimes \mathrm{id}) \circ (a^2_{x_1} \otimes a^2_\emptyset + a^2_\emptyset \otimes a^2_{x_1}) - a^2_{x_0x_1} \otimes \mathbf{1} \\
&= (\theta_{x_1} \otimes \mu) \circ (a^2_{x_1} \otimes \mathbf{1} \otimes a^1_\emptyset + a^2_\emptyset \otimes \mathbf{1} \otimes a^1_{x_1}) \\
&\quad + (\theta_{x_2} \otimes \mu) \circ (a^2_{x_1} \otimes \mathbf{1} \otimes a^2_\emptyset + a^2_\emptyset \otimes \mathbf{1} \otimes a^2_{x_1}) \\
&= a^2_{x_1x_1} \otimes a^1_\emptyset + a^2_{x_1} \otimes a^1_{x_1} + a^2_{x_2x_1} \otimes a^2_\emptyset + a^2_{x_2} \otimes a^2_{x_1}.
\end{aligned}$$

In which case, the left coproduct formula yields

$$\begin{aligned}
Sa^i_\eta &= -a^i_\eta - \sum (Sa'_{(1)}) a'_{(2)} \\
Sa^2_{x_0x_1} &= -a^2_{x_0x_1} - a^2_{x_1x_1} a^1_\emptyset - a^2_{x_1} a^1_{x_1} - a^2_{x_2x_1} a^2_\emptyset - a^2_{x_2} a^2_{x_1},
\end{aligned}$$

while the right coproduct formula gives

$$\begin{aligned} Sa^i_\eta &= -a^i_\eta - \sum a'_{(1)} Sa'_{(2)} \\ Sa^2_{x_0x_1} &= -a^2_{x_0x_1} - a^2_{x_1x_1}a^1_\emptyset - a^2_{x_1}a^1_{x_1} - a^2_{x_2x_1}a^2_\emptyset - a^2_{x_2}a^2_{x_1}. \end{aligned}$$

Example 3.1.4. Let $X = \{x_0, x_1\}$ and $\eta = x_0^2$. When applying the coproduct to the term on the left

$$\begin{aligned} Sa^1_\eta &= -a^1_{x_0^2} - S(a^1_{x_1})a^1_{x_0} - S(a^1_{x_1x_0})a^1_\emptyset - S(a^1_{x_0x_1})a^1_\emptyset - S(a^1_{x_1^2})(a^1_\emptyset)^2 \\ &= -a^1_{x_0^2} - (-a^1_{x_1})a^1_{x_0} - (-a^1_{x_1x_0} + \cancel{a^1_{x_1^2}a^1_\emptyset})a^1_\emptyset \\ &\quad - (-a^1_{x_0x_1} + (a^1_{x_1})^2 + a^1_{x_1^2}a^1_\emptyset)a^1_\emptyset - \cancel{(-a^1_{x_1^2})(a^1_\emptyset)^2} \\ &= -a^1_{x_0^2} + a^1_{x_1}a^1_{x_0} + a^1_{x_1x_0}a^1_\emptyset + a^1_{x_0x_1}a^1_\emptyset - (a^1_{x_1})^2a^1_\emptyset - a^1_{x_1^2}(a^1_\emptyset)^2. \end{aligned}$$

For the left coproduct method applied to x_0^2 as in Example 3.1.4 it can be seen that of the six unique terms generated, one of them has a cancellation which leaves the term with a coefficient of 1. Table I counts the terms for a few antipodes of coordinate functions in the form $a^i_{x_0^n}$, $n \in \mathbb{Z}^+$, following the same convention as the previous example.

TABLE I: [18] Total number of terms (with and without multiplicities) and cancellations when the left coproduct recursion is applied and $m = 1$.

Coordinate Map Degree	Total Number of Unique Terms	Total Number of Terms	Number of Cancellations
3	2	2	0
5	6	6	1
7	17	26	9
9	50	150	70
11	139	1082	427
13	390	9366	2417
15	1059	94,586	12,730

3.2 DERIVATION METHOD

Analogous to θ_{x_k} in the previous section, consider now the right augmentation operator.

Definition 3.2.1. For any $x_i \in X$, $\eta \in X^*$, and a_η^j in H, the *right augmentation operator* is defined as

$$\tilde{\theta}_{x_i}(a_\eta^j) := a_{\eta x_i}^j$$

with $\tilde{\theta}_{x_i}(\mathbf{1}) := \mathbf{0}$. For any words $\eta, \xi \in X^*$ with $\eta := x_{i_1} x_{i_2} \cdots x_{i_k}$ let

$$\tilde{\theta}_\eta(a_\xi^j) := \tilde{\theta}_{x_{i_k}} \circ \cdots \circ \tilde{\theta}_{x_{i_2}} \circ \tilde{\theta}_{x_{i_1}}(a_\xi^j).$$

In the case of products of coordinate functions in H, the right augmentation operator acts as a derivation (Leibniz rule). That is,

$$\tilde{\theta}_\eta(a_{\eta_1}^{j_1} a_{\eta_2}^{j_2} \cdots a_{\eta_k}^{j_k}) := \sum_{l=1}^{k} a_{\eta_1}^{j_1} a_{\eta_2}^{j_2} \cdots \tilde{\theta}_\eta(a_{\eta_l}^{j_l}) \cdots a_{\eta_k}^{j_k}.$$

Example 3.2.1. Consider

$$\begin{aligned}\tilde{\theta}_{x_0}(a_{x_0}^i a_{x_1}^i a_{x_0x_2}^i) &= \tilde{\theta}_{x_0}(a_{x_0}^i) a_{x_1}^i a_{x_0x_2}^i + a_{x_0}^i \tilde{\theta}_{x_0}(a_{x_1}^i) a_{x_0x_2}^i + a_{x_0}^i a_{x_1}^i \tilde{\theta}_{x_0}(a_{x_0x_2}^i) \\ &= a_{x_0^2}^i a_{x_1}^i a_{x_0x_2}^i + a_{x_0}^i a_{x_1x_0}^i a_{x_0x_2}^i + a_{x_0}^i a_{x_1}^i a_{x_0x_2x_0}^i.\end{aligned}$$

The following operations will also be useful:

$$\begin{aligned}\tilde{\theta}'_{x_0}(a_\eta^j) &= -\tilde{\theta}_{x_0}(a_\eta^j) + \sum_{k=1}^{m} a_\emptyset^k \tilde{\theta}_{x_k}(a_\eta^j) \\ &= -a_{\eta x_0}^j + \sum_{k=1}^{m} a_\emptyset^k a_{\eta x_k}^j \\ \tilde{\theta}'_{x_i}(a_\eta^j) &= -\tilde{\theta}_{x_i}(a_\eta^j) = -a_{\eta x_i}^j, \quad i \neq 0.\end{aligned} \tag{4}$$

Analogously,

$$\tilde{\Theta}'_\xi = \tilde{\theta}'_{x_{i_k}} \circ \cdots \circ \tilde{\theta}'_{x_{i_2}} \circ \tilde{\theta}'_{x_{i_1}},$$

where $\xi := x_{i_1} x_{i_2} \cdots x_{i_k}$. Since $\tilde{\theta}'_{x_i}$ is defined linearly in terms of $\tilde{\theta}_{x_i}$, it will also act as a derivation on H, and consequently, so will $\tilde{\Theta}'_\xi$. The following theorem describes how to compute the antipode, S, exclusively in terms of right-augmentation operators.

Theorem 3.2.1. *[23] For any nonempty word $\eta \in X^*$, the antipode $S : H \to H$ in the output feedback Hopf algebra can be written as*

$$S a_\eta^i = (-1)^{|\eta|-1} \tilde{\Theta}'_\eta(a_\emptyset^i).$$

Example 3.2.2. Let $X = \{x_0, x_1, x_2\}$ and $\eta = x_0x_1$. Applying the derivation method gives

$$\begin{aligned} Sa^2_\eta =& (-1)^{|\eta|-1}\tilde{\Theta}'_\eta(a^2_\emptyset) \\ =& (-1)^{2-1}\tilde{\theta}'_{x_1} \circ \tilde{\theta}'_{x_0}(a^2_\emptyset) \\ =& -\tilde{\theta}'_{x_1}(-a^2_{x_0} + a^1_\emptyset a^2_{x_1} + a^2_\emptyset a^2_{x_2}) \\ =& -a^2_{x_0x_1} + a^1_{x_1}a^2_{x_1} + a^1_\emptyset a^2_{x_1x_1} + a^2_{x_1}a^2_{x_2} + a^2_\emptyset a^2_{x_2x_1}. \end{aligned}$$

This calculation agrees with that given in Example 3.1.3 for the coproduct methods, but it clearly requires fewer steps.

CHAPTER 4

ON THE PERFORMANCE OF ANTIPODE METHODS

4.1 DERIVATION IMPLEMENTATION IN MATHEMATICA

In this section an implementation in *Mathematica* is presented for the derivation method to compute the antipode of the output feedback Hopf algebra. To provide a benchmark for performance, code from [22] was modified for the coproduct methods by including the internal summation in Theorem 3.1.1 (3) for the multivariable case. In order to achieve a more robust handling of noncommutative algebraic manipulations than what is offered by the base *Mathematica* engine, the package *NCAlgebra* [29] was utilized. The functions presented here are fully reliant on the method in which noncommutative multiplication is treated by invoking the *NCAlgebra* package.

To validate the accuracy of the antipode code developed here, two types of unit tests were performed. The first collection of tests ensured the function generates valid results by using the antipode to calculate the inverse of a set of series. When composing the inverses with the original series, the expected result is the identity element. The second set of tests only computed the antipodes using this new code. The results were directly compared to those obtained from doing the same calculations using the previous coproduct method code, which was independently validated.

4.1.1 NOTATION

Special consideration must be taken in this environment to define the letters of the underlying alphabet X. In the SISO case, for example, the alphabet X is established by

```
X={x0,x1};
SetNonCommutative/@X;
```

which allows for both `x0` and `x1` to be treated as variables which do not commute under *Mathematica*'s explicitly defined noncommutative multiplication. Words are represented as products under this noncommutative operator, for example,

```
x0**x1**x0
```

represents $x_0x_1x_0 \in X^*$. The double asterisk operation specifies noncommutative multiplication. The empty word, $\emptyset$, is treated here as the monomial $1\emptyset$, which through a slight abuse of notation is represented in software by simply the real number `1`. The coordinate functions of H are each denoted with the head label `A` and two arguments: the first is the index $i \in \{1, 2, \ldots, m\}$ for the series c_i when $c \in \mathbb{R}^m\langle\langle X \rangle\rangle$, and the second is the associated word which acts as an index for the function. For example,

```
A[2,x0**x1]
```

represents the coordinate function $a^2_{x_0x_1}$. For the SISO case where $c \in \mathbb{R}\langle\langle X \rangle\rangle$, the first argument is 1 by default.

4.1.2 FUNCTIONS

With a structured means of representing the coordinate functions in H, it is necessary to build the functions which perform the operations as defined in the previous chapter.

Function 4.1.1. The implementation of the right-augmentation operator, $\tilde{\theta}_{x_i}$, is denoted by `RightAugment` and given below.

```
RightAugment[a_Plus,i_,x_List]:= Map[RightAugment[#,i,x]&,a]
RightAugment[a_Times,i_,x_List]:= RightAugment[a,i,x]=
  Sum[MapAt[RightAugment[#,i,x]&,a,ic], {ic,Length[a]}]
RightAugment[a_A^exp_,i_,x_List]:=
  exp*a^(exp-1)*RightAugment[a,i,x]
RightAugment[a_A,i_,x_List]:= A[a[[1]],a[[2]]**x[[i+1]]]
RightAugment[a_,i_,x_List]:= 0
```

- The first argument is any polynomial expression of coordinate functions, `a`, for which $a \in \mathbb{R}^\ell\langle X \rangle$.
- The second argument is the one-based[1] index, `i`, of the augmenting letter, $x_i \in X$, corresponding to its canonically ordered position in X.
- The last argument allows one to specify the alphabet X. Here, the list `x` is assumed to be canonically ordered.

[1] As opposed to the zero-based indexing more typically encountered in computer science contexts.

In order, the definitions in Function 4.1.1 are responsible for:

1. enforcing the linearity of the operator over addition, ensuring that it is correctly distributed over sums of coordinate functions;
2. making the operator act as a derivation on products (Liebniz's rule);
3. optimally extending the derivation rule for the cases where coordinate functions are repeated in a product, shortcutting the expansion of terms to just perform the power rule;
4. defining the actual augmentation operator $\tilde{\theta}_{x_i}(a^j_\eta) := a^j_{\eta x_i}$; and
5. handling the special case where $\tilde{\theta}_{x_i}(\mathbf{1}) := \mathbf{0}$.

Note also that the second definition evokes the technique of memoization[2] by caching its result each time it is called in order to avoid repeating deep recursions that have already been calculated.

Example 4.1.1. $\tilde{\theta}_{x_2}(a^1_{x_0})$ on an alphabet $X = \{x_0, x_1, x_2\}$ is computed by

```
RightAugment[A[1,x0],2,{x0,x1,x2}]
```

which returns the result $a^1_{x_0x_2}$ written in the form

```
A[1,x0**x2]
```

Function 4.1.2. The implementation of the augmentation operator in Equation (4), $\tilde{\theta}'_{x_i}$, is denoted by `ModRightAugment` and given below.

```
ModRightAugment[a_Plus,i_,x_List]:=
  Map[ModRightAugment[#,i,x]&,a]
ModRightAugment[a_,0,x_List]:= ModRightAugment[a,0,x]=
  -RightAugment[a,0,x]+Sum[A[ic,1]*
  RightAugment[a,ic,x],{ic,1,Length[x]-1}]
ModRightAugment[a_,i_,x_List]:= -RightAugment[a,i,x]
```

- The first argument is any polynomial expression of coordinate functions, `a`, for which $a \in \mathbb{R}^\ell\langle X\rangle$.

[2] A technique where a function's computations are stored, allowing subsequent calls to that function to avoid the time it takes to compute the result again [30].

- The second argument is the one-based index, `i`, of the augmenting letter, $x_i \in X$, corresponding to its canonically ordered position in X.
- The last argument allows one to specify the alphabet X. Here, the list `x` is assumed to be canonically ordered.

Function 4.1.2 consists of definitions which:

1. exercise the linearity of the operator over addition;
2. implement the rule $\tilde{\theta}'_{x_0}(a^j_\eta) = -\tilde{\theta}_{x_0}(a^j_\eta) + \sum_{k=1}^m a^k_\emptyset \tilde{\theta}_{x_k}(a^j_\eta) = -a^j_{\eta x_0} + \sum_{k=1}^m a^k_\emptyset a^j_{\eta x_k}$; and
3. implement $\tilde{\theta}'_{x_i}(a^j_\eta) := -\tilde{\theta}_{x_i}(a^j_\eta) = -a^j_{\eta x_i}$, $i \neq 0$.

Note here that there is no explicit rule given for $\tilde{\theta}'_{x_i}$ to act as a derivation. The derivation property is inherited algebraically as a consequence of $\tilde{\theta}'_{x_i}$ being defined linearly in terms of $\tilde{\theta}_{x_i}$. Just like the rules for the previous function, the second definition in this instance caches its result.

Function 4.1.3. The implementation of the Hopf algebra antipode computation, S, is denoted by `Antipode` and given below.

```
Antipode[A[s_,a_NonCommutativeMultiply],x_List]:=
 (-1)^(WordLength[a]-1)*
 Fold[ModRightAugment[#1,FirstPosition[x,#2][[1]]-1,x]&,A[s,1],a]
Antipode[A[s_,1],x_List]:= -A[s,1]
Antipode[a_A,x_List]:=
 ModRightAugment[A[a[[1]],1],FirstPosition[x,a[[2]]][[1]]-1,x]
```

- The first argument is any single coordinate function in H.
- The second is the alphabet X for the indexing word. The list `x` is assumed to be presented in a canonical order.

In order, the definitions in Function 4.1.3:

1. implement Theorem 3.2.1, namely, $Sa^j_\eta := (-1)^{|\eta|-1}\tilde{\Theta}'_\eta(a^j_\emptyset)$, with a functional composition of $\tilde{\theta}'_{x_i}$ operators;
2. handle the specific case of a coordinate function indexed by the empty word, $a^j_\emptyset$; and

3. address the case of a coordinate function for a single letter word, that is, $a^j_{x_i}$ for $i \in \{0, 1, \ldots, m\}$.

Example 4.1.2. The antipode $Sa^2_{x_0x_1}$ is calculated on the alphabet $X = \{x_0, x_1, x_2\}$ by

```
Antipode[A[2,x0**x1],{x0,x1,x2}]
```

which confirms the results given in Examples 3.1.3 and 3.2.2, namely,

```
A[1,x1]A[2,x1] + A[2,x1]A[2,x2] - A[2,x0**x1]
  + A[1,1]A[2,x1**x1] + A[2,1]A[2,x2**x1]
```

4.2 PERFORMANCE ANALYSIS

In this section, a performance comparison between the derivation method and the two coproduct methods is given using the *Mathematica* implementations described above. The code for the derivation and coproduct methods use very different data structures to represent the results. For this reason, a greater emphasis is placed on the timing benchmarks as a metric in making performance comparisons. In order to analyze the spatial results between all three methods, special considerations are taken to "normalize" the benchmarks based on the differences in series representations, as described in Section 4.2.2. The spatial data also serves to ensure that memory resources are not saturated during the calculations, which would skew or inflate the timing results obtained.

For consistency, all tests were run on the same Windows-based computer with a 2.70 GHz Intel Core i7-3740QM processor and 4×4 GB of 1600 MHz DDR3 SDRAM. *Mathematica* version 10.4 was used as well as excerpts of code from *NCAlgebra* version 4.0.6. These code excerpts were utilized by all three antipode methods tested.

Two sets of tests were run for each implementation. One set examined performance for a given alphabet $X_S := \{x_0, x_1\}$ corresponding to a SISO system, while the other performed calculations based on a space built on the alphabet $X_M := \{x_0, x_1, x_2\}$, which models a two-input, two-output system. For each test case, the antipode operation was applied to a single coordinate function. Across cases, the varied parameter was the degree of each coordinate function. Functions of the form $a^j_{x_0^i}$ are considered the worst case out of all functions of degree $2i + 1$, $i \geq 0$ because of the number of terms and recursions which they produce. The test cases were comprised exclusively of coordinate functions of this form

so as to determine an empircal upper-bound on performance with respect to functions of odd-numbered degrees.

Each test was run independently on the same machine. *Mathematica* caches results which have been previously calculated during a kernel session and calls upon those previously determined values if a function call with the same parameters is made again later. For this reason, a fresh kernel was initialized before each calculation. This avoids biases that would be present in recursive functions utilizing cached values from previous runs. Both the timing and memory data were obtained using *Mathematica*'s built-in performance metric tools. The tests were run on successively increasing degrees of coordinate functions up to and until a particular test case required memory utilization in excess of 10 GB, at which point the calculation was aborted and no larger degree functions were examined. Because of this resource restriction, not all data sets presented cover the same range of functional degrees. No limit for execution time was imposed on the tests. The included data points are averages of the results obtained from *Mathematica*'s built-in tools run over five trials in each test case.

TABLE II: Execution times (s) of antipode methods for X_S.

Degree	Derivation	Right Coproduct	Left Coproduct
1	0.000015	0.000133	0.000132
3	0.000131	0.000668	0.000573
5	0.000303	0.001243	0.002165
7	0.000918	0.002829	0.008646
9	0.003188	0.007377	0.045618
11	0.012189	0.020672	0.332750
13	0.045569	0.065496	2.937636
15	0.171627	0.221144	28.604094
17	0.683118	0.911619	312.933532
19	3.200415	3.779574	-
21	17.238338	20.226803	-
23	116.778690	132.333375	-
25	1028.466542	1069.589595	-

4.2.1 TIMING PERFORMANCE

First, the execution times (wall times) of the antipode calculation are compared for each method. The performances of all three methods on a SISO system with the alphabet $X_S = \{x_0, x_1\}$ are provided in Figure 10. Figure 11 plots the results for antipodes on the same coordinate functions with the alphabet $X_M = \{x_0, x_1, x_2\}$, modeling a two-input, two-output system. In Table II the timing results averaged over all tests for the SISO case are listed for comparison. For the test cases where execution times were greater than 1 second, the standard deviation of each method was less than 4% of the mean.

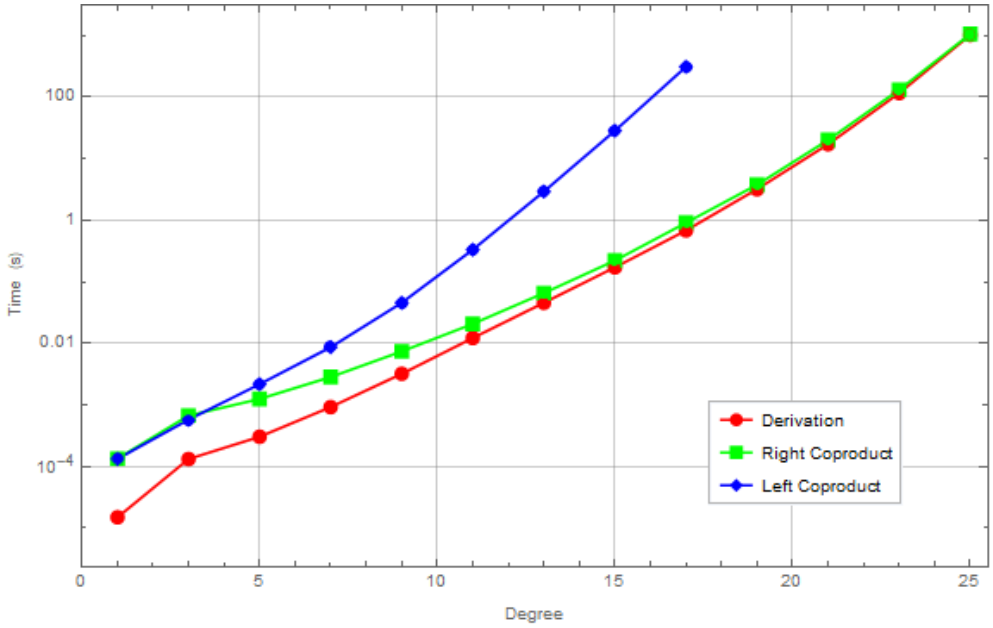


Fig. 10: Execution times of antipode methods for X_S.

Note that in the SISO case the relative difference in execution times between the derivation and right coproduct methods decreases as the problem size increases with the derivation method maintaining slightly better performance throughout. The left and right coproduct methods have roughly the same timings for degrees where there are relatively few cancellations in the left coproduct calculation (for example, there is one cancellation for degree five, 427 cancellations for degree 11 and 12,730 cancellations for degree 15 as shown in Table I). In the multi-input, multi-output (MIMO) case, the derivation method consistently outperforms the other methods in terms of speed. In fact, this trend was observed to hold for all MIMO systems up to five letters during tests not reported here.

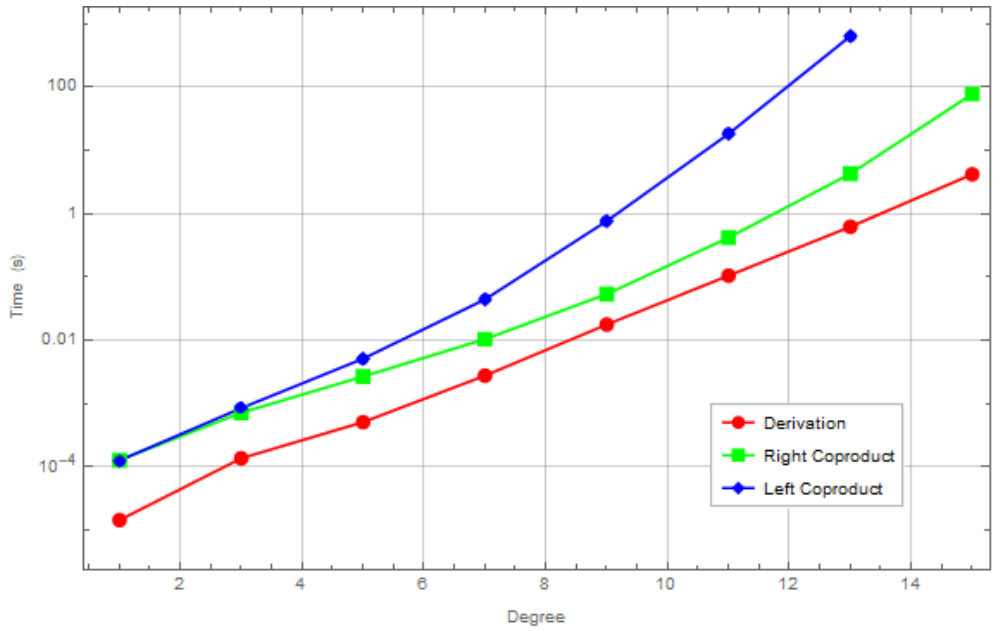


Fig. 11: Execution times of antipode methods for X_M.

4.2.2 SPATIAL PERFORMANCE

Next the memory utilization of each antipode method is presented. During the execution of each test, the total amount of system memory being utilized is monitored in real-time, and at the end of the calculation the peak usage is reported. This helps to validate the trends observed in the timing analysis by ensuring that the memory resources of the system were not saturated during the tests. Such a saturation would increase execution times due to increased memory management overhead. This would then create a discontinuity in the performance trend that would not be reflective of the algorithm's performance.

Additionally, one can identify from these results the consumption trends of each method; this allows for predictions on the size of the input that will result in the steepest performance decreases due to resource saturation. The results for the SISO system tests are presented in Figure 12, while Figure 13 gives the results for the MIMO system. For test cases utilizing more than 1 megabyte of memory, the standard deviation was less than 1% of the mean.

Observe that both the derivation and right coproduct methods have similar growth trends in the SISO case up to coordinate functions of degree 19. After that point the right coproduct method's consumption increases at a significantly faster rate than the derivation method. After degree 25, the right coproduct method was unable to complete the calculation

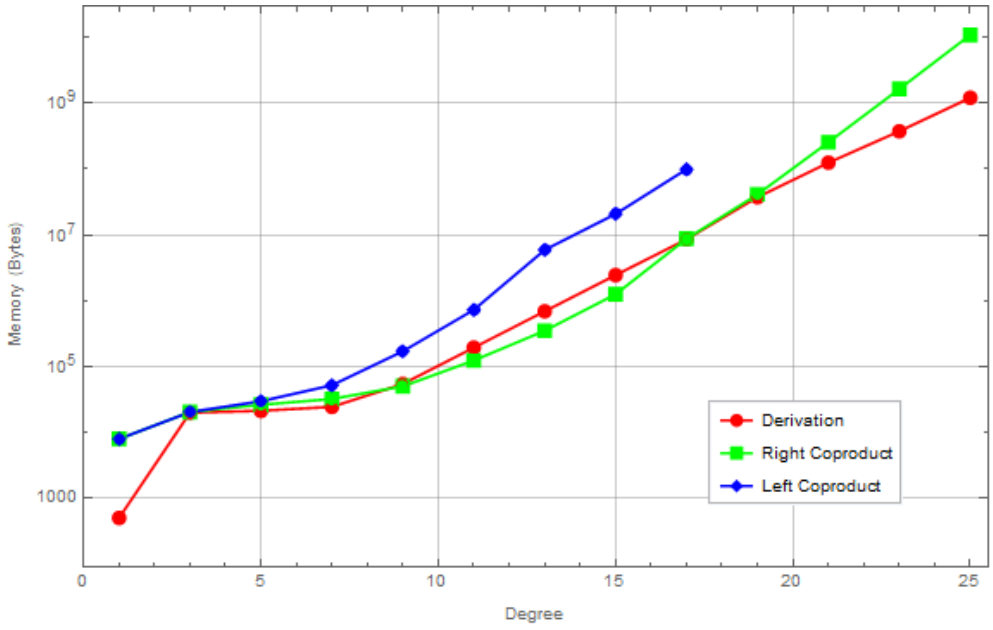


Fig. 12: Peak memory consumption of antipode methods for X_S.

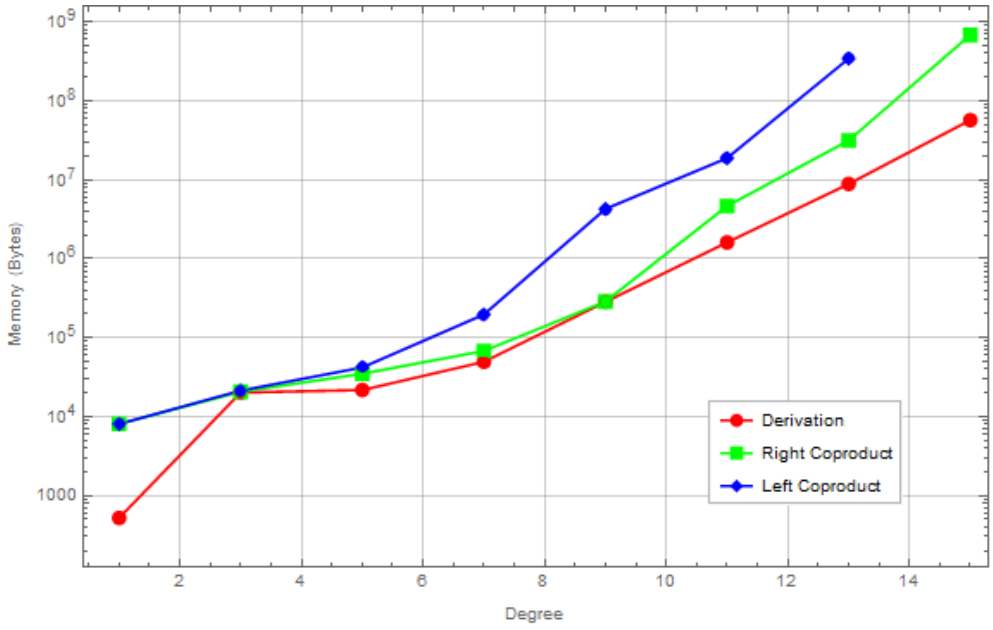


Fig. 13: Peak memory consumption of antipode methods for X_M.

under the imposed memory constraint.

The left coproduct method had the highest peak consumption in each SISO test most likely due to the excessive inter-term cancellations that it generates. For the MIMO case, the derivation method not only utilizes fewer resources but grows at a notably slower rate than the other methods. Therefore, it is able to compute higher degree antipodes than the other two methods when given the same memory constraints.

The sudden jump in memory usage from degree 1 to degree 3 in the derivation method is a consequence of the implementation rather than the algorithm itself. Degree 1 is written as a transformation consisting only of a sign change on the argument. Higher degree terms require recursive function calls which involve considerably more overhead, by comparison.

In addition to the peak interim memory usage, the spatial analysis is supplemented by also examining the resultant memory usage. This metric counts the size of information that persists after the calculation. This includes data such as the resulting solution's representation and cached interim values like those generated by memoization. By comparing the resource consumption observed under this metric and the previous one, a deeper insight into the efficiency of each algorithm is possible.

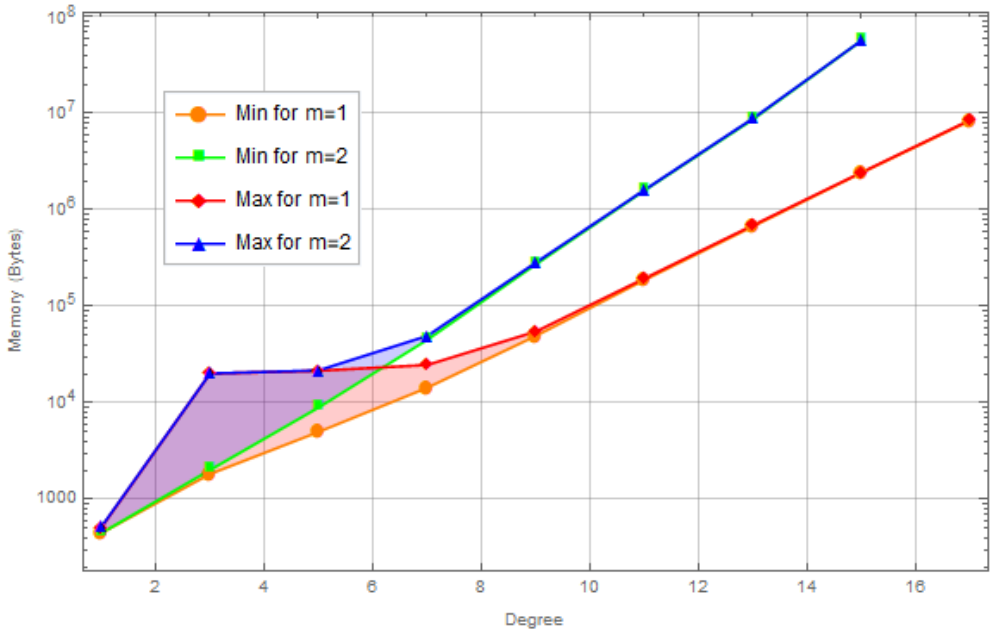


Fig. 14: Memory utilization for derivation implementation. The peak interim memory (Max) and the final memory (Min) for each alphabet are joined by shaded regions.

Figure 14 shows both the interim memory usage (max) and the final resultant memory usage (min) for the derivation method plotted together. The red shaded region joins together

both metrics for the SISO case, while the blue shaded region lies between the two metrics for the MIMO case. In both instances, these regions highlight the difference between the number of resources used to perform the calculation and the number needed to store the result. The interim and resultant resource counts quickly converge towards the same relative values as the degree increases. Two conclusions can be drawn from this observation: the first is that both the size of the result and the resources needed to calculate it have similar asymptotic bounds; the second is that there is relatively little extraneous overhead required to perform the calculation.

The left coproduct data is presented in a similar manner in Figure 15. Here the gap between resultant and total utilized memory remains tight as the size of the problem grows. From the log scale, it is evident that the relative difference remains more or less constant. This is somewhat surprising, even despite the optimizations described in [22], given that the left coproduct method is responsible for a rapidly growing number of cancellations as shown in Table I. In contrast, Figure 16 shows the right coproduct method trends tending toward similar bounds initially. Eventually, however, the intermediate memory usage follows a much steeper curve than the one observed for the resultant memory usage. This is again unexpected when compared to the behavior observed in the left coproduct method. It is worth highlighting that the diverging trend in the right coproduct method begins at about the same functional degree that the left coproduct method testing terminates. For this degree and higher, the left coproduct method could not calculate results without exceeding the imposed memory restriction. Therefore, more extensive examination of the left coproduct method's performance may reveal a similar or perhaps worse trend behavior.

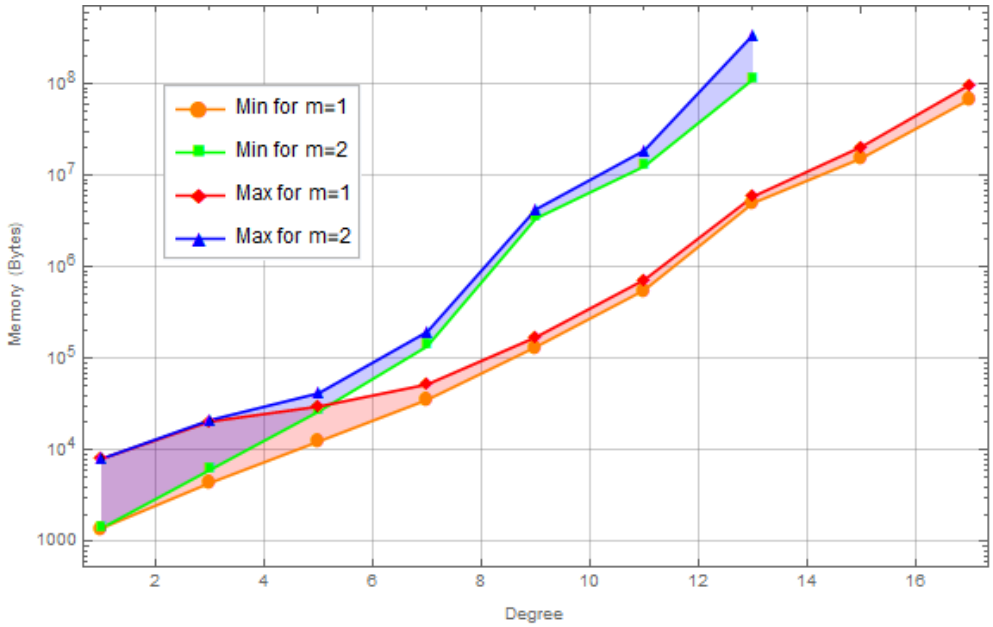


Fig. 15: Memory utilization for left coproduct implementation. The peak interim memory (Max) and the final memory (Min) for each alphabet are joined by shaded regions.

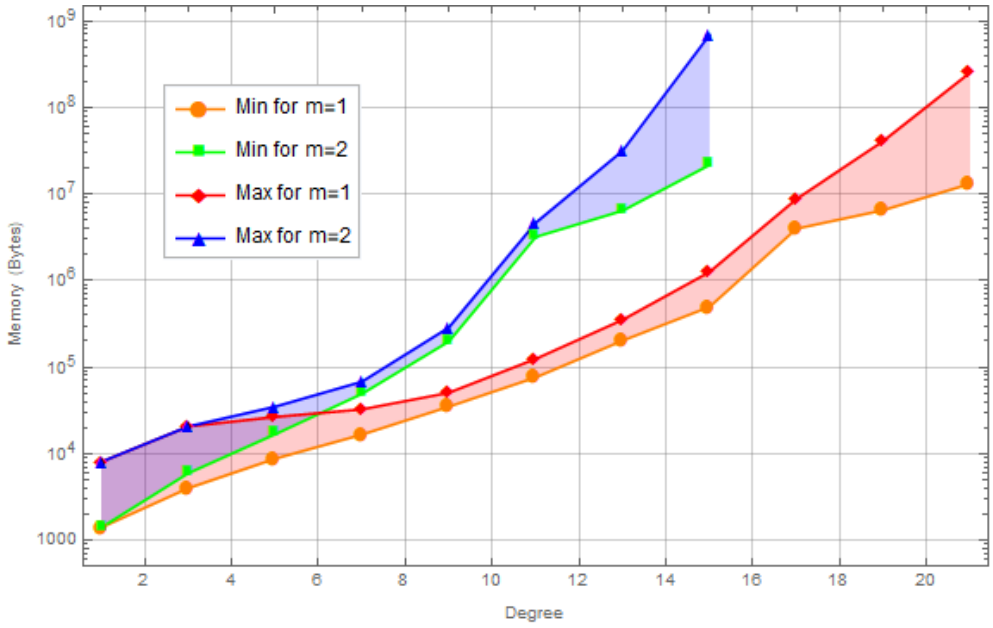


Fig. 16: Memory utilization for right coproduct implementation. The peak interim memory (Max) and the final memory (Min) for each alphabet are joined by shaded regions.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

The first goal of this thesis was to develop a *Mathematica* implementation of the output feedback Hopf algebra antipode using the method of derivations. This was done successfully as described in Chapter 4 and Appendix A.

The second goal was to determine how this approach performs in comparison to the previous methods. The results of this analysis were presented in Chapter 4. It was observed that for a Hopf algebra formed over an alphabet of two letters, the derivation algorithm has approximately the same time complexity as the more efficient of the two coproduct approaches. In the cases of larger alphabets, the derivation outperformed the other methods by a full order of magnitude.

Future research should include analytically determining asymptotic upper and lower bounds on the complexity behavior of the derivation approach. This would add additional evidence to support what was observed empirically in this project. In addition, certain optimizations may be possible regarding the derivations. A more rigorous analysis of the recursion formula may yield a simpler, closed form approach to calculating the derivative. Furthermore, the right coproduct method may also be worth reexamining with an implementation done on specialized hardware architectures in order to determine if it can be more viable for applications. Just as matrix operations excel on highly parallelized architectures like graphical processing units, the recent popularity of machine learning networks has driven the creation of a number of hardware platforms which specialize in tensor operations [31–33].

REFERENCES

[1] A. Isidori, *Nonlinear Control Systems*. Berlin, Heidelberg: Springer-Verlag, 3rd ed., 1995.

[2] R. W. Brockett, "Volterra series and geometric control theory," *Automatica*, vol. 12, no. 2, pp. 167–176, 1976.

[3] M. Lamnabhi, *Séries de Volterra et séries génératrices non commutatives*. Universite Paris-Sud, 1980.

[4] M. Fliess, "Fonctionnelles causales non linéaires et indéterminées non commutatives," *Bulletin de la Société Mathématique de France*, vol. 109, pp. 3–40, 1981.

[5] M. Fliess, "Réalisation locale des systèmes non linéaires, algèbres de lie filtrées transitives et séries génératrices non commutatives," *Inventiones mathematicae*, vol. 71, pp. 521–537, Mar 1983.

[6] M. Fliess, M. Lamnabhi, and F. Lamnabhi-Lagarrigue, "An algebraic approach to nonlinear functional expansions," *IEEE Transactions on Circuits and Systems*, vol. 30, no. 8, pp. 554–570, 1983.

[7] H. H. Nijmeijer, *Nonlinear dynamical control systems*. New York: Springer-Verlag, 1990.

[8] W. J. Rugh, *Nonlinear system theory the Volterra/Wiener approach*. Johns Hopkins studies in information science and systems, Baltimore: Johns Hopkins University Press, 1980.

[9] M. Schetzen, *The Volterra and Wiener theories of nonlinear systems*. New York: Wiley, 1980.

[10] V. Volterra, *Sopra le funzioni che dipendono da altre funzioni*. Tip. della R. Accademia dei Lincei, 1887.

[11] Y. Wang, "Algebraic differential equations and nonlinear control systems," 1990.

[12] W. S. Gray and Y. Li, "Generating series for interconnected analytic nonlinear systems," *SIAM Journal on Control and Optimization*, vol. 44, no. 2, pp. 646–672, 2005.

[13] I. M. Winter-Arboleda, W. S. Gray, and L. A. Duffaut Espinosa, "Fractional Fliess operators: Two approaches," in *2015 49th Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6, March 2015.

[14] W. Gray and Y. Wang, "Fliess operators on L_p spaces: convergence and continuity," *Systems and Control Letters*, vol. 46, no. 2, pp. 67–74, 2002.

[15] A. Ferfera, *Combinatoire du Monoïde Libre Appliquée à la Composition et aux Variations de Certaines Fonctionnelles Issues de la Théorie des Systèmes*. PhD thesis, University of Bordeaux I, 1979.

[16] A. Ferfera, "Combinatoire du monoïde libre et composition de certains systèmes non linéaires," in *Analyse des systèmes*, no. 75-76 in Astérisque, pp. 87–93, Société mathématique de France, 1980.

[17] W. S. Gray and M. Thitsa, "A unified approach to generating series for mixed cascades of analytic nonlinear input-output systems," *International Journal of Control - INT J CONTR*, vol. 85, pp. 1–18, 11 2012.

[18] L. A. Duffaut Espinosa, K. Ebrahimi-Fard, and W. S. Gray, "A combinatorial Hopf algebra for nonlinear output feedback control systems," *Journal of Algebra*, vol. 453, no. C, pp. 609–643, 2016.

[19] W. S. Gray and L. A. Duffaut Espinosa, "A Faà di Bruno Hopf algebra for a group of Fliess operators with applications to feedback," *Systems and Control Letters*, vol. 60, no. 7, pp. 441–449, 2011.

[20] W. S. Gray, L. A. Duffaut Espinosa, and K. Ebrahimi-Fard, "Faà di Bruno Hopf algebra of the output feedback group for multivariable Fliess operators," *Systems and Control Letters*, vol. 74, no. C, pp. 64–73, 2014.

[21] W. S. Gray, L. A. Duffaut Espinosa, and M. Thitsa, "Left inversion of analytic nonlinear SISO systems via formal power series methods," *Automatica*, vol. 50, no. 9, pp. 2381–2388, 2014.

[22] W. Gray, L. Duffaut Espinosa, and K. Ebrahimi-Fard, "Recursive algorithm for the antipode in the SISO feedback product," *Proc. 21st International Symposium on the Mathematical Theory of Networks and Systems*, pp. 1088–1093, 01 2014.

[23] K. Ebrahimi-Fard and W. S. Gray, "Center problem, Abel equation and the Faà di Bruno Hopf algebra for output feedback," *International Mathematics Research Notices*, vol. 2017, no. 17, pp. 5415–5450, 2017.

[24] D. E. Radford, *Hopf algebras*. Ebrary science and technology e-book collection, Hackensack, New Jersey: World Scientific, 2012.

[25] E. Abe, *Hopf Algebras*. Cambridge Tracts in Mathematics, Cambridge University Press, 2004.

[26] S. Dascalescu, C. Nastasescu, and S. Raianu, *Hopf Algebra: An Introduction*. Chapman & Hall/CRC Pure and Applied Mathematics, Taylor & Francis, 2000.

[27] M. Sweedler, *Hopf algebras*. Mathematics lecture note series, W. A. Benjamin, 1969.

[28] H. Figueroa and J. M. Gracia-Bondia, "Combinatorial Hopf algebras in quantum field theory I," *Reviews in Mathematical Physics*, vol. 17, no. 8, pp. 881–976, 2004.

[29] M. Stankus, J. W. Helton, and M. de Oliveira, "NCAlgebra." `https://math.ucsd.edu/~ncalg/`, 2019.

[30] T. H. Cormen, *Introduction to algorithms*, p. 387. Cambridge, Massachusetts: MIT Press, 3rd ed., 2009.

[31] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, pp. 21–29, Mar 2018.

[32] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-L. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. Mackean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *2017*

ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), vol. 128643, pp. 1–12, ACM, 2017.

[33] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, pp. 82–99, January 2018.

APPENDIX A

SOFTWARE DOCUMENTATION

NCFPS

Noncommutative Formal Power Series Commands for Mathematica

Lance Berlin, Maurício de Oliveira, Luis A. Duffaut Espinosa,
W. Steven Gray, Benjamin C. Greenberg, J. William Helton

November 11, 2019

0 Reserved Labels

- **A**
- **Description**

  For any $\eta \in X^*$, the coefficient, or coordinate, functions a_η^k are maps on formal series $c \in \mathbb{R}^m \langle\langle X \rangle\rangle$ yielding coeffcients

  $$a_\eta^k(c) := \langle c_k, \eta \rangle$$

  where $c_k = \Sigma_{\eta \in \langle X \rangle} \langle c_i, \eta \rangle \eta$

  This letter is reserved as the head for such functions and is used primarily with the functions `RhoRightAugment`, `RightAugment`, `MapCoordinateFunctions`, and `Antipode`.
- **Syntax**

  `A[index,word]`

  `index` is the k^{th} element of c and **`word`** is the noncommutative word, η, which index a coordinate function a_η^k.

  DO NOT include the series from which the function is to draw coefficients. Instead use the function MapCoordinate-Functions. This label exists for algebraic purposes.
- **Example**

```
A[2,x0**x1]
```

```
A[2, x0 ** x1]
```

1 Elementary Commands

- **CharacteristicSeries**
- **Description**

  Given a subset $L \subseteq X^*$, the characteristic series of L is the element in $\mathbb{R}\langle\langle X \rangle\rangle$ defined by $\text{char}(L) = \sum_{v \in L} v$.

- **Syntax**

  `CharacteristicSeries[list]`

  `list` is a list of words describing the language L.

- **Example**

```
L={x0**x0,x1,x2**x1};
CharacteristicSeries[L]
```

```
x1 + x0 ** x0 + x2 ** x1
```

FirstLetter

- **Description**

  Given $\eta = x_{i_1} x_{i_2} \cdots x_{i_k} \in X^*$, the first letter of η is x_{i_1} . The definition is extended linearly to series.

- **Syntax**

  `FirstLetter[series]`

  `series` is noncommutative formal power series.

- **Example**

```
FirstLetter[x1**x0+2x1**x0**x0]
```

```
3 x1
```

ImproperPart

- **Description**

  The improper part of a series $c \in \mathbb{R}^{\ell}\langle\langle X \rangle\rangle$ is $(c, \emptyset)$. The current implementation assumes $\ell = 1$.

- **Syntax**

  `ImproperPart[series]`

  `series` is a noncommutative formal power series.

- **Example**

```
ImproperPart[1+x0+x0**x0+x0**x0**x0]
```

```
1
```

LanguageFilter

- **Description**

  The polynomial c^* is the largest polynomial in $c \in \mathbb{R}^{\ell}\langle\langle X \rangle\rangle$ such that $\text{supp}(c^*) \subseteq L \subset X^*$. The current implementation assumes $\ell = 1$.

- **Syntax**

  `LanguageFilter[c,L]`

  returns the largest polynomial in the noncommutative series **`c`** such that the support of that polynomial is a subset of the language **`L`**.

  `LanguageFilter[c,Y,X]`

  returns the largest polynomial in the noncommutative series **`c`** such that the support of that polynomial is a subset of

the language $\mathbf{Y}^* \subset \mathbf{X}^*$, where **Y** and **X** are each noncommutative alphabets such that $\mathbf{Y} \subset \mathbf{X}$.

- **Example**

```
c=2+x0-3x1**x0+4x0**x1**x0;
lang={1,x0,x1,x1**x0};
LanguageFilter[c,lang]
```

```
2 + x0 - 3 x1 ** x0
```

```
c=2+x0+3x1**x0-4x2**x0;
y={x0,x1};
x={x0,x1,x2};
LanguageFilter[c,y,x]
```

```
2 + x0 + 3 x1 ** x0
```

NCDegree

- **Description**

Given $p \in \mathbb{R}^{\ell}\langle X\rangle$, the degree of p is $\deg(p) = \max\{|\eta| : \eta \in \mathrm{supp}(p)\}$. By definition, $\deg(0) = -\infty$. The partial degree with respect to $x_i \in X$ is $\deg_{x_i}(p) = \max\{|\eta|_{x_i} : \eta \in \mathrm{supp}(p)\}$. By definition, $\deg_{x_i}(0) = -\infty$. The current implementation assumes $\ell = 1$.

- **Syntax**

NCDegree[**polynomial**]

polynomial is a non-commutative formal polynomial.

NCDegree[**polynomial,symbol**]

symbol is a letter in a noncommutative alphabet.

- **Example**

```
NCDegree[x1**x0**x0**x1+x1**x1**x1]
```

```
4
```

```
NCDegree[x1**x0**x0**x1+x1**x1**x1,x1]
```

```
3
```

NCOrder

- **Description**

Given $c \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$, the order of c is $\mathrm{ord}(c) = \min\{|\eta| : \eta \in \mathrm{supp}(c)\}$. By definition, $\mathrm{ord}(0) = \infty$. The partial order with respect to $x_i \in X$ is $\mathrm{ord}_{x_i}(c) = \min\{|\eta|_{x_i} : \eta \in \mathrm{supp}(c)\}$. By definition, $\mathrm{ord}_{x_i}(0) = \infty$. The current implementation assumes $\ell = 1$.

- **Syntax**

NCOrder[**series**]

series is a noncommutative formal power series.

NCOrder[**series,symbol**]

symbol is a letter in a noncommutative alphabet.

- **Example**

```
NCOrder[x1**x0**x0**x1+x1**x1**x1]
```

```
3
```

```
NCOrder[x1**x0**x0**x1+x1**x1**x1,x1]
```

```
2
```

- **NCPrefix**
- **Description**

Given words $\eta, \nu \in X^*$, the word η is a prefix of the word $\eta\nu$.

- **Syntax**

NCPrefix[**monomial**,**list**]

monomial is a noncommutative monomial, and **list** is a list of noncommutative monomials.

- **Example**

```
NCPrefix[x1**x0,{x1,x0**x1}]
```

```
{x1 ** x0 ** x1, x1 ** x0 ** x0 ** x1}
```

- **ProperPart**
- **Description**

The proper part of a series $c \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$ is $c' = c$ - $(c, \emptyset)$. The current implementation assumes $\ell = 1$.

- **Syntax**

ProperPart[**series**]

series is a noncommutative formal power series.

- **Example**

```
ProperPart[1+x0+x0**x0+x0**x0**x0]
```

```
x0 + x0 ** x0 + x0 ** x0 ** x0
```

- **ProperQ**
- **Description**

The proper part of a series $c \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$ is $c' = c - (c, \emptyset)$. The current implementation assumes $\ell = 1$.

- **Syntax**

ProperQ[**series**]

series is a noncommutative formal power series.

- **Example**

```
ProperQ[1+x0+x0**x0+x0**x0**x0]
```

```
False
```

RelativeDegree

Description

Suppose $X = \{x_0, x_1, \ldots, x_m\}$. Any series $c \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$ can be written as $c = c_N + c_F$, where $c_N := \sum_{k\geq 0}(c, x_0^k)\, x_0^k$ and $c_F := c - c_N$. The distinguished letter x_0 is called the drift letter. For each component series $c_i \in \mathbb{R}\langle\langle X\rangle\rangle$, $i = 1, 2, \ldots, \ell$, let $r_i \geq 1$ be the largest integer such that $\mathrm{supp}(c_{i,F}) \subseteq x_0^{r_i-1}X^*$. Then c_i has *relative degree* r_i if the linear word $x_0^{r_i-1}x_j \in \mathrm{supp}(c_i)$ for some $j \in \{0, 1, \ldots, m\}$, otherwise it is not well defined. In addition, c has (vector) relative degree $\{r_1, r_2, \ldots, r_\ell\}$ if the $\ell \times m$ matrix

$$A = \begin{pmatrix} (c_1, x_0^{r_1-1}x_1) & (c_1, x_0^{r_1-1}x_2) & \cdots & (c_1, x_0^{r_1-1}x_m) \\ (c_2, x_0^{r_2-1}x_1) & (c_1, x_0^{r_2-1}x_2) & \cdots & (c_1, x_0^{r_1-1}x_m) \\ \vdots & \vdots & \ddots & \vdots \\ (c_\ell, x_0^{r_\ell-1}x_1) & (c_\ell, x_0^{r_\ell-1}x_2) & \cdots & (c_\ell, x_0^{r_\ell-1}x_m) \end{pmatrix}$$

has full rank. Otherwise, c does not have relative degree. By far, the most common occurrence of this definition is for the case where $\ell = m$. In which case, the condition on A is equivalent to $\det(A) \neq 0$.

Syntax

RelativeDegree[series,list,symbol]

series is a noncommutative formal power series in the alphabet defined by **list**. **symbol** is the drift letter.

Example

```
RelativeDegree[x0+x0**x1+x0**x0**x1+x0**x0**x1**x0+x0**x1**x0**x1,{x0,x1},x0]
```

```
{2}
```

RhoRightAugment

Description

Let $\tilde{\theta}_i^{\rho}$ denote the antipodal right-augmentation operator and $X = \{x_0, x_1, \ldots, x_m\}$. This mapping operates on coordinate functions (see A) according to

$$\tilde{\theta}_i^{\rho}(a_\eta^k) := -\tilde{\theta}_i(a_\eta^k) \quad \forall\ i \neq 0$$
$$\tilde{\theta}_0^{\rho}(a_\eta^k) := -\tilde{\theta}_0(a_\eta^k) + \sum_{r=1}^{m} a_\emptyset^r\, \tilde{\theta}_r(a_\eta^k)$$

See RightAugment.

This operator acts as a derivation

$$\tilde{\theta}_i^{\rho}(a_{\eta_1}^{k_1} \cdots a_{\eta_s}^{k_s}) := \sum_{j=1}^{s} a_{\eta_1}^{k_1} \cdots \tilde{\theta}_i^{\rho}(a_{\eta_j}^{k_j}) \cdots a_{\eta_s}^{k_s}$$

Syntax

RhoRightAugment[cfunc,ind,alph]

cfunc is a coordinate function of the type described in A, **ind** is the index of the corresponding letter in the alphabet **alph**.

Example

```
X={x0,x1};
RhoRightAugment[A[1,x0**x1],0,X]
```

```
-A[1, x0 ** x1 ** x0] + A[1, 1] × A[1, x0 ** x1 ** x1]
```

RightAugment

Description

Let $\tilde{\theta}_i$ denote the right-augmentation operator and $X = \{x_0, x_1, \ldots, x_m\}$. This endomorphism adds letters to the right of a specified coordinate function (see **A**)

$$\tilde{\theta}_i\left(a_\eta^k\right) := a_{\eta x_i}^k$$
$$\tilde{\theta}_i(1) = 0$$

May need to overhaul to implement the following functionality:
For a word $\eta := x_{i_1} \cdots x_{i_r}$ and an arbitrary word $\psi \in X^*$

$$\tilde{\theta}_\eta\left(a_\eta^k\right) = \tilde{\theta}_{i_r} \circ \cdots \circ \tilde{\theta}_{i_1}\left(a_\psi^k\right)$$

This operator acts as a derivation

$$\tilde{\theta}_i\left(a_{\eta_1}^{k_1} \cdots a_{\eta_s}^{k_s}\right) := \sum_{j=1}^{s} a_{\eta_1}^{k_1} \cdots a_{\eta_j x_i}^{k_j} \cdots a_{\eta_s}^{k_s}$$

Syntax

RightAugment[**cfunc,ind,alph**]

cfunc is a coordinate function of the type described in **A**, **ind** is the index of the corresponding letter in the alphabet **alph**.

Example

```
X={x0,x1};
RightAugment[A[1,x0**x1],0,X]
```

```
A[1, x0 ** x1 ** x0]
```

Support

Description

Given $c \in \mathbb{R}^\ell\langle\langle X\rangle\rangle$, the support of c is supp(c) = $\{\eta \in X^* : (c, \eta) \neq 0\}$. The current implementation $\ell = 1$.

Syntax

Support[**series**]

series is noncommutative formal power series.

Example

```
Support[x1**x0+2x1**x0**x0]
```

```
{x1 ** x0, x1 ** x0 ** x0}
```

TruncateSeries

Description

A series $c \in \mathbb{R}^\ell\langle\langle X\rangle\rangle$ is truncated to a polynomial in $\mathbb{R}^\ell\langle X\rangle$ of degree n. The current implementation assumes $m = \ell = 1$.

Syntax

TruncateSeries[**series,integer**]

series is a noncommutative formal power series, and **integer** is a nonnegative integer.

■ **Example**

```
TruncateSeries[1+x0+x0**x0+x0**x0**x0,2]
```

```
1 + x0 + x0 ** x0
```

■ **WordLength**

■ **Description**

Given $\eta \in X^*$, the word length, $|\eta|$, is the number of symbols in η. The word length with respect to $x_i \in X$, $|\eta|_{x_i}$, is the number of symbols in η that match x_i.

■ **Syntax**

`WordLength[list]`

`list` is a list of monic noncommutative monomials.

`WordLength[list,symbol]`

`symbol` is a letter in X.

■ **Example**

```
WordLength[{x1**x0**x0**x1,1,x1**x1**x0}]
```

```
{4, 0, 3}
```

```
WordLength[{x1**x0**x0**x1,1,x1**x1**x0},x0]
```

```
{2, 0, 1}
```

2 Unitary Commands

■ **Antipode**

■ **Description**

For a word $\eta := x_{i_1} \cdots x_{i_r} \neq \emptyset$, the antipode S can be written

$$S(a_\eta^k) = (-1)^{r-1}\, \tilde{\Theta}_\eta^\rho(a_\emptyset^k)$$

where

$$\tilde{\Theta}_\eta^\rho := \tilde{\theta}_{i_r}^\rho \circ \cdots \circ \tilde{\theta}_{i_1}^\rho$$

See `RhoRightAugment`.

■ **Syntax**

`Antipode[cfunc,alph]`

`cfunc` is a coordinate function of the type described in `A`, **`alph`** is the alphabet for the context being operated in.

■ **Example**

```
X={x0,x1};
Antipode[A[1,x0**x1],X]
```

```
A[1, x1]^2 - A[1, x0 ** x1] + A[1, 1] A[1, x1 ** x1]
```

■ GlobalGrowthConstants

■ **Description**

A series $c \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$ is said to be *globally convergent* if there exists real numbers $K,M > 0$ such that $|(c, \eta)| \leq KM^{|\eta|}$ for all $\eta \in X^*$, where $|z| = \max_i |z_i|$ when $z \in \mathbb{R}^{\ell}$, and $|\eta|$ denotes the length of the word η. The smallest such constants are called the *global growth constants*. When possible, this command returns the minimal K,M (in this order) and plots $\log(|(c, \eta)|)$ versus word length, as well as the uniform bound $\log(M)|\eta| + \log(K)$.

Assume there exists an expression such that $|(c, \eta)| \leq KM_0^{|\eta|_{x_0}} M_1^{|\eta|_{x_1}} \cdots M_m^{|\eta|_{x_m}}$ for all $\eta \in X^*$ and $X = \{x_0, x_1, \ldots, x_m\}$. When including a list alphabet, the command returns the minimal $K,M_0,M_1,\ldots,M_m$ (in this order) as well as showing a table of error values for the linearized fit of the form $\log(|(c, \eta)|) = \log(K) + \log(M_0)|\eta|_{x_0} + \log(M_1)|\eta|_{x_1} + \cdots + \log(M_m)|\eta|_{x_m}$

■ **Syntax**

GlobalGrowthConstants[series]

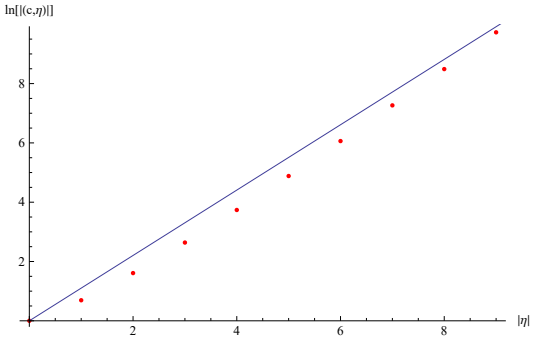
series is a noncommutative formal power series.

GlobalGrowthConstants[series,list]

list is a list containing an alphabet of letters.

■ **Example**

```
c = CharacteristicSeries[Table[CatalanNumber[n],{n,10}]*KleeneStar[{x0},9]];
GlobalGrowthConstants[c]
```




```
{1., 3.01008}
```

```
c=2+3x0+5x1+7x0**x0+9x0**x1+11x1**x0+12x0**x1**x0+15x0**x0**x0;
X={x0,x1};
GlobalGrowthConstants[c,X]
```

	Estimate	Standard Error	t-Statistic	P-Value
1	0.6854	0.158152	4.33381	0.0074724
x0	0.617353	0.0790759	7.8071	0.000552573
x1	0.869613	0.15313	5.67893	0.00235791

{1.98457, 1.85401, 2.38599}

KleeneStar

Description

Given languages $L, \overline{L} \subseteq$, define the product $L\overline{L} = \{\eta\overline{\eta} \in X^* : \eta \in L, \overline{\eta} \in \overline{L}\}$. The Kleene star of L is $L^* = \sum_{k=0}^{\infty} L^k$ where L^0 denotes the empty word, and $L^{k+1} = LL^k$. In practice, only the first $n + 1$ terms of the summation can be computed, where $n \geq 0$.

Syntax

KleeneStar[list,integer]

list is a list of monic monomials, and **integer** is a nonnegative integer.

Example

```
KleeneStar[{x0,x1},2]
```

{1, x0, x1, x0 ** x0, x0 ** x1, x1 ** x0, x1 ** x1}

LeftShift

Description

Given any $\xi \in X^*$, the corresponding left-shift operator on X^* is defined as

$$\xi^{-1} : X^* \to \mathbb{R}\langle X\rangle : \eta \mapsto \begin{cases} \eta' & \eta = \xi\eta' \\ 0 & \text{otherwise.} \end{cases}$$

Note that in the second half of this definition, η is being mapped to the zero polynomial, i.e., $p = 0$, as opposed to the empty word $\emptyset$. So this operator is a mapping into $\mathbb{R}\langle X\rangle$ and not into X^*. For any $c \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$, this definition is extended linearly as

$$\xi^{-1}(c) = \sum_{\eta \in X^*} (c,\eta)\, \xi^{-1}(\eta) = \sum_{\eta \in X^*} (c,\xi\eta)\eta.$$

The current implementation assumes $\ell = 1$.

Syntax

LeftShift[series,monomial]

series is a noncommutative formal power series, and **monomial** is a monic noncommutative monomial.

LeftShift[series,monomial,power]

power is a nonnegative integer.

■ **Example**

```
LeftShift[2x0**x1+x0**x1**x0+x1**x0**x0+x0**x0**x0,x0**x1]
```

```
2 + x0
```

```
LeftShift[x0**x1**x0**x1**x0**x0,x0**x1,2]
```

```
x0 ** x0
```

```
LeftShift[x0**x1**x0**x1**x0**x0,x0**x1,0]
```

```
x0 ** x1 ** x0 ** x1 ** x0 ** x0
```

■ LocalGrowthConstants

■ **Description**

A series $c \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$ is said to be *locally convergent* if there exists real numbers $K,M > 0$ such that $|(c, \eta)| \leq KM^{|\eta|}|\eta|!$ for all $\eta \in X^*$, where $|z| = \max_i |z_i|$ when $z \in \mathbb{R}^{\ell}$, and $|\eta|$ denotes the length of the word η. The smallest such constants are called the *local growth constants*. When possible, this command returns the minimal K,M (in this order) and plots $\log(|(c,\eta)|/|\eta|!)$ versus word length, as well as the uniform bound $\log(M)\,|\eta| + \log(K)$.

Assume there exists an expression such that $|(c, \eta)| \leq KM_0^{|\eta|_{x_0}} M_1^{|\eta|_{x_1}} \cdots M_m^{|\eta|_{x_m}} |\eta|_{x_0}!|\eta|_{x_1}!\cdots|\eta|_{x_m}!$ for all $\eta \in X^*$ and $X = \{x_0, x_1, \ldots, x_m\}$. When including a list alphabet, the command returns the minimal $K,M_0,M_1,\ldots,M_m$ (in this order) as well as showing a table of error values for the linearized fit of the form $\log[|(c, \eta)|/(|\eta|_{x_0}|\eta|_{x_1}\cdots|\eta|_{x_m})] = \log(K) + \log(M_0)|\eta|_{x_0} + \log(M_1)|\eta|_{x_1} + \cdots + \log(M_m)|\eta|_{x_m}$

■ **Syntax**

LocalGrowthConstants[**series**]

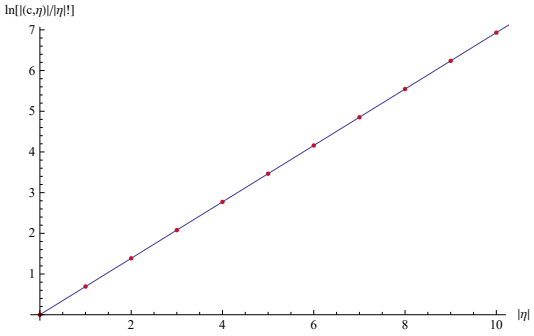
series is a noncommutative formal power series.

LocalGrowthConstants[**series,list**]

list is a list containing an alphabet of letters.

■ **Example**

```
c = ShuffleInverse[1-2x0,10];
LocalGrowthConstants[c]
```




```
{1., 2.}
```

```
c=2+3x0+5x1+7x0**x0+9x0**x1+11x1**x0+12x0**x1**x0+15x0**x0**x0;
X={x0,x1};
LocalGrowthConstants[c,X]
```

	Estimate	Standard Error	t-Statistic	P-Value
1	0.97384	0.339656	2.86713	0.0351121
x0	−0.113217	0.169828	−0.666658	0.534514
x1	1.00113	0.328871	3.04414	0.028615

```
{2.64809, 0.892957, 2.72136}
```

■ **ShuffleInverse**

■ **Description**

Let $c \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$ with $(c, \emptyset) \neq 0$. The shuffle inverse of c is

$$c^{\text{ш}\,-1} = \big((c,\ \emptyset)\,(1\ -\ c')\big)^{\text{ш}\,-1} = (c,\ \emptyset)^{-1} \sum_{k=0}^{\infty} (c')^{\text{ш}\,k}.$$

The inverse is approximated to order $n \geq 0$ if the upper bound of the summation above is n. The current implementation assumes $\ell = 1$.

■ **Syntax**

`ShuffleInverse[series,integer]`

`series` is a noncommutative formal power series, and **`integer`** is a nonnegative integer.

■ **Example**

```
c = 1+x0+x1;
cinv = NCExpand[ShuffleInverse[c,3]]
NCExpand[ShuffleProduct[c,cinv]]
```

```
1 - x0 - x1 + 2 x0 ** x0 + 2 x0 ** x1 + 2 x1 ** x0 + 2 x1 ** x1 - 6 x0 ** x0 ** x0 - 6 x0 ** x0 ** x1 -
 6 x0 ** x1 ** x0 - 6 x0 ** x1 ** x1 - 6 x1 ** x0 ** x0 - 6 x1 ** x0 ** x1 - 6 x1 ** x1 ** x0 - 6 x1 ** x1 ** x1
1 - 24 x0 ** x0 ** x0 ** x0 - 24 x0 ** x0 ** x0 ** x1 - 24 x0 ** x0 ** x1 ** x0 - 24 x0 ** x0 ** x1 ** x1 -
 24 x0 ** x1 ** x0 ** x0 - 24 x0 ** x1 ** x0 ** x1 - 24 x0 ** x1 ** x1 ** x0 - 24 x0 ** x1 ** x1 ** x1 -
 24 x1 ** x0 ** x0 ** x0 - 24 x1 ** x0 ** x0 ** x1 - 24 x1 ** x0 ** x1 ** x0 - 24 x1 ** x0 ** x1 ** x1 -
 24 x1 ** x1 ** x0 ** x0 - 24 x1 ** x1 ** x0 ** x1 - 24 x1 ** x1 ** x1 ** x0 - 24 x1 ** x1 ** x1 ** x1
```

3 Binary Commands

■ **CompositionProduct**

■ **Description**

Suppose $X = \{x_0, x_1, \ldots, x_m\}$, $\tilde{X} = \{\tilde{x}_0, \tilde{x}_1, \ldots, \tilde{x}_{\tilde{m}}\}$, $c \in \mathbb{R}^{\tilde{\ell}}\langle\langle \tilde{X}\rangle\rangle$ and $d \in \mathbb{R}^{\tilde{m}}\langle\langle X\rangle\rangle$. Define the family of mappings

$$D_{\tilde{x}_i} : \mathbb{R}\langle\langle X\rangle\rangle \to \mathbb{R}\langle\langle X\rangle\rangle : e \mapsto x_0(d_i \text{ ш } e),$$

$i = 0, 1, \ldots, \tilde{m}$ with $d_0 = 1$. Assume $D_\emptyset$ is the identity map on $\mathbb{R}\langle\langle X\rangle\rangle$ and compose these operators in the obvious way so that $D_{\tilde{x}_i \tilde{x}_j} =: D_{\tilde{x}_i} D_{\tilde{x}_j}$. The composition of a word $\eta \in \tilde{X}$ and d is defined as

$$\tilde{\eta} \circ d = (\tilde{x}_{i_k} \tilde{x}_{i_{k-1}} \cdots \tilde{x}_{i_1}) \circ d = D_{\tilde{x}_{i_k}} D_{\tilde{x}_{i_{k-1}}} \cdots D_{\tilde{x}_{i_1}}(1) = D_{\tilde{\eta}}(1).$$

In which case, the composition product of c and d is

$$c \circ d = \sum_{\tilde{\eta} \in \tilde{X}^*} (c, \tilde{\eta})\, \tilde{\eta} \circ d.$$

The composition power is defined as

$$c^{\circ i} = c \circ c \circ \cdots \circ c \ ,$$

where c appears i times and $c^{\circ 0} := 1$.

The current implementation assumes $\tilde{X} = X$ and $\tilde{\ell} = 1$.

■ **Syntax**

CompositionProduct[series,list1,list2]

series is a noncommutative formal power series, **list1** is a list of noncommutative formal power series, and **list2** is a list of symbols defining an alphabet.

CompositionProduct[series,list2,power]

power is a nonnegative integer.

■ **Example**

```
X = {x0,x1,x2};
c = x1**x2;
d = {x0+2x1,x2};
CompositionProduct[c,d,X]
```

```
2 x0 ** x0 ** x0 ** x2 + 2 x0 ** x0 ** x1 ** x2 +
 x0 ** x0 ** x2 ** x0 + 2 x0 ** x0 ** x2 ** x1 + 2 x0 ** x1 ** x0 ** x2
```

```
CompositionProduct[x0**x1,{x0,x1},2]
```

```
x0 ** x0 ** x0 ** x1
```

■ MapCoordinateFunctions

■ **Description**

This function takes an expression of coordinate functions and applies them to a noncommutative formal power series.

■ **Syntax**

`MapCoordinateFunctions[cexp,series]`

`cexp` is an expression of coordinate functions (see `A`), **`series`** is any noncommutative formal power series.

NOTE: It is very important that **`cexp`** does not contain any coordinate functions of words whose lengths are greater than the degree of **`series`**. This is because words which are not encountered in series are taken to have coefficients of 0. If **`series`** has been truncated such that **NCDegree[series]** evaluates to **n** and **`cexp`** contains a coordinate function **A[word]** such that **WordLength[word]>n** is **True**, then a cancellation of nonzero coefficients will occur. It is, however, safe for **`series`** to be of a greater degree than the largest word of any coordinte function in **`cexp`**.

■ **Example**

```
c=2+x0-x1**x0;
MapCoordinateFunctions[A[1,1]^A[1,x1**x0]+A[1,x1],c]
```

```
-2
```

■ ModifiedCompositionProduct

■ **Description**

Suppose $X = \{x_0, x_1, \ldots, x_m\}$, $\tilde{X} = \{\tilde{x}_0, \tilde{x}_1, \ldots, \tilde{x}_{\tilde{m}}\}$, $c \in \mathbb{R}^{\tilde{l}}\langle\langle \tilde{X}\rangle\rangle$ and $d \in \mathbb{R}^{\tilde{m}}\langle\langle X\rangle\rangle$. Define the family of mappings

$$\tilde{D}_{\tilde{x}_i} : \mathbb{R}\langle\langle X\rangle\rangle \to \mathbb{R}\langle\langle X\rangle\rangle : e \mapsto x_i e + x_0(d_i \sqcup\!\sqcup e),$$

$i = 0, 1, \ldots, \tilde{m}$ with $d_0 := 0$. Assume $\tilde{D}_{\emptyset}$ is the identity map on $\mathbb{R}\langle\langle X\rangle\rangle$ and compose these operators in the obvious way so that $\tilde{D}_{\tilde{x}_i \tilde{x}_j} =: \tilde{D}_{\tilde{x}_i}\tilde{D}_{\tilde{x}_j}$. The composition of a word $\tilde{\eta} \in \tilde{X}$ and d is defined as

$$\tilde{\eta} \circledcirc d = (\tilde{x}_{i_k}\tilde{x}_{i_{k-1}}\cdots\tilde{x}_{i_1}) \circledcirc d = \tilde{D}_{\tilde{x}_{i_k}}\tilde{D}_{\tilde{x}_{i_{k-1}}}\cdots\tilde{D}_{\tilde{x}_{i_1}}(1) = \tilde{D}_{\tilde{\eta}}(1).$$

In which case, the composition product of c and d is

$$c \circledcirc d = \sum_{\tilde{\eta}\in\tilde{X}^*}(c, \tilde{\eta})\,\tilde{\eta} \circledcirc d.$$

The modified composition power is defined as

$$c^{\circledcirc i} = c \circledcirc c \circledcirc \cdots \circledcirc c\ ,$$

where c appears i times and $c^{\tilde{\circ} 0} := 1$.

The current implementation assumes $\tilde{X} = X$ and $\tilde{\ell} = 1$.

- **Syntax**

`ModifiedCompositionProduct[series,list1,list2]`

`series` is a noncommutative formal power series, **`list1`** is a list of noncommutatie formal power series, and **`list2`** is a list of symbols defining an alphabet.

`ModifiedCompositionProduct[series,list2,power]`

`power` is a nonnegative integer.

- **Example**

```
X = {x0,x1,x2};
c = x1**x2;
d = {x0+2x1,x2};
ModifiedCompositionProduct[c,d,X]
```

```
x1 ** x2 + x0 ** x0 ** x2 + 2 x0 ** x1 ** x2 + x0 ** x2 ** x0 +
 2 x0 ** x2 ** x1 + x1 ** x0 ** x2 + 2 x0 ** x0 ** x0 ** x2 + 2 x0 ** x0 ** x1 ** x2 +
 x0 ** x0 ** x2 ** x0 + 2 x0 ** x0 ** x2 ** x1 + 2 x0 ** x1 ** x0 ** x2
```

```
ModifiedCompositionProduct[x0**x1,{x0,x1},2]
```

```
x0 ** x1 + x0 ** x0 ** x0 ** x1
```

ScalarProduct

- **Description**

Given $c,d \in \mathbb{R}^{\ell}\langle\langle X\rangle\rangle$, the scalar product is

$$(c, d) = \sum_{\eta \in X^*} \sum_{i=1}^{\ell} (c_i, \eta)\,(d_i, \eta).$$

Of course for many series this sum diverges. The current implementation assumes $\ell = 1$.

- **Syntax**

`ScalarProduct[series1,series2]`

`series1` and **`series2`** are noncommutative formal power series.

- **Example**

```
ScalarProduct[x0**x1+3x0**x0,x1**x1**x0+2x0**x1]
```

```
ScalarProduct[3 x0 ** x0 + x0 ** x1, 2 x0 ** x1 + x1 ** x1 ** x0]
```

ShuffleProduct

- **Description**

The shuffle product of two words $\eta = x_j\eta'$, $\xi = x_k\xi' \in X^*$ is defined iteratively as

$$\eta \text{ ш } \xi = (x_j\eta') \text{ ш } (x_k\xi') = x_j(\eta' \text{ш } \xi) + x_k(\eta \text{ ш } \xi')$$

with ξ ш $\emptyset = \emptyset$ ш $\xi = \xi$. The definition of the shuffle product is extended linearly to any two series $c,d \in \mathbb{R}\langle\langle X\rangle\rangle$ by letting

$$c \text{ ш } d = \sum_{\eta,\xi \in X^*} (c, \eta)\,(d, \xi)\,\eta \text{ ш } \xi.$$

Given two series $c,d \in \mathbb{R}^\ell\langle\langle X\rangle\rangle$ the shuffle product c ш d is defined componentwise, i.e., the i-th component series of c ш d is $(c \text{ ш } d)_i = c_i \text{ ш } d_i$, where $1 \le i \le \ell$. The current implementation assumes $\ell = 1$.

- **Syntax**

  ShuffleProduct[series1,series2]

  series1 and **series2** are noncommutative formal power series.

- **Example**

```
NCExpand[ShuffleProduct[2x0**x1,x0+x1]]
```

```
4 x0 ** x0 ** x1 + 2 x0 ** x1 ** x0 + 4 x0 ** x1 ** x1 + 2 x1 ** x0 ** x1
```

- **UltraMetricDistance**

- **Description**

  An ultrametic on $\mathbb{R}^\ell\langle\langle X\rangle\rangle$ is a mapping of the form

$$\text{dist} : \mathbb{R}^\ell\langle\langle X\rangle\rangle \times \mathbb{R}^\ell\langle\langle X\rangle\rangle \to \mathbb{R} : (c, d) \mapsto \sigma^{\text{ord}(c-d)}$$

  where σ such that $0 < \sigma < 1$. The current implementation assumes $\ell = 1$.

- **Syntax**

  UltrameticDistance[series1,series2,real]

  series1 and **series2** are noncommutative formal power series, and **real** is a real number strictly between 0 and 1.

- **Example**

```
UltrametricDistance[x0+x0**x1**x1+x1**x0**x0,x0+x0**x0+x0**x0**x0,0.5]
```

```
0.25
```

4 Other Commands

- **RealizationToSeries**

- **Description**

  Let $G = \{g_0, g_1, \ldots, g_m\}$ be a set of smooth $\mathbb{R}^n$-vector fields defined on a neighborhood W of $z_0 \in \mathbb{R}^n$. g_0 is taken as the drift vector field. Let h be a smooth $\mathbb{R}^\ell$-valued function on W. The generating series of the realization (G, h, z_0) is $c \in \mathbb{R}^\ell\langle\langle X\rangle\rangle$, where $X = \{x_0, x_1, \ldots, x_m\}$ and

$$(c, \eta) = L_{g_{i_1}} \cdots L_{g_{i_k}} h(z_0),\ \eta = x_{i_k} \cdots x_{i_1} \in X^*$$

  with $L_{g_i} : h \mapsto \partial h/\partial z \cdot g_i$ and $L_\emptyset h = h$.

- **Syntax**

  RealizationToSeries[list1,function,list2,list3,integer]

  list1 is a list of m+1 mappings from $\mathbb{R}^n$ to $\mathbb{R}^n$, **function** is a mapping from $\mathbb{R}^n$ to $\mathbb{R}^\ell$, **list2** is a list of n real numbers describing an initial condition, and **list3** is the m+1 letter alphabet for the generating series c. The output series c is truncated to degree **integer**. The current implementation assumes $\ell = 1$.

■ **Example**

```
g0[z1_,z2_,z3_,z4_]:={z1 z2-z1^3,z1,-z3,z1^2+z2}
g1[z1_,z2_,z3_,z4_]:={0,2+2 z3,1,0}
h[z1_,z2_,z3_,z4_]:=z4
z0:={0,0,0,0}
X={x0,x1};
RealizationToSeries[{g0,g1},h,z0,X,5]
```

```
2 x0 ** x1 + 2 x0 ** x1 ** x1 - 2 x0 ** x1 ** x0 ** x1 + 2 x0 ** x1 ** x0 ** x0 ** x1
```

APPENDIX B

MODIFIED TENSOR CODE

```
(*Generating NC alphabet for multi-input*)
$RecursionLimit = Infinity;
FdBdegree = 7;
(*Numbers in character maps*)
CharacterMapsNums = ToString /@ Flatten[Range[0, m]];
(*Generating alphabet*)
alphabet = ToExpression[StringJoin["x", #] & /@ CharacterMapsNums];
(*ϕ is used to denote the empty word*)
SetNonCommutative /@ Flatten[Append[{ϕ}, alphabet]];


(*Concatenate coordinate function prefix with word index*)
Clear[Catm]
Catm[w_, x_List] := Flatten[StringJoin[w, #] & /@ x];
Clear[CatmList]
CatmList[w_List, x_] := Flatten[Catm[#, x] & /@ w];

(*Generate all coordinate functions with words of up to length n*)
Clear[CharacterMaps]
CharacterMaps[prefix_, y_List, n_Integer] :=
 Catm[prefix, Flatten[NestList[CatmList[#, y] &, {""}, n]]]

(*Generating \ell series for multi-output *)
acoeffs = Table[ToExpression[
    CharacterMaps["a" <> ToString[i], CharacterMapsNums, FdBdegree]], {i, 1, l}];

(*Define distribution over tensor products for performance gain*)
Clear[DistributeTensor]
DistributeTensor[(c_:1) * tensor_] := Distribute[c * Distribute[tensor]];
DistributeTensor[tensor_Plus] := DistributeTensor[#] & /@ tensor;

(*Define a custom tensor product to replace notation*)
Clear[NCTensorProduct]
SetAttributes[NCTensorProduct, {Flat, Listable, OneIdentity}]
NCTensorProduct[(a_ + b_), c_] := NCTensorProduct[a, c] + NCTensorProduct[b, c];
NCTensorProduct[a_, (b_ + c_)] := NCTensorProduct[a, b] + NCTensorProduct[a, c];
NCTensorProduct[(c_?NumericQ * a_), (d_?NumericQ * b_)] :=
  DistributeTensor[c * d * NCTensorProduct[a, b]];
NCTensorProduct[(c_?NumericQ * a_), b__] := DistributeTensor[c * NCTensorProduct[a, b]];
NCTensorProduct[a__, (d_?NumericQ * b_)] := DistributeTensor[d * NCTensorProduct[a, b]];

(*Define notation using custom tensor*)
<< Notation`

InfixNotation[ ⊗ , NCTensorProduct]

Unprotect[Times]
Times[NCTensorProduct[a1_Symbol, b1_Symbol],
   NCTensorProduct[a2_Symbol, b2_Symbol]] := (a1 * a2) ⊗ (b1 * b2);

Notation[ (x1_⊗x2_) * (y1_⊗y2_)  ⟺
```

```
  Times[NCTensorProduct[x1_, x2_], NCTensorProduct[y1_, y2_]] ]

(*Define our tensor product to use the above noncommuting tensor notation*)
Clear[ProductTensor]
SetAttributes[ProductTensor, {Flat, Listable, Orderless}]
ProductTensor[(c1_:1) * a__, ((c2_:1) * b__)] := If[NumericQ[c1], If[NumericQ[c2],
    DistributeTensor[c1 * c2 * ProductTensor[DistributeTensor[a], DistributeTensor[b]]],
    DistributeTensor[c1 * ProductTensor[DistributeTensor[a], DistributeTensor[c2 * b]]]],
   DistributeTensor[ProductTensor[DistributeTensor[c1 * a], DistributeTensor[c2 * b]]]];
ProductTensor[a__Plus, b__] := ProductTensor[#, b] & /@ a;
ProductTensor[a__, b__Plus] := ProductTensor[a, #] & /@ b;
ProductTensor[a__Plus, b__Plus] := ProductTensor[a, #] & /@ b;
ProductTensor[NCTensorProduct[a1__, b1__], NCTensorProduct[a2__, b2__]] :=
  NCTensorProduct[a1 * a2, b1 * b2];
ProductTensor[(c1_:1) * NCTensorProduct[a1__, b1__],
   ((c2_:1) * NCTensorProduct[a2__, b2__])] := c1 * c2 (a1 * a2) ⊗ (b1 * b2);

(*Modify Times to use our noncommuting tensor notation*)
Unprotect[Times]
Times[(c1_:1) * NCTensorProduct[a1__, b1__], ((c2_:1) * NCTensorProduct[a2__, b2__])] :=
  c1 * c2 (a1 * a2) ⊗ (b1 * b2);
Notation[ (x1_⊗x2_) * (y1_⊗y2_) ⟸
  ProductTensor[NCTensorProduct[x1_, x2_], NCTensorProduct[y1_, y2_]] ]

(*Define the product on the Hopf algebra*)
Clear[MuProduct]
MuProduct[tensor_] := DistributeTensor[tensor /. NCTensorProduct → Times];
MuProduct[a_List] := Apply[Times, a];
MuProduct[a_ ⊗ "1"] := a;
MuProduct["1" ⊗ a_] := a;
MuProduct[{a_, "1"}] := a;
MuProduct[{"1", a_}] := a;

(*Permute tensor products*)
Clear[TauPermutation]
TauPermutation[a_ ⊗ b_] := NCTensorProduct[b, a];
TauPermutation[a_, b_] := NCTensorProduct[b, a];
TauPermutation[a_Plus] := TauPermutation[#] & /@ a;
TauPermutation[(c_:1) * a__ ⊗ ((d_:1) * b__)] :=
  If[NumericQ[c], If[NumericQ[d], c * d * DistributeTensor[TauPermutation[a, b]],
    c * DistributeTensor[TauPermutation[a, d * b]]],
   d * DistributeTensor[TauPermutation[c * a, b]]];

(*Define the left shift operator*)
Clear[ThetaMap]
ThetaMap[(c_:1) * a_, num_] := c * ToExpression[
    StringTake[ToString[a], 2] <> ToString[num] <> StringDrop[ToString[a], 2]];
Clear[ThetaMapShift]
ThetaMapShift[(c_:1) * a_, num_] := c * If[StringTake[ToString[a], {3}] === ToString[num],
```

```
    StringReplacePart[ToString[a], "", {3, 3}], Message[ThetaMapShift::nnarg, x];
    0];
ThetaMapShift::nnarg = "No shift possible.";
ThetaMapShift[0, num_] := 0;

(*Define functions for tensors with distributing compositions*)
Clear[TensorComposition2]
TensorComposition2[f_, g_, tensor_] :=
  DistributeTensor[tensor /. NCTensorProduct[a_, b_] :> NCTensorProduct[f[a], g[b]]];
Clear[TensorComposition]
TensorComposition[f_, tensor_, side_] := If[side == 1,
   DistributeTensor[tensor /. NCTensorProduct[a_, b_] :> NCTensorProduct[f[a], b]],
   If[side == 2, tensor /. NCTensorProduct[a_, b_] :> NCTensorProduct[a, f[b]]]];

(*Performance gain for the case of words with 0 coefficients*)
Clear[IdentityReplace]
IdentityReplace[a_, rules_] := Identity[a] /. rules;

(*Define tensor distribution for order 3 tensor compositions*)
Clear[TensorCompositionMu]
TensorCompositionMu[f_, g_, tensor_] := DistributeTensor[
  tensor /. NCTensorProduct[a_, b_, c_] :> NCTensorProduct[f[a], g[NCTensorProduct[b, c]]]]

(*Default value (1) is because there is only one series*)
Clear[CoproductShuffle]
CoproductShuffle[a_, component_: 1] :=
 CoproductShuffle[a, component] = Module[{s1, s2}, s1 = ToString[a];
   If[StringLength[s1] <= 2, If[NumericQ[a] , Message[CoproductShuffle::nnarg, x];
     0, ToExpression[a] ⊗ ToExpression[StringDrop[s1, -1] <> ToString[component]]],
    s2 = ToExpression[StringTake[s1, {3, 3}]];
    TensorComposition[ThetaMap[#, s2] &, CoproductShuffle[
       ThetaMapShift[s1, s2], component], 1] + TensorComposition[
      ThetaMap[#, s2] &, CoproductShuffle[ThetaMapShift[s1, s2], component], 2]]]
CoproductShuffle::nnarg = "Incorrect argument. Argument should be, for example,
    ai1120. But if coefficient is zero, then the evaluation is zero too";

(*Setting properties of ϕ with respect to NonCommutativeMultiply*)
NonCommutativeMultiply[ϕ, a__] := a;
NonCommutativeMultiply[a__, ϕ] := a;
NonCommutativeMultiply[ϕ, ϕ] := ϕ;

(*Define multiplicative behavior for empty word*)
Unprotect[Times]
Times[ϕ, a__] := a;
Times[a__, ϕ] := a;
Times[ϕ, ϕ] := ϕ;

(*CoproductShuffle[a]-ϕ⊗a-a⊗ϕ*)
Clear[ReducedCoproductShuffle]
ReducedCoproductShuffle[a_, component_: 1] :=
  ReducedCoproductShuffle[a, component] = CoproductShuffle[a, component] -
```

```
    NCTensorProduct[ToExpression[StringTake[ToString[a], 2]], ToExpression[
      StringTake[ToString[a], 1] <> ToString[component] <> StringDrop[ToString[a], 2]]] -
    NCTensorProduct[a, ToExpression[StringTake[ToString[a], 1] <> ToString[component]]];

(*Implement tensor products composed with  μ*)
Clear[MuRightTensor]
MuRightTensor[f_, tensor_, side_] := MuProduct[TensorComposition[f, tensor, side]];
MuRightTensor[f_, 0] := 0;

(*Δ̃*)
Clear[CoproductComposition]
CoproductComposition[a_, rules_: 1] :=
  CoproductComposition[a, rules] = Module[{s1, s2, s3}, s1 = ToString[a];
    If[StringLength[s1] ≤ 2, If[NumericQ[a], Message[CoproductComposition::nnarg, x],
      NCTensorProduct[ToExpression[a], "1"]], s2 = ToExpression[StringTake[s1, {3, 3}]];
     If[ rules === 1, s3 = TensorComposition[ThetaMap[#, s2] &,
        CoproductComposition[ThetaMapShift[s1, s2], rules], 1];
      If[NumericQ[s2], If[s2 == 0,
        s3 + Sum[TensorCompositionMu[ThetaMap[#, i] &, MuProduct,
            TensorComposition[CoproductComposition[#, rules] &, CoproductShuffle[
              ThetaMapShift[s1, s2], i], 1]], {i, 1, m}], If[s2 ≠ 0, s3] ]],
      s3 = TensorComposition2[ThetaMap[#, s2] &, IdentityReplace[#, rules] &,
        CoproductComposition[ThetaMapShift[s1, s2], rules]];
      If[NumericQ[s2], If[s2 == 0,
        s3 + Sum[TensorCompositionMu[ThetaMap[#, i] &, MuProduct, TensorComposition2[
            CoproductComposition[#, rules] &, IdentityReplace[#, rules] &,
            CoproductShuffle[ThetaMapShift[s1, s2], i]]], {i, 1, m}], If[s2 ≠ 0, s3]]]]]];
CoproductComposition::nnarg = "Incorrect argument";

Clear[ReducedCoproductComposition]
ReducedCoproductComposition[a_, rules_: 1] :=
  ReducedCoproductComposition[a, rules] = CoproductComposition[a, rules] - a ⊗ "1";

(*Antipode of the Hopf algebra applied to the left (1) or right (2)*)
Clear[CompositionAntipode]
CompositionAntipode[a_, side_: 1, rules_: 1] :=
  CompositionAntipode[a, side, rules] = Outer[CompositionAntipode, a, side, rules];
CompositionAntipode[a_Times, side_: 1, rules_: 1] :=
 CompositionAntipode[a, side, rules] = Outer[CompositionAntipode[#, side, rules] &, a]
CompositionAntipode[a_^n_, side_: 1, rules_: 1] := CompositionAntipode[a^n, side, rules] =
  Expand[Product[CompositionAntipode[a, side, rules], {i, 1, n}]]
CompositionAntipode[a_, side_: 1, rules_: 1] := CompositionAntipode[a, side, rules] =
   CompositionAntipode[a, side, rules] = -a - MuRightTensor[
      CompositionAntipode[#, side, rules] &, ReducedCoproductComposition[a, rules], side] ;
```

APPENDIX C

SAMPLE TEST CODE

```
Quit[](* Ensure a fresh kernel is initialized before next test case is run *)
```

Tensor Method: Time and Final Memory

```
<< NCFPS` (* Load package and initialize kernel *)

(* Define dimensions of system *)
m = 1;
l = m;
(* Select coordinate function from list *)
wordNum = 2;
cfs = {a1, a10, a100, a1000, a10000, a100000, a1000000, a10000000}[[wordNum]];

(* Define CompositionAntipode and its
 associated functions. Not shown here for brevity. *)


{}

(* Define time constraint on execution *)
t = 60 * 60 * 1;

(* Execute test, obtaining timing and memory needed to store result *)
pre = MemoryInUse[];
TimeConstrained[CompositionAntipode[cfs, 2]; // AbsoluteTiming, t]
post = MemoryInUse[];
post - pre
```

```
Quit[] (* Ensure a fresh kernel is initialized before next test case is run *)
```

Tensor Method: Intermediate Memory

```
<< NCFPS` (* Load package and initialize kernel *)

(* Define dimensions of system *)
m = 1;
l = m;
(* Select coordinate function from list *)
wordNum = 2;
cfs = {a1, a10, a100, a1000, a10000, a100000, a1000000, a10000000}[[wordNum]];

(* Define CompositionAntipode and its
 associated functions. Not shown here for brevity. *)


{}

(* Define time constraint on execution *)
t = 60 * 60 * 1;

(* Execute test, identifying peak memory consumption during execution. *)
TimeConstrained[CompositionAntipode[cfs, 2]; // MaxMemoryUsed, t]
```

```
Quit[](* Ensure a fresh kernel is initialized before next test case is run *)
```

Derivation Method: Time and Final Memory

```
<< NCFPS` (* Load package and initialize kernel *)

(* Define alphabet and set letters as noncommuting *)
X = {x0, x1, x2, x3, x4};
SetNonCommutative /@ X;

(* Choose indexing word and define the size of the alphabet used for testing *)
wordNum = 2;
letters = 2;

(* Define index word *)
word = KleeneStar[{x0}, 7][[wordNum]]
```

```
x0
```

```
(* Define time constraint on execution *)
t = 60 * 60 * 1;
(* Execute test, obtaining timing and memory needed to store result *)
pre = MemoryInUse[];
TimeConstrained[Antipode[A[1, word], X[[1 ;; letters]]]; // AbsoluteTiming, t]
post = MemoryInUse[];
post - pre
```

```
Quit[](* Ensure a fresh kernel is initialized before next test case is run *)
```

Derivation Method: Intermediate Memory

```
<< NCFPS` (* Load package and initialize kernel *)

(* Define alphabet and set letters as noncommuting *)
X = {x0, x1, x2, x3, x4};
SetNonCommutative /@ X;

(* Choose indexing word and define the size of the alphabet used for testing *)
wordNum = 2;
letters = 2;

(* Define index word *)
word = KleeneStar[{x0}, 7][[wordNum]]
```

```
x0
```

```
(* Define time constraint on execution *)
t = 60 * 60 * 1;
(* Execute test, identifying peak memory consumption during execution. *)
TimeConstrained[Antipode[A[1, word], X[[1 ;; letters]]]; // MaxMemoryUsed, t]
```

VITA

Lance Berlin
Department of Electrical and Computer Engineering
Old Dominion University
Norfolk, VA 23529

EDUCATION

- B. S. Electrical Engineering, Computer Engineering, Old Dominion University, Norfolk, Virginia, USA, 2013

PUBLICATIONS

Conference Publications:

1. **L. Berlin**, W. S. Gray, L. A. D. Espinosa and K. Ebrahimi-Fard, "On the performance of antipode algorithms for the multivariable output feedback Hopf algebra," *51st Annual Conference on Information Sciences and Systems (CISS)*, Baltimore, MD, 2017, pp. 1-6.

Typeset using LaTeX.