Prepared for submission to JHEP

CP3-19-17, CERN-TH-2019-045
SLAC-PUB-17423

PolyLogTools – Polylogs for the masses

Claude Duhr[a,b] **Falko Dulat**[c]

[a] *Theoretical Physics Department, CERN, Geneva, Switzerland*
[b] *Center for Cosmology, Particle Physics and Phenomenology (CP3), Université Catholique de Louvain, 1348 Louvain-La-Neuve, Belgium*
[c] *SLAC National Accelerator Laboratory, Stanford University, Stanford, CA 94309, USA*

E-mail: claude.duhr@cern.ch, dulatf@slac.stanford.edu

Abstract: We review recent developments in the study of multiple polylogarithms, including the Hopf algebra of the multiple polylogarithms and the symbol map, as well as the construction of single valued multiple polylogarithms and discuss an algorithm for finding fibration bases. We document how these algorithms are implemented in the Mathematica package PolyLogTools and show how it can be used to study the coproduct structure of polylogarithmic expressions and how to compute iterated parametric integrals over polylogarithmic expressions that show up in Feynman integal computations at low loop orders.

Keywords: Feynman integrals, multiple polylogarithms, Hopf algebras.

arXiv:1904.07279v1 [hep-th] 15 Apr 2019

Contents

1 Introduction **2**

2 Installation of the package **3**

3 Multiple polylogarithms in PolyLogTools **4**

4 The shuffle and stuffle algebras of MPLs **6**
- 4.1 Shuffle algebras . . . 7
- 4.2 The shuffle algebra of MPLs . . . 8
- 4.3 The stuffle algebra of MPLs . . . 9

5 Special values of MPLs **10**

6 The Hopf algebra of MPLs **11**
- 6.1 A short review of Hopf algebras . . . 12
- 6.2 The Hopf algebra of MPLs . . . 13
- 6.3 The Lie coalgebra of indecomposables . . . 15

7 Symbols of MPLs **18**
- 7.1 Symbols in PolyLogTools . . . 18
- 7.2 The shuffle Hopf algebra of symbols . . . 21
- 7.3 The cobracket conjecture, classical and Nielson polylogarithms . . . 23

8 Working with PolyLogTools **26**
- 8.1 Manipulating expressions . . . 26
- 8.2 Series expansions of MPLs . . . 28
- 8.3 Differentiation and integration . . . 30
- 8.4 Numerical evaluation of MPLs . . . 32
- 8.5 Fibration bases . . . 34

9 Single-valued MPLs **37**
- 9.1 Single-valued MPLs in PolyLogTools . . . 38
- 9.2 The Lie coalgebra of clean single-valued functions . . . 41

10 Validation **42**

11 An example calculation **43**

12 Conclusion **47**

1 Introduction

Multiple polylogarithms (MPLs) are arguably among the most important class of functions in many areas of modern high-energy physics. Beyond the mathematical study of MPLs and their special values, the multiple zeta values (MZVs) [1–12], these objects have played an important role in many developments in high-energy physics in recent years, ranging from the study of scattering amplitudes in various string theories [13–15], supergravity theories [16, 17] and maximally supersymmetric Yang-Mills (SYM) theory [18–27] to the calculation of scattering amplitudes and cross sections for processes at the Large Hadron Collider (LHC) [28–41]. Even before the understanding of multiple polylogarithms as a general class of functions, various special cases have played important roles in the physics literature. The appearance of the dilogarithm one-loop virtual corrections has been known since the early days of QED. In ref. [42] Vermaseren and Remiddi introduced a special class of multiple polylogarithms, which they dubbed harmonic polylogarithms due to their relation to harmonic sums. These functions have appeared in virtually every multiloop calculation in the last twenty years. In recent years the various special classes of polylogarithms have been understood as arising from the general class of multiple polylogarithms.

An important problem in the practical use of these various classes of functions has been the wealth of functional identities that they satisfy. In many cases these functional identities needed to be derived manually for each case under consideration, hampering an efficient use of the more general classes of polylogarithms. However, in recent years, Goncharov's seminal paper [4] on the Hopf algebra structure of multiple polylogarithms has spurred a flurry of research both on the side of mathematics as well as in physics to better understand the algebraic structures of multiple polylogarithms. This has lead in particular to the realization that the Hopf algebra of multiple polylogarithms and its associated coproduct or coaction allow the derivation of functional identities between MPLs in a purely combinatorial fashion, see, e.g., refs. [7, 28]. Based on this, an algorithmic way to compute Feynman integrals was devised [7, 30] that makes use of the ability to derive any needed functional equation using the coproduct. Similar algorithms were further developed and automated in ref. [43–46]. In particular, the package HyperInt has recently been used to evaluate complicated Feynman parameter integrals [47–50].

In parallel the algebraic properties of multiple polylogarithms have been exploited in more formal developments such as the study of motivic amplitudes [18, 19] and the amplitudes / cluster bootstrap in planar $\mathcal{N} = 4$ SYM [21–24, 26, 51–53]. These approaches exploit a particular component of the Hopf algebra of the multiple polylogarithms, the so-called symbol map, to study the algebraic structure of scattering amplitudes. Building on the understanding of the branch-cut structure of the multiple polylogarithms, it was also possible to employ the coaction to build a special class of multiple polylogarithms, called single-valued multiple polylogarithms (SVMPLs) that are free of branch cuts [8, 54]. These SVMPLs have played an important role in the study of scattering amplitudes in the so-called multi-Regge Limit [55–58] and in the calculation and study of certain periods in ϕ^4-theory [59]. Inspired by the Hopf-algebra of the multiple polylogarithms and Brown's generalization to a conjectured cosmic Galois group [60], it was shown in refs. [61–63] that

the coaction of one-loop Feynman integrals can be cast in a remarkably simple and compact form.

All these developments have profited enormously from the mathematical advances in the understanding of the algebraic properties of the multiple polylogarithms. The goal of this paper is to document the MATHEMATICA package POLYLOGTOOLS that provides an implementation of many of these algebraic structures. The focus of this implementation is on providing a basis for future exploration and experimentation, as well the ability to compute Feynman integrals that appear in two-loop and three-loop amplitudes [30]. This package in particular is not optimized for the performance that would be required to tackle Feynman integrals at arbitrary high loop orders, but it provides well-tested implementations of all required algorithms and provides the flexibility to adapt it to virtually any application mentioned above.[1] POLYLOGTOOLS provides many routines that allow the user to work with multiple polylogarithms in MATHEMATICA; from integrating expressions in terms of polylogarithms through numerical evaluation and calculation of symbols and coproducts to the automated derivation of functional identities. The goal of this paper is to review the relevant mathematics of polylogarithms and to document the routines in POLYLOGTOOLS that implement the respective mathematical algorithms.

This paper is organized as follows: In section 2 we describe how to obtain and install the package and its prerequisites. Then, in section 3 we review the basic construction of the multiple polylogarithms and describe how they are realized in POLYLOGTOOLS. In section 4 we review the first important algebraic structure that multiple polylogarithms are equipped with, the so-called shuffle algebra, and show how to use it in the package. Next, in section 5, we discuss the degeneration of multiple polylogarithms to multiple zeta values for special values of their arguments and show how they are implemented in POLYLOGTOOLS. In section 6 we review the Hopf algebra and coproduct of multiple polylogarithms and its implementation. Then, in section 7 we discuss the relation between the coproduct and the symbol map. In section 8 we discuss how to perform basic calculations in POLYLOGTOOLS. In section 9 we review the theory of SVMPLs and discuss the routines dedicated to handling these functions. Afterwards, in section 10 we review a few applications of POLYLOGTOOLS in the literature. Finally, in section 11 we discuss the calculation of a Feynman integral from start to finish to illustrate the practical use of POLYLOGTOOLS for actual calculations.

2 Installation of the package

POLYLOGTOOLS resides in a git repository at https://gitlab.com/pltteam/plt. From this gitlab URL it is possible to download a compressed file with the latest version of the repository. Alternatively, POLYLOGTOOLS can be obtained by cloning the repository using git, e.g. by issuing the following command in any shell that has the `git` command available:

```
git clone https://gitlab.com/pltteam/plt.git
```

[1] In fact private versions of this package have been used in many recent computations, see Section 10.

This will clone the PolyLogTools repository into a subfolder `plt`. From within that folder it is then possible to obtain the latest changes and bugfixes to PolyLogTools by simply issuing the following command on the shell:

```
git pull
```

PolyLogTools depends on the Mathematica package Hpl and the GiNaC library. Both codes need to be installed separately before PolyLogTools can be loaded into the current Mathematica session. The path in which Hpl has been installed needs to be present in the Mathematica variable `$Path`, which should be the case if Hpl is installed properly. The GiNaC library needs to be installed with support for the `ginsh` command line tool and `ginsh` needs to be in a path that can be found by the system (i.e., it must be possible to start GiNaC by simply typing `ginsh` into a shell command line). We note that there seems to be a problem in Mathematica on MacOS that prevents Mathematica from running the `ginsh` command from within a Mathematica session unless Mathematica has been started from the terminal. This can usually be achieved by running the following command from the terminal:

```
/Applications/Mathematica.app/Contents/MacOS/Mathematica
```

Afterwards PolyLogTools can be loaded using:

```
In[1]:=   $PolyLogPath = SetDirectory[ < path > ];
In[2]:=   <<PolyLogTools`;
```

where `< path >` is the path to the folder containing the file `PolyLogTools.m`. This loads the Mathematica packages Hpl and Combinatorica into the kernel as well. Note that PolyLogTools tries to make sure that `ginsh` is available by running '`which ginsh`'. On some systems this can fail due to the `which` command not being available. In these cases PolyLogTools will complain about not being able to locate `ginsh`, however, if `ginsh` is accessible from the terminal, this warning can be safely ignored.

3 Multiple polylogarithms in PolyLogTools

In this section we give a short review of the main actors in the PolyLogTools package, the *multiple polylogarithms* (MPLs). The aim of this section is not to provide an extensive overview of MPLs but to introduce our notations and conventions, and how these functions and some of their most basic properties are implemented into PolyLogTools.

MPLs can be defined recursively via the iterated integral ($n \geq 0$) [2, 3]

$$G(a_1, \dots, a_n; z) = \int_0^z \frac{dt}{t - a_1} G(a_2, \dots, a_n; t) \,, \tag{3.1}$$

with $G(z) = 1$ and a_i and z are complex variables. We call the vector $\vec{a} = (a_1, \dots, a_n)$ the *weight vector*, and its length n is called the *weight*. In the special case where all the a_i's are zero, we define, using the obvious vector notation $\vec{a}_n = (\underbrace{a, \dots, a}_{n})$,

$$G(\vec{0}_n; z) = \frac{1}{n!} \log^n z \,, \tag{3.2}$$

consistent with the case $n = 0$ above. In the case where the a_i are constants, MPLs are often referred to as hyperlogarithms. MPLs define a very general class of functions that generalize the well-known logarithm and (Nielsen) polylogarithm functions, e.g., for $a \neq 0$,

$$\begin{aligned} G(\vec{a}_n; z) &= \frac{1}{n!} \log^n \left(1 - \frac{z}{a}\right) , \\ G(\vec{0}_{n-1}, a; z) &= -\mathrm{Li}_n \left(\frac{z}{a}\right) , \\ G(\vec{0}_{n-k}, \vec{a}_k; z) &= (-1)^k\, S_{n-k,k} \left(\frac{z}{a}\right) . \end{aligned} \tag{3.3}$$

The function $G(a_1, \ldots, a_n; z)$ is represented within PolyLogTools by the symbol `G[a1,...,an,z]`. Here the arguments `a1`,...,`an` and `z` can be any valid Mathematica expression (in practice, we will only deal with cases where the arguments are rational or algebraic expressions). Since $G(z) = 1$, the function `G` automatically evaluates to unity if it only has a single argument.

In the case where all the a_i are 0 or ± 1, MPLs are often referred to as *harmonic polylogarithms* (HPLs) in the physics literature [42]. HPLs are equal to MPLs, up to a sign,

$$H(a_1, \ldots, a_n; z) = (-1)^p\, G(a_1, \ldots, a_n; z) , \quad a_i \in \{0, \pm 1\} , \tag{3.4}$$

where p denotes the number of elements in $\vec{a}$ equal to $(+1)$. HPLs are represented within PolyLogTools by the symbols `H[a1,...,an,z]`.

The functions `HToG[expr]` and `GToH[expr]` allow one to switch from the `H` to the `G` notation inside the Mathematica expression `expr`. Note that since the Hpl package is loaded automatically with PolyLogTools, also the notation for HPLs from the Hpl package can be used inside PolyLogTools. The functions `HToHPL[expr]`, `GToHPL[expr]`, `HPLToH[expr]` and `HPLToG[expr]` allow the user to switch between the notations used by Hpl and PolyLogTools. In particular, the function `HPLToG[expr]` can be used to convert a HPL in compressed notation to a `G` in standard notation, for example:

```
In[1]:=    HPLToG[ HPL[{2},x] ]
Out[1]:=   -G[0,1,x]
```

There is a second way to define MPLs, using nested sums rather than iterated integrals [2]:

$$\begin{aligned} \mathrm{Li}_{m_1,\ldots,m_k}(z_1, \ldots, z_k) &= \sum_{0<n_1<n_2<\cdots<n_k} \frac{z_1^{n_1} z_2^{n_2} \cdots z_k^{n_k}}{n_1^{m_1} n_2^{m_2} \cdots n_k^{m_k}} \\ &= \sum_{n_k=1}^{\infty} \frac{z_k^{n_k}}{n_k^{m_k}} \sum_{n_{k-1}=1}^{n_k-1} \cdots \sum_{n_1=1}^{n_2-1} \frac{z_1^{n_1}}{n_1^{m_1}} , \end{aligned} \tag{3.5}$$

where this definition makes sense whenever the sums converge (e.g., for $|z_i| < 1$). The number k of indices is called the *depth* of the MPL. The function $\mathrm{Li}_{m_1,\ldots,m_k}(z_1, \ldots, z_k)$ is represented inside PolyLogTools by the symbol `Li[{m1,...,mk}, {z1,...,zk}]`.

Table 1: Definition of MPLs

In addition to the functions defined by HPL:

`G[a1,a2,...,an,z]`	The multiple polylogarithm $G(a_1,\ldots,a_n;z)$. The indices a_i can be numbers, symbolic constants or functions of other variables.
`H[a1,a2,...,an,x]`	The harmonic polylogarithm $H(a_1,\ldots,a_n;z)$. The indices a_i have to be integers from the set $\{-1,0,1\}$.
`Li[{m1,...,mn},{z1,...,zn}]`	The multiple polylogarithm $\text{Li}_{m_1,\ldots,m_n}(z_1,\ldots,z_n)$ in sum notation. The m_i are integers.
Conversion functions:	
`GToH[expr]`	Replaces every `G` in the expression `expr` with the corresponding `H`, if the indices of the multiple polylogarithm are integers from the set $\{-1,0,1\}$.
`HToHPL[expr]`	Converts every `H` in `expr` to the `HPL` notation.
`GToHPL[expr]`	Replaces every `G` in `expr` with the corresponding `HPL`, if the indices of the multiple polylogarithm are integers from the set $\{-1,0,1\}$.
`HPLToH[expr]`	Replaces every `HPL` in `expr` with the corresponding `H`. Automatically converts from the $a-$notation defined by HPL to the $m-$notation.
`HToG[expr]`	Replaces every `H` in `expr` with the corresponding `G`.
`HPLToG[expr]`	Replaces every `HPL` in `expr` with the corresponding `G`. Automatically converts from the $a-$notation defined by HPL to the $m-$notation.
`GToLi[expr]`	Converts every `G` in `expr` to the `Li` notation.
`LiToG[expr]`	Converts every `Li` and classical polylogarithm in standard MATHEMATICA notation in `expr` to the `G` notation.

For depth $k = 1$ this definition naturally reduces to the usual series representation of the polylogarithm function $\text{Li}_m(z)$. The function `Li[{m}, {z}]` is therefore equivalent to the built-in MATHEMATICA function `PolyLog[m,z]`. The G and Li functions define (essentially) the same class of functions and are related by ($a_i \neq 0$)

$$G\Big(\vec{0}_{m_1-1}, a_1, \ldots, \vec{0}_{m_k-1}, a_k; z\Big) = (-1)^k \,\text{Li}_{m_k,\ldots,m_1}\left(\frac{a_{k-1}}{a_k}, \ldots, \frac{a_1}{a_2}, \frac{z}{a_1}\right) . \tag{3.6}$$

The functions `LiToG[expr]` and `GToLi[expr]` allow the user to switch between the two representations of MPLs, either as iterated integrals (`G`) or as nested sums (`Li`).

4 The shuffle and stuffle algebras of MPLs

One of the most basic properties of MPLs is that they can be equipped with two algebra structures, one related to the representation of MPLs as iterated integrals – the shuffle algebra – and another one related to the representation as nested series – the stuffle

algebra. Most of the functionalities of PolyLogTools rely on the shuffle algebra properties of MPLs. Since we will encounter another shuffle algebra in Section 7, we start this section by reviewing the mathematics of shuffle algebras in general before discussing the implementation of the shuffle algebra of MPLs in PolyLogTools.

4.1 Shuffle algebras

Consider a (finite) set A, whose elements we will denote as *letters*, and we call A the *alphabet*. For concreteness, we will choose here $A = \{a, b, c, \ldots\}$, though A can be any finite set. We define a vector space V as the vector space formed by all $\mathbb{Q}$-linear combinations of words formed from the letters in A, including the empty word, which we denote simply as 1. The *length* of a word is defined as the number of letters that the word is made of. Words of length 1 are simply the letters. The concatenation of two words w_1 and w_2 is defined in the obvious way and denoted by w_1w_2. V has the structure of a *graded* vector space, i.e., it admits a direct sum decomposition

$$V = \bigoplus_{n=0}^{\infty} V_n = V_0 \oplus V_{>0}\,, \tag{4.1}$$

where $V_0 = \mathbb{Q}$ and V_n and $V_{>0}$ denote the subspaces of V spanned by all words of length n and all words of non-zero length respectively.

V can be given the structure of a commutative algebra equipped with the *shuffle product*. The shuffle product assigns to a pair of words (w_1, w_2) the sum of all their *shuffles*, i.e., the sum of all possible ways of permuting the letters of their union without changing the order of the letters within each word. The shuffle product can also be defined recursively: if $\alpha, \beta \in A$ are letters and $w_1, w_2 \in V$ are words, then the shuffle product of the words αw_1 and βw_2 is defined recursively by

$$\alpha w_1 \sqcup\!\sqcup\, \beta w_2 = \alpha(w_1 \sqcup\!\sqcup\, \beta w_2) + \beta(\alpha w_1 \sqcup\!\sqcup\, w_2)\,, \tag{4.2}$$

and the recursion starts with $1 \sqcup\!\sqcup\, w = w \sqcup\!\sqcup\, 1 = w$. For example, the shuffle product of the words ab and cd is

$$ab \sqcup\!\sqcup\, cd = abcd + acbd + cabd + acdb + cadb + cdab\,. \tag{4.3}$$

It is easy to check that the shuffle product is associative and commutative. Moreover, it preserves the length, i.e., the shuffle product of two words of lengths n_1 and n_2 is a linear combination of words of length $n_1 + n_2$. In this way V becomes a *graded* algebra,

$$V_{n_1} \sqcup\!\sqcup\, V_{n_2} \subseteq V_{n_1+n_2}\,. \tag{4.4}$$

It is often convenient to work with a set of generators for V, i.e., a minimal set of words such that every element of V can be written as a linear combination of products (a polynomial) in these generators. In order to define such a set of generators, we first need to define an ordering $<$ on the set of letters A (for concreteness, we can choose here the lexicographic ordering among the letters, but any other choice of ordering would do).

A theorem by Radford states that the shuffle algebra V is isomorphic to the polynomial algebra (over $\mathbb{Q}$) formed by a subset of words, called *Lyndon words* [64]. A Lyndon word is a non-empty word that is less (for the chosen ordering $<$) than any of its proper right factors, i.e., a word w is a Lyndon word if for all non-empty words w_1 and w_2 with $w = w_1 w_2$ we have $w < w_2$. A consequence of Radford's theorem is that for a given ordering every word can be uniquely written as a polynomial in Lyndon words. For example, the word baa is not a Lyndon word for the lexicographic ordering (because $baa > aa$), so it can be written as a polynomial in the Lyndon words,

$$baa = aab - a \sqcup\!\sqcup\, ab + \frac{1}{2}\, a \sqcup\!\sqcup\, a \sqcup\!\sqcup\, b\,. \tag{4.5}$$

4.2 The shuffle algebra of MPLs

Let us now consider the $\mathbb{Q}$-vector space of all MPLs of the form $G(\vec{a}; z)$ ending in the same variable z. It can be shown that this vector space forms a graded shuffle algebra, where $G(\vec{a}; z)$ is identified with the word $\vec{a} = (a_1, \ldots, a_n)$. The weight of an MPL corresponds to the length of the word. Explicitly, the shuffle product of two MPLs ending in the same variable can be written as

$$G(\vec{a}; z)\, G(\vec{b}; z) = \sum_{\vec{c} = \vec{a} \sqcup\!\sqcup \vec{b}} G(\vec{c}; z)\,, \tag{4.6}$$

where the sum runs over all shuffles of the two weight vectors $\vec{a}$ and $\vec{b}$.

It is possible to use PolyLogTools to linearise all shuffle products of MPLs ending in the same argument z by using the function `ShuffleG` as shown in the following example (cf. eq. (4.3)),

```
In[1]:=    ShuffleG[ G[a,b,z] * G[c,d,z] ]

Out[1]:=   G[a,b,c,d,z] + G[a,c,b,d,z] + G[a,c,d,b,z] + G[c,a,b,d,z] +
           G[c,a,d,b,z] + G[c,d,a,b,z]
```

Since every word can be represented as a polynomial in Lyndon words for a given ordering, we can also decompose every MPL into a linear combination of products of MPLs whose weight vectors are Lyndon words. This is achieved as follows (cf. eq. (4.5)),

```
In[2]:=    DecomposeToLyndonWords[ G[b,a,a,z], Alphabet -> {a,b} ]

Out[2]:=   G[a,a,b,z] - G[a,z] * G[a,b,z] + 1/2 * G[a,z]^2 * G[b,z]
```

The function `DecomposeToLyndonWords` does not only work on single MPLs, but its argument can be any Mathematica expression. PolyLogTools has tables of Lyndon words hard-coded up to words of length six, and so only MPLs up to weight six are reduced to Lyndon words. The tables of Lyndon words implemented in PolyLogTools have been generated with Sage [65]. The optional argument `Alphabet` takes as value a list of symbols and defines the alphabet as well as the ordering among the letters (simply the

order in which the letters appear inside the list). Its default value is `{0,1,-1}`. Note that only those MPLs whose weight vectors contain only letters from the `Alphabet` are decomposed into Lyndon words. Finally, let us make a comment about singularities. The MPL $G(a_1, \ldots, a_n; z)$ diverges whenever $z = a_1$, because in that case the integral in eq. (3.1) has an end-point singularity. It may happen that the decomposition into Lyndon words leads to divergent MPLs, even though the input was finite (e.g., take `z=a` in the above example box). In such a case the decomposition is not performed and a warning is shown.

It is often convenient to decompose MPLs not into a Lyndon word basis, but into a set of functions corresponding to words where a given letter does not appear in the last position (except for words that are only made of that letter). For example, in applications it is useful to represent a function in terms of MPLs where the last letter is non-zero, except for powers of $G(0; z)$. Such a representation can always be achieved algorithmically by unshuffling powers of logarithms. For example, we have

$$G(1,0,0;z) = G(0,0,1;z) - G(0;z)\,G(0,1;z) + G(0,0;z)G(1;z)\,. \tag{4.7}$$

This operation is implemented via the function `ExtractZeroes`, which can be used as shown,

```
In[3]:=   ExtractZeroes[ G[1,0,0,z] ]
--------------------------------------------------------------
Out[3]:=  G[0,0,1,z] - G[0,z]*G[0,1,z] + G[0,0,z]*G[1,z]
```

4.3 The stuffle algebra of MPLs

There is another algebra structure on MPLs coming from their series representation, called *stuffle algebra* [66]. Since in many applications to Feynman integrals the shuffle algebra structure seems more relevant, we will be brief and only present an example. For a more general discussion, we refer to ref. [66]. Consider a product of two polylogarithms of depth one. We can rearrange the sums in the following way:

$$\begin{aligned}
\mathrm{Li}_{m_1}(z_1)\,\mathrm{Li}_{m_2}(z_2) &= \left(\sum_{n_1=1}^{\infty}\frac{z_1^{n_1}}{n_1^{m_1}}\right)\left(\sum_{n_2=1}^{\infty}\frac{z_2^{n_2}}{n_2^{m_2}}\right) = \sum_{n_1,n_2=1}^{\infty}\frac{z_1^{n_1}z_2^{n_2}}{n_1^{m_1}n_2^{m_2}}\\
&= \sum_{\substack{n_1=1\\ n_2<n_1}}^{\infty}\frac{z_1^{n_1}z_2^{n_2}}{n_1^{m_1}n_2^{m_2}} + \sum_{\substack{n_2=1\\ n_1<n_2}}^{\infty}\frac{z_1^{n_1}z_2^{n_2}}{n_1^{m_1}n_2^{m_2}} + \sum_{n=1}^{\infty}\frac{(z_1z_2)^n}{n^{m_1+m_2}}\\
&= \mathrm{Li}_{m_1,m_2}(z_1,z_2) + \mathrm{Li}_{m_2,m_1}(z_2,z_1) + \mathrm{Li}_{m_1+m_2}(z_1\,z_2)\,.
\end{aligned} \tag{4.8}$$

Similar formulas can be derived for stuffle products of MPLs of higher depths. PolyLogTools can also expand products of Li-functions into stuffle products. For example, the stuffle product in eq. (4.8) is obtained as follows,

```
In[1]:=   StuffleLi[ Li[{m1},{z1}] * Li[{m2},{z2}] ]
--------------------------------------------------------------
Out[1]:=  Li[{m1,m2},{z1,z2}]+Li[{m2,m1},{z2,z1}]+Li[{m1+m2},{z1*z2}]
```

Table 2: Shuffle & stuffle products	
`ShuffleG[expr]`	Expands all products of G-functions that end in the same argument into a sum of shuffles.
`StuffleLi[expr]`	Expands all products of Li-functions into a sum of stuffles.
`DecomposeToLyndonWords[expr]`	Decomposes all G-functions into a sum of products of Lyndon words. The set of letters and their ordering can be passed as a list via the optional argument `Alphabet`. The default value is `{0,1,-1}`. MPLs which depend on a letter that is not in the `Alphabet` are not decomposed. The same applies to MPLs that would be decomposed into divergent quantities.
`ExtractZeroes[expr]`	Uses the shuffle algebra to remove all trailing zeroes from all `G`-functions in `expr`.

5 Special values of MPLs

When evaluated at certain special values of the arguments, MPLs often reduce to known transcendental constants. In particular, if HPLs are evaluated at $z = \pm 1$, then they reduce to (coloured) multiple zeta values (MZVs),

$$\zeta_{m_k,\ldots,m_1} = \sum_{0<n_1<n_2<\cdots<n_k} \frac{s_1^{n_1} s_2^{n_2} \cdots s_k^{n_k}}{n_1^{|m_1|} n_2^{|m_2|} \cdots n_k^{|m_k|}} \,, \quad s_i = \mathrm{sign}(m_i)\,. \tag{5.1}$$

The weight and the depth of (coloured) multiple zeta values are defined in the same way as for MPLs. The series in eq. (5.1) diverges whenever $m_k = 1$. For depth $k = 1$, MZVs reduce to Riemann's zeta function at positive integer arguments,

$$\zeta_m = \zeta(m) = \sum_{n=1}^{\infty} \frac{1}{n^m} \,. \tag{5.2}$$

MZVs and their coloured generalisations (which correspond to some of the m_i being negative) are implemented in the HPL package, where $\zeta_{m_k,\ldots,m_1}$ is represented by the symbol `MZV[{mk,...,m1}]`. In addition, the HPL package also knows how to reduce HPLs evaluated at $z = \pm i$ to a smaller set of transcendental constants. Since PolyLogTools uses HPL the reduction rules of HPLs evaluated at $z \in \{\pm 1, \pm i\}$ are readily available in PolyLogTools also for the `G` and `H` functions. We refer to the documentation of the HPL package for further details [67, 68]. In addition to the values of HPLs evaluated at $z = \pm i$, PolyLogTools can also reduce $G(\vec{a}; z)$ with $a_i \in \{0, \pm i\}$ and $z \in \{\pm 1, \pm i\}$, as well as HPLs for some small weight evaluated at $z \in \{\pm 1/2, \pm 1/3\}$, to known transcendental constants.

By default, PolyLogTools will automatically express any MPL in terms of transcendental constants whenever possible. It is possible to disable the automatic reduction to transcendental constants by setting

```
In[1]:=   PLT$AutoConvertToKnownConstants = False;
```

The default value of `PLT$AutoConvertToKnownConstants` is `True`.

We conclude this section with a comment on the regularisation of MPLs. We have seen that $G(a_1, \ldots, a_n; z)$ diverges whenever $z = a_1$. Hence, when evaluating HPLs at $z = \pm 1$ the result may be divergent. It is easy to see that all divergencies are logarithmic, and it is useful to introduce a formal quantity $\lim_{z\to 1} G(1; z) = -\zeta_1$ which acts as a regulator and allows one to keep track of divergences. Indeed, all divergencies show up as powers of ζ_1, and the dependence on the regulator ζ_1 must cancel for finite quantities. The regulator ζ_1 is represented by `HPL[{1},1]` in Hpl and PolyLogTools.

There is, however, a subtle point about the regularisation discussed in the previous paragraph. As usual, there is an ambiguity in how to introduce the regulator, related to shifting divergent quantities by finite terms. In practice, one would like to choose a regulator that preserves as many of the algebraic structures as possible. MPLs are equipped with both a shuffle and a stuffle algebra structure. However, it is not possible to preserve both structures at the same time, leading to two different 'schemes' to regulate MPLs and MZVs [69], referred to as *shuffle regularisation* and *stuffle regularisation* respectively. For example, using the shuffle product on MPLs, we obtain

$$\zeta_1^2 = \lim_{z\to 1} G(1; z)^2 = \lim_{z\to 1} 2\, G(1, 1; z)\,, \tag{5.3}$$

so that the shuffle-regulated value of $G(1, 1; z)$ at $z = 1$ is given by $\frac{1}{2}\zeta_1^2$. Instead, using the fact that $G(1; z) = -\mathrm{Li}_1(z)$ as well as the stuffle product, we find

$$\begin{aligned} \zeta_1^2 &= \lim_{z\to 1} G(1; z)^2 = \lim_{z\to 1} \mathrm{Li}_1(z)^2 = \lim_{z\to 1} 2\mathrm{Li}_{1,1}(z, z) + \mathrm{Li}_2(z^2) \\ &= \zeta_2 + \lim_{z\to 1} 2\, G(1, 1; z)\,, \end{aligned} \tag{5.4}$$

so that the stuffle-regulated value of $G(1, 1; z)$ at $z = 1$ is given by $\frac{1}{2}\zeta_1^2 - \frac{1}{2}\zeta_2$. We see that the shuffle- and stuffle-regulated values are different. Note that any finite quantity must be independent of the regularisation scheme chosen to compute it. However, care is needed that the same scheme is applied consistently throughout a computation!

By default, PolyLogTools uses shuffle regularisation to regulate the `G` and `H` functions, because in applications it is often more convenient to preserve the shuffle algebra structure (which is more closely related to the definition of MPLs as iterated integrals). It is possible to switch to stuffle regularisation by changing the value of the variable `PLT$ShuffleRegularisation` from `True` to `False`. We point out that, unlike PolyLogTools, the Hpl package uses stuffle regularisation to define the values of MZVs and HPLs at $z = \pm 1$. Care is thus needed if the output and functions from this package are used in conjunction with PolyLogTools.

6 The Hopf algebra of MPLs

An important property of MPLs is that they can be equipped with a coproduct turning them into a Hopf algebra. The Hopf algebra of MPLs has led to the development of

novel techniques to deal with Feynman integrals that evaluate to MPLs, and it has seen a multitude of applications over the last few years. In particular, it allows one to control more rigorously identities among MPLs, see ref. [28, 70] for a pedagogical introduction of how to derive identities among MPLs using the coproduct. Another important application of the coproduct is the amplitude bootstrap in planar $\mathcal{N} = 4$ Super Yang-Mills [21–24, 26, 51]. Here we will not discuss these applications in detail, but we focus on the implementation of the coproduct on MPLs in PolyLogTools. Since we will discuss different Hopf algebras in subsequent sections, we give a short review of Hopf algebras in general before focusing on the case of MPLs.

6.1 A short review of Hopf algebras

A *coalgebra* is a $\mathbb{Q}$-vector space H together with a linear map $\Delta : H \to H \otimes H$ called the coproduct. The coproduct is required to be *co-associative*, that means

$$(\mathrm{id} \otimes \Delta)\Delta = (\Delta \otimes \mathrm{id})\Delta \,. \tag{6.1}$$

A coalgebra must admit another map, called the *counit*, which does not play any role in the context of the Hopf algebras discussed in this paper. A Hopf algebra is a vector space that is both an algebra and coalgebra, such that the product and the coproduct are compatible,

$$\Delta(a \cdot b) = \Delta(a) \cdot \Delta(b) \,. \tag{6.2}$$

In the previous equation the multiplication of tensors on the right-hand side is defined component-wise,

$$(a_1 \otimes a_2) \cdot (b_1 \otimes b_2) = (a_1 \cdot b_1) \otimes (a_2 \cdot b_2) \,. \tag{6.3}$$

In addition, there is a map $S : H \to H$, called the *antipode*, which will be defined below.

The coalgebras encountered in this paper are *graded* and *connected*, i.e., they admit a direct sum decomposition

$$H = \bigoplus_{n=0}^{\infty} H_n = H_0 \oplus H_{>0} \,, \qquad H_0 = \mathbb{Q} \,, \tag{6.4}$$

and the coproduct respects the grading,

$$\Delta(H_n) \subseteq \bigoplus_{p+q=n} H_p \otimes H_q \,. \tag{6.5}$$

We can iterate the coproduct to obtain tensors with more and more factors. This iteration can be done in different ways, e.g., by iterating on the first or on the second factor of the coproduct. The co-associativity of the coproduct ensures that the different ways of iterating the decomposition into simpler objects give the same result. Since the coproduct respects the grading, it makes sense to define $\Delta_{i_1,\ldots,i_k}$ as the the part of the iterated coproduct that takes values in $H_{i_1} \otimes \cdots \otimes H_{i_k}$. The maps $\Delta_{i_1,\ldots,i_k} : H \to H_{i_1} \otimes \cdots \otimes H_{i_k}$ satisfy the obvious recursion

$$\Delta_{i_1,\ldots,i_k} = (\Delta_{i_1,\ldots,i_{k-1}} \otimes \mathrm{id})\Delta_{i,i_k} \,, \qquad i = i_1 + \ldots + i_{k-1} \,. \tag{6.6}$$

Let us also discuss the antipode S. In the case of graded and connected Hopf algebras the antipode is uniquely determined by the coproduct to be

$$S(x) = -x - m(\mathrm{id}\otimes S)\Delta'(x) = -x - m(S\otimes \mathrm{id})\Delta'(x)\,, \qquad x\in H_{>0}\,, \tag{6.7}$$

where $\Delta'(x) = \Delta(x) - 1\otimes x - x\otimes 1$ denotes the *reduced coproduct* and $m(a\otimes b) = a\cdot b$ is the multiplication in H. The antipode is linear, acts trivially on elements of weight 0, $S(1) = 1$, and preserves the multiplication and the coproduct

$$S(a\cdot b) = S(b)\cdot S(a) \quad \text{and} \quad \Delta S = (S\otimes S)\Delta. \tag{6.8}$$

Let us conclude this section by illustrating these definitions on a concrete example of a Hopf algebra. This example will come back in Section 7 in the context of MPLs. We start from the shuffle algebra V of words from Section 4.1. Every shuffle algebra can be turned into a graded and connected Hopf algebra whose coproduct is given by the deconcatenation of words,

$$\Delta_{\mathrm{dec}}(w) = \sum_{w=w_1w_2} w_1\otimes w_2\,, \tag{6.9}$$

where the sum runs over all deconcatenations of the word w, and including the trivial ones where either w_1 or w_2 is the empty word. One can check that this definition satisfies all the properties of a coproduct. The antipode is given by the reversal of words,

$$S_{\mathrm{dec}}(w) = (-1)^{|w|}\,\tilde{w}\,, \tag{6.10}$$

where $|w|$ denotes the length of w, and the tilde denotes the reversal of words (e.g., if $w = abc$, then $\tilde{w} = cba$).

6.2 The Hopf algebra of MPLs

In ref. [4] it was argued that MPLs form a graded and connected Hopf algebra, where the grading comes again from the weight of the functions. The explicit formula for the coproduct is rather involved, and most conveniently expressed not in terms of the MPLs defined in eq. (3.1), but in terms of the iterated integrals

$$I(a_0; a_1,\ldots,a_n; a_{n+1}) = \int_{a_0}^{a_{n+1}} \frac{dt}{t-a_n}\, I(a_0; a_1,\ldots,a_{n-1}; t)\,. \tag{6.11}$$

It is clear that the functions defined in eqs. (3.1) and (6.11) define the same space of functions. In terms of the functions defined in eq. (6.11), the coproduct on MPLs can be written in the following compact form [4],

$$\begin{aligned}\Delta(I(a_0; a_1,\ldots,a_n; a_{n+1})) = &\sum_{0=i_1<i_2<\ldots<i_k<i_{k+1}=n} I(a_0; a_{i_1},\ldots,a_{i_k}; a_{n+1}) \\ &\otimes\left[\prod_{p=0}^{k} I(a_{i_p}; a_{i_p+1},\ldots,a_{i_{p+1}-1}; a_{i_{p+1}})\right].\end{aligned} \tag{6.12}$$

Here the sum runs over all ordered subsets of a given length k of $(a_1, \ldots, a_n)$. Since the Hopf algebra of MPLs is graded and connected, the antipode is uniquely determined by the coproduct and eq. (6.7), so we do not show it here explicitly.

Let us make some comments about the formula for the coproduct in eq. (6.12). First, the individual terms in eq. (6.12) admit a simple combinatorial interpretation in terms of polygons inscribed into a semi-circle [3, 4]. It is based on this combinatorial framework that the coproduct on MPLs is implemented in PolyLogTools, and not via the explicit formula in eq. (6.12). We do not review the combinatorial picture based on semi-circles here, but we refer to ref. [70] where worked-out examples can be found. Second, we note that the formula for the coproduct in eq. (6.12) is only valid in the generic case where all the arguments a_i are distinct. In the non-generic case individual terms in eq. (6.12) may diverge, and we need to replace the MPLs on the right-hand side of eq. (6.12) with suitably regularised versions. We follow closely refs. [4, 28, 70], identifying all MPLs on the right-hand side of eq. (6.12) with their shuffle-regularised versions (cf. Section 5), and setting to zero the regulator ζ_1.

The implementation of the coproduct on MPLs is one of the main features of the PolyLogTools package, because the coproduct is the basis for many applications. The coproduct is called via the function `Delta`, which takes as argument any valid expression in terms of MPLs of weight up to twelve and returns its coproduct. For example, we have

```
In[1]:=    Delta[ G[a,b,z] ]
----------------------------------------------------------------------
Out[1]:=   CT[1, G[a,b,z]] + CT[G[a,b,z], 1] + CT[G[a,z], G[b,a]] -
           CT[G[b,z], G[a,b]] + CT[G[b,z], G[a,z]]
```

Tensors are represented in PolyLogTools by lists with head `CT`, i.e., the tensor $A_1 \otimes A_2 \otimes A_3 \otimes \ldots$ is represented by the symbol `CT[A1,A2,A3,...]`.[2] Note that `Delta` can be applied to any valid expression made of polylogarithms, independently of how they are represented (`G`, `H`, `Li`, `Log`, `PolyLog`). The only restriction is that no MPLs of weight higher than twelve are allowed in the current implementation. The coproduct also acts non trivially on MZVs (because they are just special values of MPLs), e.g.,

```
In[2]:=    Delta[ MZV[{5,3}] ]
----------------------------------------------------------------------
Out[2]:=   CT[1,MZV[{5,3}]] + CT[MZV[{5,3}],1] - 5*CT[Zeta[3],Zeta[5]]
```

The symbol `CT` satisfies the basic properties of a tensor product. In particular, it satisfies $A \otimes (B \otimes C) \otimes D = A \otimes B \otimes C \otimes D$, i.e., we have

```
In[3]:=    CT[A,CT[B,C],D]
----------------------------------------------------------------------
Out[3]:=   CT[A,B,C,D]
```

[2] `CT` is the abbreviation for `CircleTimes`, which is the name given by Mathematica to the $\otimes$ symbol. Note that in `StandardForm` or `TraditionalForm` the head `CT` is automatically formatted to look like a tensor product.

Moreover, it is linear with respect to rational numbers and the imaginary unit i, e.g.,

```
 In[4]:=   CT[A,2*I*G[a,z] + 1/2*G[b,z],B]

Out[4]:=   2*I*CT[A,G[a,z],B] + 1/2*CT[A,G[b,z],B]
```

It also automatically evaluates products of tensors (cf. eq. (6.3)),

```
 In[5]:=   CT[a1,b1] * CT[a2,b2]

Out[5]:=   CT[a1*a2,b1*b2]
```

The different components $\Delta_{i_1,\ldots,i_k}$ of the iterated coproduct can be accessed as in the following example for $\Delta_{2,1,1}$,

```
 In[6]:=   Delta[{2,1,1},G[0,0,1,1,z]]

Out[6]:=   CT[G[1,1,z],G[0,z],G[0,z]]
```

The different components of the iterated coproduct are evaluated internally using the recursion in eq. (6.6).

At this point we have to make an important comment: MPLs are multivalued functions. It is only MPLs modulo their discontinuities that form a Hopf algebra. The discontinuities are related to taking residues at the simple poles in the integrand in eq. (3.1). Therefore the discontinuities of MPLs are always proportional to $i\pi$. Hence, in order to obtain a Hopf algebra, we need to work modulo $i\pi$. In applications, however, it is important to keep track of powers of π. It is possible to incorporate $i\pi$ into the construction by introducing the special rule [7] (see also ref. [28]),[3]

$$\Delta(i\pi) = i\pi \otimes 1\,. \tag{6.13}$$

More precisely, we should work modulo $i\pi$ in all entries of the coproduct apart from the first. For example, in PolyLogTools one obtains

```
 In[7]:=   CT[I*Pi*A,B] + CT[C,I*Pi*D]

Out[7]:=   I*CT[Pi*A,B]
```

6.3 The Lie coalgebra of indecomposables

In applications it can be useful to focus on the most complicated part of an expression. One way to do so is to look at an expression defined modulo products. In particular, one may want to decide if two expressions are equal up to 'simpler' product terms. Since MPLs

[3] This special rule can be motivated because one obtains a co-module equipped with a coaction. By abuse of language, we will only refer here to the Hopf algebra and the coproduct, as in physics applications the distinction is often minor. Internally PolyLogTools always computes the coaction.

form a shuffle algebra, it can be hard to decide if an expression is zero up to product terms. For example, one may be tempted to believe that the expression $T = G(a,b;z) + G(b,a;z)$ does not involve any product terms. In this case it is easy to see that the sum in T is precisely a shuffle product, so that T actually vanishes modulo products. In this section we show how we can construct a map whose kernel is precisely generated by all products among MPLs.

If H denotes a graded and connected Hopf algebra, then we define its *space of indecomposables* $Q(H)$ as the Hopf algebra H modulo all non-trivial products (i.e., products among objects of weight at least one),

$$Q(H) = H/(H_{>0} \cdot H_{>0})\,. \tag{6.14}$$

$Q(H)$ is obviously a vector space (because it is the quotient of two vector spaces). It is, however, not an algebra (and thus not a Hopf algebra), because it was defined precisely by removing all products.

We now construct a projector $P : H \to Q(H)$ which allows one to remove all product terms. We start by recursively defining a linear map R which acts on $x \in H_n$, $n > 0$, by [71]

$$R(x) = n\,x - m(\mathrm{id} \otimes R)\Delta'(x)\,, \tag{6.15}$$

where m is the multiplication in H and Δ' is the reduced coproduct, defined below eq. (6.7). One can show that the kernel of R is precisely generated by all non-trivial products in H [71],

$$\mathrm{Ker}\ R = H_{>0} \cdot H_{>0}\,. \tag{6.16}$$

The projector $P = P^2$ is then obtained by correctly normalising R, $P(x) = \frac{1}{n}R(x)$ for $x \in H_n$. For example, we have

$$P(G(0,1;z)) \ = G(0,1;z) - \frac{1}{2}\,G(0;z)\,G(1;z)\,. \tag{6.17}$$

We see from the previous example that the image of an MPL may contain product terms, despite the fact that the goal was to remove product terms. There is no contradiction: the elements of $Q(H)$ are equivalence classes of MPLs defined modulo product terms. The projector P assigns to a function a canonical representative of its equivalence class modulo products. The product terms ensure that relations among equivalence classes are satisfied. For example, we have

$$\begin{aligned} P(G(1,0;z)) \ &= G(0,1;z) - \frac{1}{2}\,G(0;z)\,G(1;z) \\ &= \frac{1}{2}\,G(1,0;z) - \frac{1}{2}\,G(0,1;z) \\ &= -P(G(0,1;z))\,, \end{aligned} \tag{6.18}$$

In agreement with the fact that $G(1,0;z)+G(0,1;z)$ is a shuffle product, and thus vanishes modulo products.

The projector P is implemented via the function `ProductProjector`. For example, we can obtain eq. (6.18) from PolyLogTools,

```
In[1]:=   ProductProjector[ G[0,1,z] ]

Out[1]:=  G[0,1,z] - 1/2*G[0,z]*G[1,z]
```

The coproduct on H induces a new structure on the space of indecomposables $Q(H)$, called a *Lie coalgebra*, equipped with a *cobracket* $\delta : Q(H) \to Q(H) \wedge Q(H)$ defined by

$$\delta(x) = (P \otimes P)(1 - \tau)\Delta(x)\,, \tag{6.19}$$

where $\tau(a \otimes b) = b \otimes a$ is the map that reverses tensors. The cobracket has recently appeared in physics in the context of scattering amplitudes in planar $\mathcal{N} = 4$ SYM [19, 72]. We take the opportunity to make some technical comments: First, the cobracket in eq. (6.19) is really only defined on MPLs modulo their discontinuities, i.e., we have to put to zero of powers of π in both entries of the wedge product. Second, in applications the cobracket in eq. (6.19) is directly defined on the MPLs themselves, i.e., on elements of H, while it is in principle only defined on the equivalence classes living in the space of indecomposables $Q(H)$. This is possible because the map δ in eq. (6.19) has the property $\delta P = \delta$, i.e., the image of an element of H under δ agrees with the image of its projection on $Q(H)$. For example, the cobracket of the dilogarithm is

$$\delta(G(0, 1; z)) = -G(0; z) \wedge G(1; z)\,. \tag{6.20}$$

The cobracket on MPLs is implemented via the function `Cobracket`, which can be used as shown in the following example (cf. eq. (6.20)),

```
In[2]:=   Cobracket[ G[0,1,z] ]

Out[2]:=  - CTW[ G[0,z],G[1,z] ]
```

The wedge product is implemented via the symbol `CTW`.[4] This symbol inherits its properties from the symbol `CT` defining the tensor product. In particular, it is linear with respect to rational numbers and i, and puts to zero all occuurences of π,

```
In[3]:=   CTW[A,2*I*G[a,z] + 1/2*G[b,z]]
In[4]:=   CTW[A,I*Pi] + CTW[I*Pi,B]

Out[3]:=  2*I*CTW[A,G[a,z]] + 1/2*CTW[A,G[b,z]]
Out[4]:=  0
```

In addition, `CTW` is antisymmetric, and always reorders its arguments into a canonical order, with the correct sign,

```
In[5]:=   CTW[B,A]

Out[5]:=  - CTW[A,B]
```

[4]The name of the symbol is composed of `CT` for `CircleTimes` and `W` for `Wedge`. In `StandardForm` and `TraditionalForm` this head is automatically formatted as wedge produt.

Table 3: The Hopf algebra of MPLs

`Delta[expr]`	Computes the coproduct (rather, the coaction) of `expr`.
`Delta[{i1,...,in},expr]`	Computes the $(i_1,\ldots,i_n)$ component of the coproduct of `expr`.
`CT[a1,...,an]`	Represents the tensor $a_1\otimes\cdots\otimes a_n$.
`Antipode[expr]`	Computes the antipode of the symbol `expr`.
`ProductProjector[expr]`	Applies the projector P to `expr`.
`Cobracket[expr]`	Computes the cobracket of `expr`.
`Cobracket[{p,q}, expr]`	Computes the component (p,q) of the cobracket of `expr`.
`CTW[a,b]`	Represents the wedge product $a\wedge b$.

7 Symbols of MPLs

7.1 Symbols in PolyLogTools

While the coproduct on MPLs is very useful in applications, it often contains too much information. It is often useful to focus on a piece of the coproduct which is easier to work with, albeit at the price of losing some information. Such a quantity is the *symbol* [4, 6, 18, 28, 73, 74], which can be defined as the maximal iteration of the coproduct (modulo $i\pi$),

$$\mathcal{S}(x)=\Delta_{1,\ldots,1}(x)\mod i\pi\,. \tag{7.1}$$

The symbol contains the same information as the maximal iteration of the coproduct. It is very easy to work with, because its entries are MPLs of weight one, i.e., ordinary logarithms. For this reason it is customary to drop the log-signs when talking about the symbol, e.g.,

$$\begin{aligned}\Delta_{1,1}(\mathrm{Li}_2(z))&=-\log(1-z)\otimes\log z\,,\\ \mathcal{S}(\mathrm{Li}_2(z))&=-(1-z)\otimes z\,.\end{aligned}\tag{7.2}$$

Besides this notational difference, there is no difference between the symbol $\mathcal{S}$ and the maximal iteration of the coproduct $\Delta_{1,\ldots,1}$.

PolyLogTools contains a function that allows the user to turn a maximal iteration of a coproduct into a symbol (which effectively only amounts to removing the log-signs), e.g.,

```
In[1]:=   X = Delta[{1,1}, PolyLog[2,z]]
In[2]:=   ToSymbol[X]

Out[1]:=  - CT[Log[1-z],Log[z]]
Out[2]:=  - CiTi[1-z, z]
```

There is also a function which readily combines the maximal iteration of the coproduct with the function `ToSymbol`:

```
 In[3]:=    SymbolMap[PolyLog[2,z]]
---
Out[3]:=    - CiTi[1-z, z]
```

Symbol tensors are represented inside the code by lists with head `CiTi`.[5] The reason for introducing another definition for tensor products comes from the fact that the two tensors `CT` and `CiTi` have different linearity properties: while `CT[1-z,z]` automatically evaluates to the sum `CT[1,z] - CT[z,z]`, this should obviously not be the case for the symbol tensor `CiTi[1-z,z]`. Instead, symbol tensors inherit their linearity properties from the logarithm function,

$$A \otimes (\pm 1) \otimes B = 0\,, \tag{7.3}$$

$$A \otimes (x \cdot y) \otimes B = A \otimes x \otimes B + A \otimes y \otimes B\,. \tag{7.4}$$

Hence, whenever an entry in a list with head `CiTi` is ± 1, it automatically evaluates to zero,

```
 In[4]:=    CiTi[A,1,B] + CiTi[C,-1,B]
---
Out[4]:=    0
```

The additivity of the symbol in eq. (7.4) is not applied automatically. Instead, the user can instruct PolyLogTools to apply it whenever possible via the function `SymbolExpand`

```
 In[5]:=    S = CiTi[A,x*y,B]
 In[6]:=    SymbolExpand[ S ]
---
Out[5]:=    CiTi[A,x*y,B]
Out[6]:=    CiTi[A,x,B] + CiTi[A,y,B]
```

In applications, the entries in a symbol tensor – the so-called *letters* – are often rational functions of kinematic variables. Using the additivity in eq. (7.4), one can always write such a symbol in a form where all the letters are polynomials. The function `SymbolExpand` automatically maximally factors all polynomial symbol entries over the integers, so that after the application of this function all letters are irreducible polynomials over $\mathbb{Z}$. For example, we have

```
 In[7]:=    S = CiTi[1-x^3,x^2]
 In[8]:=    SymbolExpand[ S ]
---
Out[7]:=    CiTi[1-x^3,x^2]
Out[8]:=    2*CiTi[1-x,x] + 2*CiTi[1+x+x^2,x]
```

[5] Which is another variant of an abbreviation for `CircleTimes`. In `StandardForm` and `TraditionalForm` this head is also automatically formated as tensor product.

Sometimes it can be useful to undo the expansion of the symbol, in order to combine expanded terms back into products or ratios. The user can instruct PolyLogTools to attempt to combine such terms via the function `SymbolFactor`

```
In[9]:=   SymbolFactor[ 2*CiTi[A,x,B]-CiTi[A,y,B] ]
---
Out[9]:=  CiTi[A,x^2/y,B]
```

This works best when the coefficients of the symbol tensors are integers, since factors of $1/2$ can exponentiate to unintended square roots in this procedure. This function is particularly useful when the symbol alphabet contains factored roots of a quadratic equation that one wants to combine again. For example we can use the function `SymbolFactor` to perform simplifications of the form:

```
In[10]:=   SymbolFactor[CiTi[A,1-Sqrt[x]]+CiTi[A,1+Sqrt[x]]
---
Out[10]:=  CiTi[A,1-x]
```

It is possible to obtain the list of all letters in a symbol – the *symbol alphabet* – by applying the function `GetSymbolAlphabet`. For example, we have

```
In[11]:=   S = CiTi[1-x^3,x^2]
In[12]:=   GetSymbolAlphabet[ S ]
---
Out[11]:=  CiTi[1-x^3,x^2]
Out[12]:=  {x^2,1-x^3}
```

We note here that the alphabet returned by `GetSymbolAlphabet` does not necessarily consist of irreducible polynomial letters, but it simply collects all the entries in the symbol. In order to obtain an alphabet of irreducible letters, the `GetSymbolAlphabet` can be composed with the `SymbolExpand` function,

```
In[13]:=   GetSymbolAlphabet[ SymbolExpand[ S ] ]
---
Out[13]:=  {1-x,x,1+x+x^2}
```

There are various equivalent definitions of symbols in the literature. An important feature of the implementation of the symbol map in PolyLogTools is that it does not put to zero symbol tensors that contain a constant letter, e.g.,

```
In[14]:=   SymbolMap[ Log[2]*Log[x] ]
---
Out[14]:=  CiTi[2,x] + CiTi[x,2]
```

Keeping constant letters in the symbol sometimes provides valuable information on the underlying function, cf., e.g., refs. [28, 74].

PolyLogTools contains another implementation of the symbol map $\mathcal{S}$ based on dissections of decorated rooted polygons attached to MPLs [74]. It can be called through the function `ComputeSymbol`. The result is fully equivalent to applying the function `ToSymbol` to the maximal iteration of the coproduct, e.g.,

```
  In[15]:=    ComputeSymbol[ PolyLog[2, z] ]
  In[16]:=    SymbolMap[ PolyLog[2, z] ]
--------------------------------------------
 Out[15]:=    - CiTi[1-z,z]
 Out[16]:=    - CiTi[1-z,z]
```

While the two implementations are fully equivalent, we mention here that (based on experience) the implementation of `ComputeSymbol` is usually faster for weights less than four, whereas `SymbolMap` works more efficiently for higher weights.[6]

The symbol map assigns to an MPL expression a symbol tensor. The question then naturally arises if any symbol tensor constructed from a certain alphabet can be the symbol of a function. It turns out that this is not the case in general, however, but the symbol tensor

$$S = \sum_{I=(i_1,\ldots,i_n)} c_I\, a_{i_1} \otimes \ldots \otimes a_{i_n}\,, \qquad c_I \in \mathbb{Q}\,, \tag{7.5}$$

is the symbol of a function, i.e., there is a function F such that $\mathcal{S}(F) = S$, if and only if it satisfies the *integrability condition*

$$\sum_{I=(i_1,\ldots,i_n)} c_I\, a_{i_1} \otimes \ldots \otimes a_{i_{p-1}} \otimes a_{i_{p+2}} \otimes \ldots \otimes a_{i_n}\, d\log a_{i_p} \wedge d\log a_{i_{p+1}} = 0\,, \tag{7.6}$$

for every consecutive pair of indices (i_p, i_{p+1}), $1 \le p < n$ and $\wedge$ denotes the usual wedge product of differential forms. For every value of p, the integrability condition translates into a system of linear constraints on the coefficients c_I. PolyLogTools can generate these constraints for a given symbol `S` via the command `IntegrablityCondition[ S, p ]`. We emphasise that PolyLogTools does not attempt to solve the integrability constraints, because these linear constraints usually give rise to very large linear systems whose solution may require dedicated algorithms and/or specialised software.

7.2 The shuffle Hopf algebra of symbols

The symbol map is not only linear, but it also preserves the multiplication of MPLs and maps a product of MPLs to the shuffle product of their symbol tensors,

$$\mathcal{S}(x \cdot y) = \mathcal{S}(x) \sqcup\!\sqcup\, \mathcal{S}(y)\,. \tag{7.7}$$

In Section 6.1 we have seen that every shuffle algebra is naturally a Hopf algebra, with the coproduct and antipode given by the deconcatenation and reversal of words,

[6] Note that `ComputeSymbol` is only implemented through weight six, while `SymbolMap` has in principle support through the same weight as `Delta`, i.e. weight twelve.

cf. eqs. (6.9) and (6.10). It follows that the shuffle algebra B of all symbol tensors – integrable or not – forms a Hopf algebra with the deconcatenation coproduct and antipode. It is then interesting to ask what is the relation between the coproduct Δ on MPLs and the deconcatenation coproduct Δ_{dec} on symbol tensors. The shuffle algebra B, however, is too large, because the image of the symbol map is the subspace of B consisting of integrable symbol tensors, which we denote by H^0B. It is easy to check that H^0B is a Hopf subalgebra of B. Moreover, the symbol map preserves the coproducts (and also the antipodes), i.e., $\mathcal{S}$ maps the coproduct of an MPL to the deconcatenation coproduct of its symbol, and similarly for the antipode,

$$\begin{aligned} \Delta_{\text{dec}}(\mathcal{S}(x)) &= (\mathcal{S}\otimes\mathcal{S})\Delta(x) \\ S_{\text{dec}}(\mathcal{S}(x)) &= \mathcal{S}(S(x))\,. \end{aligned} \tag{7.8}$$

The deconcatenation coproduct and antipode are implemented in the code via the functions `DeltaDeconcatenation` and `AntipodeDeconcatenation`, which act on symbol tensors as follows,

```
 In[1]:=   DeltaDeconcatenation[ CiTi[a,b,c] ]
 In[2]:=   AntipodeDeconcatenation[ CiTi[a,b,c] ]
-------------------------------------------------------------
Out[1]:=   CT[1,CiTi[a,b,c]] + CT[CiTi[a],CiTi[b,c]] +
           CT[CiTi[a,b],CiTi[c]] + CT[CiTi[a,b,c],1]
Out[2]:=   - CiTi[c,b,a]]
```

Since H^0B is a graded and connected Hopf algebra, we can apply the results of Section 6.3 and study its Lie coalgebra of indecomposables. First, following eq. (6.15) we define a linear map ρ whose kernel is precisely generated by all shuffles,

$$\rho(a_1\otimes\ldots\otimes a_n) = n\,a_1\otimes\ldots\otimes a_n - ⧢(\text{id}\otimes\rho)\Delta'_{\text{dec}}(a_1\otimes\ldots\otimes a_n)\,. \tag{7.9}$$

The previous definition can be cast in the equivalent form [75, 76]

$$\rho(a_1\otimes\ldots\otimes a_n) = \rho(a_1\otimes\ldots\otimes a_{n-1})\otimes a_n - \rho(a_2\otimes\ldots\otimes a_n)\otimes a_1\,. \tag{7.10}$$

From this map we define a projector $\Pi^2 = \Pi$ by [74]

$$\Pi(a_1\otimes\ldots\otimes a_n) = \frac{1}{n}\,\rho(a_1\otimes\ldots\otimes a_n)\,, \tag{7.11}$$

and the cobracket on symbols is given by (cf. eq. (6.19))

$$\begin{aligned} \delta_{\text{dec}}(a_1\otimes\ldots\otimes a_n) &= (\Pi\otimes\Pi)(1-\tau)\Delta_{\text{dec}}(a_1\otimes\ldots\otimes a_n) \\ &= \sum_{k=1}^{n-1}(a_1\otimes\ldots\otimes a_k)\wedge(a_{k+1}\otimes\ldots\otimes a_n)\,. \end{aligned} \tag{7.12}$$

The projector Π and the deconcatenation cobracket δ_{dec} are implemented via the functions `ProductProjector` and `CobracketDeconcatenation`. They have the same properties as their analogues acting on MPLs described in Section 6.3, so we will not describe them here in more detail.

7.3 The cobracket conjecture, classical and Nielson polylogarithms

In ref. [18] a conjectural criterion on the symbol of a function was presented that allows one to determine if a function of weight four can be expressed in terms of classical polylogarithms only, and it was applied to the analytic result for the two-loop six-point remainder function in planar $\mathcal{N} = 4$ Super Yang-Mills of ref. [77, 78]. The criterion can be concisely stated in terms of the cobracket δ_{dec}: if T is a polylogarithmic function of weight four such that its symbol satisfies

$$\delta_{\text{dec},2,2}(\mathcal{S}(T)) = 0\,, \tag{7.13}$$

then T can be expressed in terms of classical polylogarithms only. The cobracket on symbol tensors of length $n = 4$ has a particularly simple form,

$$\delta_{\text{dec},2,2}(a_1 \otimes a_2 \otimes a_3 \otimes a_4) = \frac{1}{4}\,(a_1 \wedge a_2) \wedge (a_3 \wedge a_4)\,. \tag{7.14}$$

It is of course interesting to speculate how this criterion extends beyond weight four. Naive speculation would lead to the guess that if T is a polylogarithmic expression weight n such that $\delta_{\text{dec},p,n-p}(\mathcal{S}(T)) = 0$ for all $1 < p < n-1$, then T can be expressed (possibly up to terms in the kernel of $\mathcal{S}$) in terms of classical polylogarithms only. This extension, however, seems to be too naive, because already at weight five it is not known how to express the Nielsen polylogarithm $S_{2,3}(x)$ in terms of classical polylogarithms only, even though $\delta_{\text{dec},2,3}(\mathcal{S}(S_{2,3}(x))) = 0$. Here $S_{n,p}(x)$ denotes the Nielsen polylogarithm [79],

$$S_{n,p}(x) = (-1)^p\, G(\underbrace{0,\ldots,0}_{n\text{ times}},\underbrace{1,\ldots,1}_{p\text{ times}};x)\,. \tag{7.15}$$

Instead, a folklore conjecture (which can be proven in some cases [80]) states that[7]

> If T is a polylogarithmic expression of weight n such that $\delta_{\text{dec},p,n-p}(\mathcal{S}(T)) = 0$ for all $1 < p < n-1$, then T can be expressed (possibly up to terms in the kernel of $\mathcal{S}$) in terms of Nielsen polylogarithms only.

It would be interesting to have a sharper extension of the criterion of ref. [18], which allows one to determine if a function can be expressed in terms of classical polylogarithms only even at higher weight. In the following we present such a conjecture. As a starting point, we note that the cobrackets in eqs. (6.19) and (7.12) are related by

$$\delta_{\text{dec}}\mathcal{S} = (\Pi\mathcal{S} \otimes \Pi\mathcal{S})\delta\,. \tag{7.16}$$

In the remainder of this section we give evidence for the following conjecture:

> If T is a polylogarithmic expression of weight n such that $\delta_{p,n-p}(T) = 0$ for all $1 < p < n-1$, then T can be expressed in terms of classical polylogarithms only.

[7] We thank Steven Charlton for clarifying correspondance on this topic.

In the remainder of this section we present some evidence for this conjecture. In particular, we show through weight six relations that relate different Nielsen polyogarithms up to classical polylogarithms.

First we note that the conjecture is always true up to weight three, in agreement with the fact that up to weight three all MPLs can be expressed in terms of classical polylogarithms only [1, 81, 82]. More generally, it is known that $S_{1,n-1}(x)$ and $S_{n-1,1}(x)$ can always be expressed in terms of classical polylogarithms, and indeed we have

$$\delta_{p,n-p}(S_{1,n-1}(x)) = \delta_{p,n-p}(S_{n-1,1}(x)) = 0\,, \text{ for all } 1 < p < n-1\,. \tag{7.17}$$

Next let us discuss the Nielsen polylogarithm $S_{2,2}(x)$. It is easy to check that $\delta_{2,2}(S_{2,2}(x)) = 0$. We can indeed express $S_{2,2}(x)$ in terms of classical polylogarithms only,

$$\begin{aligned} S_{2,2}(x) &= \mathrm{Li}_4(x) - \mathrm{Li}_4(1-x) + \mathrm{Li}_4\left(\frac{x}{x-1}\right) - \mathrm{Li}_3(x)\log(1-x) + \frac{1}{24}\log^4(1-x) \\ &- \frac{1}{6}\log x\,\log^3(1-x) + \frac{1}{2}\zeta_2\,\log^2(1-x) + \zeta_3\,\log(1-x) + \zeta_4\,. \end{aligned} \tag{7.18}$$

At weight five, we find for the first time Nielsen polylogarithms that have a non-trivial cobracket,

$$\begin{aligned} \delta_{2,3}(S_{2,3}(x)) &= -\delta_{3,2}(S_{2,3}(x)) = \frac{1}{2}\,G(0,1;x)\wedge\zeta_3 - \frac{1}{2}\,G(1,0;x)\wedge\zeta_3\,, \\ \delta_{2,3}(S_{3,2}(x)) &= -\delta_{3,2}(S_{3,2}(x)) = \frac{1}{2}\,G(0,1;x)\wedge\zeta_3 - \frac{1}{2}\,G(1,0;x)\wedge\zeta_3\,. \end{aligned} \tag{7.19}$$

Since classical polylogarithms of weight five lie in the kernel of $\delta_{2,3}$, we conclude from the previous equations that $S_{2,3}(x)$ and $S_{3,2}(x)$ cannot be expressed in terms of classical polylogarithms alone. Note that, since $\mathcal{S}(\zeta_3) = 0$, we have,

$$\delta_{\mathrm{dec},2,3}(\mathcal{S}(S_{2,3}(x))) = \delta_{\mathrm{dec},2,3}(\mathcal{S}(S_{3,2}(x))) = 0\,, \tag{7.20}$$

and so the cobracket computed from the symbol alone does not allow one to reach this conclusion. However, we learn something more from eq. (7.19): we see that $\delta_{2,3}(S_{2,3}(x)) = \delta_{2,3}(S_{3,2}(x))$, and so the conjecture implies that the two functions are equal up to classical polylogarithms. Indeed, we find the following relation,

$$\begin{aligned} S_{3,2}(x) &= S_{2,3}(x) + \mathrm{Li}_5(x) + \mathrm{Li}_5(1-x) + \mathrm{Li}_5\left(\frac{x}{x-1}\right) - \mathrm{Li}_4(1-x)\,\log(1-x) \\ &+ \mathrm{Li}_4\left(\frac{x}{x-1}\right)\log(1-x) - \frac{1}{2}\,\mathrm{Li}_3(x)\,\log^2(1-x) + \frac{1}{30}\log^5(1-x) \\ &- \frac{1}{8}\,\log x\,\log^4(1-x) + \frac{1}{3}\,\zeta_2\,\log^3(1-x) + \frac{1}{2}\,\zeta_3\,\log^2(1-x) - \zeta_5\,. \end{aligned} \tag{7.21}$$

Finally, let us analyse weight six. We find

$$\delta_{2,4}(S_{2,4}(x)) = \delta_{2,4}(S_{4,2}(x)) = \delta_{2,4}(S_{3,3}(x)) = 0\,, \tag{7.22}$$

and

$$\begin{aligned}
\delta_{3,3}(S_{2,4}(x)) &= \frac{1}{3}\,G(1,0,1;x)\wedge\zeta_3 - \frac{1}{3}\,G(1,1,0;x)\wedge\zeta_3\,,\\
\delta_{3,3}(S_{4,2}(x)) &= -\delta_{3,3}(S_{2,4}(1-x))\,,\\
\delta_{3,3}(S_{3,3}(x)) &= -\delta_{3,3}(S_{2,4}(x)+S_{4,2}(x))\,.
\end{aligned} \tag{7.23}$$

In agreement with our conjecture, we find that we can express $S_{4,2}(x)$ and $S_{3,3}(x)$ up to classical polylogarithms in terms of $S_{2,4}(x)$ and $S_{2,4}(1-x)$. The explicit expressions are

$$\begin{aligned}
S_{4,2}(x) &= -S_{2,4}(1-x) - S_{2,3}(1-x)\,\log x - \mathrm{Li}_5(x)\,\log(1-x) + \frac{1}{2}\,\mathrm{Li}_4(x)\,\log^2 x\\
&- \frac{1}{2}\,\mathrm{Li}_4(1-x)\,\log^2 x + \frac{1}{2}\,\mathrm{Li}_4\left(\frac{x}{x-1}\right)\,\log^2 x + \frac{1}{3}\,\mathrm{Li}_3(1-x)\,\log^3 x\\
&- \frac{1}{12}\,\log^3 x\,\log^3(1-x) - \frac{1}{3}\,\zeta_3\,\log^3 x + \frac{1}{2}\,\zeta_3\,\log^2 x\,\log(1-x) + \zeta_5\,\log(1-x)\\
&+ 2\,\zeta_5\,\log x - \zeta_2\,\zeta_3\,\log x - \frac{1}{3}\,\zeta_2\,\log^3 x\,\log(1-x) + \frac{1}{4}\,\zeta_2\,\log^2 x\,\log^2(1-x)\\
&+ \frac{1}{2}\,\zeta_4\,\log^2 x + \frac{1}{48}\,\log^2 x\,\log^4(1-x) + \frac{5}{48}\,\log^4 x\,\log^2(1-x)\\
&+ \zeta_4\,\log x\,\log(1-x) - \frac{1}{2}\,\zeta_3^2 + \frac{3}{4}\,\zeta_6\,,
\end{aligned} \tag{7.24}$$

and

$$\begin{aligned}
S_{3,3}(x) &= S_{2,4}(x) - S_{2,4}(1-x) + \mathrm{Li}_6(1-x) - \mathrm{Li}_6(x) - \mathrm{Li}_6\left(\frac{x}{x-1}\right)\\
&- S_{2,3}(1-x)\,\log x - \mathrm{Li}_5(1-x)\,\log(1-x) - \mathrm{Li}_5(x)\,\log(1-x)\\
&- \mathrm{Li}_5\left(\frac{x}{x-1}\right)\,\log(1-x) + \frac{1}{2}\,\mathrm{Li}_4(1-x)\,\log^2(1-x)\\
&- \frac{1}{2}\,\mathrm{Li}_4\left(\frac{x}{x-1}\right)\,\log^2(1-x) - \frac{1}{2}\,\mathrm{Li}_4(1-x)\,\log^2 x + \frac{1}{2}\,\mathrm{Li}_4(x)\,\log^2 x\\
&+ \frac{1}{2}\,\mathrm{Li}_4\left(\frac{x}{x-1}\right)\,\log^2 x + \frac{1}{6}\,\mathrm{Li}_3(x)\,\log^3(1-x) + \frac{1}{3}\,\mathrm{Li}_3(1-x)\,\log^3 x\\
&- \frac{1}{72}\,\log^6(1-x) + \frac{1}{20}\,\log x\,\log^5(1-x) - \frac{1}{12}\,\log^3 x\,\log^3(1-x)\\
&+ \frac{1}{48}\,\log^2 x\,\log^4(1-x) + \frac{5}{48}\,\log^4 x\,\log^2(1-x) - \frac{1}{8}\,\zeta_2\,\log^4(1-x)\\
&- \frac{1}{3}\,\zeta_2\,\log^3 x\,\log(1-x) + \frac{1}{4}\,\zeta_2\,\log^2 x\,\log^2(1-x) - \frac{1}{6}\,\zeta_3\,\log^3(1-x)\\
&- \frac{1}{3}\,\zeta_3\,\log^3 x + \frac{1}{2}\,\zeta_3\,\log^2 x\,\log(1-x) + \frac{1}{2}\,\zeta_4\,\log^2 x + \zeta_4\,\log x\,\log(1-x)\\
&+ \zeta_5\,\log(1-x) + 2\,\zeta_5\,\log x - \zeta_2\,\zeta_3\,\log x - \frac{1}{2}\,\zeta_3^2 - \frac{1}{4}\,\zeta_6\,.
\end{aligned} \tag{7.25}$$

Table 4: Symbols	
`SymbolMap[expr]`	Computes the symbol of `expr` as the maximal iteration of the coproduct.
`ComputeSymbol[expr]`	Computes the symbol of `expr` from dissections decorated polygons.
`ToSymbol[expr]`	If `expr` is a maximal iteration of a coproduct, then `ToSymbol` turns it into a symbol expression, by removing the log-signs.
`CiTi[a1,...,an]`	Represents the symbol tensor $a_1 \otimes \ldots \otimes a_n$.
`SymbolExpand[sym]`	Maximally factors all the polynomial symbol letter and uses the additivity of the symbol `sym` to expand them out.
`SymbolFactor[sym]`	Attempts to combine terms into products and ratios in the symbol. Works best if the coefficients of the symbol are integers.
`GetSymbolAlphabet[sym]`	Returns the symbol alphabet of the symbol `sym`.
`IntegrabilityCondition[sym, p]`	Returns the constraint from the integrability condition applied to the factors p and $p+1$ in the symbol `sym`.
`DeltaDeconcatanation[sym]`	Computes the deconcatenation coproduct of the symbol `sym`.
`AntipodeDeconcatanation[sym]`	Computes the deconcatenation antipode of the symbol `sym`.
`ProductProjector[sym]`	Applies the projector Π to the symbol `sym` and projects it to the space of indecomposables.
`CobracketDeconcatenation[sym]`	Computes the cobracket attached to the deconcatenation coproduct of the symbol `sym`.

8 Working with PolyLogTools

So far we have only discussed the implementation of the basic objects – MPLs, their coproduct and symbols – into PolyLogTools, and we have discussed their very basic usage (e.g., how to compute the coproduct or the symbol of an MPL). In this section we describe a set of functions that can be used at runtime to manipulate expressions involving MPLs in Mathematica. Note that since PolyLogTools automatically loads Hpl, all functions from this package are also available. For a description of these functions we refer to refs. [67, 68].

8.1 Manipulating expressions

We start by describing a collection of functions that are useful to manipulate MPL expressions inside Mathematica. A valid MPL expression is a polynomial built out of the functions `G`, `H` (or `HPL`) and `Li` (and the built in Mathematica functions `PolyLog` and `Log`), as well as the transcendental constants `MZV`, `Zeta` and `Pi` (as well as certain special

symbols like `HPLs6` defined by HPL [67, 68]). All the functions described in this section can be applied to any valid MPL expression.

When working with big expressions, it is often useful to focus on certain subexpressions, or to read out the elementary building blocks (e.g. MPLs) the expression is composed of. The function `GetGs` returns a list of all `G`s in the expression. Analogously, `GetGArguments` returns the set of arguments of the `G`, `H` and `HPL` objects in the given expression, while `GetGIndices` returns the entries of the weight vectors of the `G`, `H` and `HPL` objects. For example, we have

```
 In[1]:=   T = 3*G[a,z]+Pi^2*G[b,x]+Zeta[3] + 2*G[a,0,x];
 In[2]:=   GetGs[ T ]
 In[3]:=   GetGArguments[ T ]
 In[4]:=   GetGIndices[ T ]
----------------------------------------------------------
Out[2]:=   { G[a, z], G[b, x], G[a, 0, x] }
Out[3]:=   { 0, a, b, x, z }
Out[4]:=   { 0, a, b }
```

The function `GetWeightTerms` returns the subexpression of all terms of a given weight in an expression. For example,

```
 In[5]:=   GetWeightTerms[ T, 1 ]
 In[6]:=   GetWeightTerms[ T, 2 ]
 In[7]:=   GetWeightTerms[ T, 3 ]
----------------------------------------------------------
Out[5]:=   3*G[a,x]
Out[6]:=   2*G[a,0,x]
Out[7]:=   Pi^2*G[b,x]+Zeta[3]
```

The previous functions allow the user to focus on a specific subset of an expression. There are other functions which allow one to manipulate an expression without changing its value. In Section 4 we have already encountered the functions `ShuffleG`, `StuffleLi`, `DecomposeToLyndonWords` and `ExtractZeroes`. They all fall into this category. Since they have already been discussed in Section 4, we will not discuss them here.

Whenever $a_n \neq 0$, MPLs are invariant under a simultaneous rescaling of their arguments,

$$G(a_1, \ldots, a_n; z) = G(k \cdot a_1, \ldots, k \cdot a_n; k \cdot z)\,, \quad a_n, k \neq 0\,. \tag{8.1}$$

We can use eq. (8.1) to reduce all MPLs with $a_n \neq 0$ to the form $G(\vec{a}; 1)$. If $a_n = 0$, we can extract the trailing zeroes using the shuffle algebra properties before rescaling the last argument to unity. In this way we can always express any expression in terms of MPLs of the form $G(\vec{a}; 1)$ and $G(\vec{0}; z)$, $a_n, z \neq 0$. This operation is implemented in PolyLogTools via the function `NormalizeG`, e.g.,

```
In[8]:=   NormalizeG[ T ]
```

```
Out[8]:=  3*G[a/z,1]+Pi^2*G[b/x,1]+Zeta[3] + 2*(G[0,x]*G[a/x,1] -
          G[0,a/x,1])
```

In applications the arguments of MPLs are often complicated rational or algebraic functions. It is possible to instruct PolyLogTools to simplify the arguments of the `G` functions without touching the rest of the expression. The utility function `GArgumentSimplify` simplifies the arguments of all `G`, `Li`, `PolyLog` and `Log` objects in an expression leaving the rest of the expression unchanged.

Also the coefficients multiplying the MPLs are often functions rather than constants. It can be useful to collect the coefficients of all transcendental quantities (i.e., MPLs and MZVs). While this can in principle be done using the built-in Mathematica function `Collect`, experience shows that this function is rather slow when acting on large expressions. The PolyLogTools function `GatherTranscendentals`, which is built around the Mathematica function `GatherBy`, is usually much faster. It is used as shown in the following example

```
In[9]:=   T = x*G[a,b,x] + G[a,b,x] + x*G[c,x];
In[10]:=   GatherTranscendentals[ T ]
```

```
Out[10]:=   (1+x)*G[a,b,x] + x*G[c,x]
```

It is possible to apply the `Simplify` function to the coefficients multiplying the transcendental quantities using the function `GCoefficientSimplify` (similar to `Collect[..., ..., Simplify]`). It is also possible to collect an expression with respect to the non-transcendental prefactors,

```
In[11]:=   GatherPrefactors[ T ]
```

```
Out[11]:=   G[a,b,x] + x*(G[a,b,x]+G[c,x])
```

8.2 Series expansions of MPLs

Since MPLs are transcendental functions with logarithmic singularities, one is often interested in obtaining the series expansion of an MPL expression close to a (singular) point. If the function $f(z)$ has a logarithmic singularity (branch point) at the point $z = z_0$, then we cannot expand $f(z)$ into a Laurent series in any neighbourhood of the point z_0. Instead, f admits an expansion of the form

$$f(z) = \sum_{k=0}^{n} f_k(z)\,\log^k(z - z_0)\,, \tag{8.2}$$

where the functions $f_k(z)$ are analytic in a neighbourhood of $z = z_0$ and can be expanded into a Laurent series. In the case of MPLs the value of n is bounded by the weight. In

Table 5: Manipulating expressions

`GetGs[expr]`	Returns a set of all `Gs` in `expr`.
`GetGArguments[expr]`	Returns the set of all arguments of the `Gs` in `expr`.
`GetGIndices[expr]`	Returns the set of all entries in the weight vectors of the `Gs` in `expr`.
`GetWeightTerms[expr,n]`	Returns the weight `n` terms in `expr`.
`NormalizeG[expr]`	First applies `ExtractZeroes`, then replaces every $G(a_1,\ldots,a_n;z)$ in `expr` with $G(\frac{a_1}{z},\ldots,\frac{a_n}{z};1)$ if $a_n, z \neq 0$.
`GArgumentSimplify[expr]`	Simplifies the arguments of all `G`, `Li`, `PolyLog` and `Log` functions in `expr`.
`GatherTranscendentals[expr]`	Groups terms in `expr` by transcendental object.
`GatherPrefactors[expr]`	Groups terms in `expr` by rational/algebraic prefactor.
`GCoefficientSimplify[expr]`	Same as `GatherTranscendentals[expr]`, but applies the `Simplify` function to the coefficients.

the following we discuss how to compute the expansion of MPLs of the form $G(\vec{a};z)$ where the weight vector is independent of z around the point $z=0$. In that case the expansion can be obtained in an algorithmic way which has been implemented into PolyLogTools. Expansions around other points $z_0 \neq 0$ can be obtained by letting $z = z_0 - z'$ and working out the functional equations that map MPLs of the form $G(\vec{a};z_0-z')$ to those of the form $G(\vec{a}';z')$. In this way the problem is reduced to finding the expansion around $z'=0$. Finding the required functional equations can often be done in an algorithmic way as well, and we refer for example to refs. [28, 42, 67, 70, 83] and to Section 8.5.

Let us now focus on a single MPL of the form $G(a_1,\ldots,a_n;z)$, where we assume that the a_i are independent of z. This function has a logarithmic singularity at $z=0$ if and only if $a_n=0$. In Section 4 we have seen that we can use the shuffle algebra properties to write any MPL as a linear combination of products of $G(0;z)=\log z$ and MPLs where the last element in the weight vector is non-zero. The result of this operation (implemented in PolyLogTools via the function `ExtractZeroes`) is precisely the decomposition of $G(a_1,\ldots,a_n;z)$ in eq. (8.2) into powers of logarithms multiplied by functions that are analytic at the origin. We have in this way reduced the problem to finding the series expansion of MPLs with $a_n \neq 0$. We can then use eqs. (3.5) and (3.6) to obtain the expansion around $z=0$ ($a_i \neq 0$),

$$\begin{aligned} &G\Big(\vec{0}_{m_1-1},a_1,\ldots,\vec{0}_{m_k-1},a_k;z\Big) \\ &\quad= (-1)^k \sum_{n=1}^{\infty} \frac{1}{n^{m_1}} \left(\frac{z}{a_1}\right)^n Z_{m_2,\ldots,m_k}\left(\frac{a_1}{a_2},\ldots,\frac{a_{k-1}}{a_k};n-1\right), \end{aligned} \tag{8.3}$$

where the coefficients on the right-hand side are written in terms of Z-sums [84], which can

be thought of as a variant of harmonic numbers [85] depending on additional variables,

$$Z_{m_1,\ldots,m_k}(x_1,\ldots,x_k;n) = \sum_{p=1}^{n} \frac{x_1^p}{p^{m_1}}\, Z_{m_2,\ldots,m_k}(x_2,\ldots,x_k;p-1)\ . \tag{8.4}$$

The Z-sums are polynomials. They can be efficiently computed for every value of n in MATHEMATICA using the recursive definition in eq. (8.4) (and caching intermediate sums). Using these steps, we can generate series expansions of MPLs of the form $G(a_1,\ldots,a_n;z)$ up to any desired order.

Series expansions of MPL expressions can be obtained in POLYLOGTOOLS from the function `ExpandPolyLogs` as shown in the following example,

```
In[1]:=    ExpandPolyLogs[ G[a,0,z]/(1-z), {z,0,2} ]
--------------------------------------------------------------
Out[1]:=   (z*(1 - G[0,z]))/a + (z^2*(1 + 4*a - 2*G[0,z] -
           4*a*G[0,z]))/(4*a^2)
```

The argument of `ExpandPolyLogs` can be any expression made out of objects for which the `Series` function can obtain a series representation, as well as MPLs $G(\vec{a};z)$ where the weight vector $\vec{a}$ is independent of z. `ExpandPolyLogs` automatically uses `ExtractZeroes` internally, to extract the logarithmic terms as seen in the above example.

8.3 Differentiation and integration

Two of the basic operations on MPLs are differentiation and integration. In this section we discuss how these operations are implemented in POLYLOGTOOLS.

Let us start by discussing derivatives. From the definition in eq. (3.1) it is easy to see that MPLs satisfy the differential equation

$$\partial_x G(a_1,\ldots,a_n;x) = \frac{1}{x-a_1}\, G(a_2,\ldots,a_n;x)\,. \tag{8.5}$$

The previous equation is only true assuming that the weight vector $\vec{a}$ is independent of x. In applications one often encounters situations where one needs to compute partial derivatives with respect to the arguments of the weight vector. While it is possible to obtain closed formulæ for these partial derivatives [3], in POLYLOGTOOLS partial derivatives of MPLs are evaluated with the help of the coproduct. Indeed, the coproduct of a derivative Δ and the differential operator ∂_x are related through the identity [28, 86]

$$\Delta\partial_x = (\mathrm{id}\otimes\partial_x)\Delta\,. \tag{8.6}$$

Using the fact that the Hopf algebra of MPLs is graded and connected, one can obtain the following formula for the derivative of an MPL expression F of uniform weight n,

$$\partial_x F = m(\mathrm{id}\otimes\partial_x)\Delta_{n-1,1}(F)\,. \tag{8.7}$$

On the right-hand side of this equation the derivative only acts on MPLs of weight one, i.e., ordinary logarithms, for which the derivatives are easily computed.

Partial derivatives are implemented in PolyLogTools through the function DG. This function takes two arguments. The first argument is the expression whose derivative one wishes to compute, and the second argument is the variable with respect to which one differentiates. Moreover, this function satisfies all the basic properties of a derivative (linearity and Leibniz rule) in its first argument, and it evaluates to the built-in Mathematica implementation of the derivative whenever the first argument does not depend on G, H, HPL or Li functions. When acting on an MPL, it evaluates its derivative with respect to the second argument by means of eq. (8.7). The following example illustrates the usage of the function DG,

```
In[1]:=    T = 1/(1-x)*G[1/(1-x),0,0,1,1,1/(1+z)] +
           (1+z)/z*G[-2,3,z];
In[2]:=    GCoefficientSimplify[ DG[ T, x ] ]
In[3]:=    GCoefficientSimplify[ DG[ T, z ] ]
--------------------------------------------------------------------
Out[2]:=   -(G[0,0,1,1,(1 + z)^(-1)]/((-1 + x)*(x + z))) +
           G[-(-1+x)^(-1),0,1,1,(1 + z)^(-1)]/(-1+x)^2 +
           G[-(-1+x)^(-1),0,0,1,1,(1+z)^(-1)]/(-1+x)^2
Out[3]:=   ((1+z)*G[3,z])/(2*z+z^2) - G[-2,3,z]/z^2
           + G[0,0,1,1,(1+z)^(-1)]/(x+z+x*z+z^2)
```

An important property of MPLs is that they behave nicely under integration. Consider the vector space generated by all functions of the form $R(z)\,G(\vec{a};z)$, where the weight vector is independent of z and $R(z)$ is a rational function. This vector space is closed under taking primitives, i.e., for every function $f(z)$ in that vector space there is a function $F(z)$ in the same space such that $\partial_z F = f$. The function F can be found in an algorithmic way using integration by parts (see, e.g., ref. [87]). This algorithm is implemented in PolyLogTools through the function GIntegrate. Its first argument is the integrand f and the second argument the variable z. For example,

```
In[4]:=    T = GIntegrate[ G[0,1,z] z/(a-z)^2, z ]
In[5]:=    GCoefficientSimplify[ DG[ T , z] ]
--------------------------------------------------------------------
Out[4]:=   a*(-(G[0,1,z]/a) - G[0,1,z]/(-a+z) + G[a,1,z]/a) + G[a,0,1,z]
Out[5]:=   G[0,1,z] z/(a-z)^2
```

Whenever the weight vector depends on the integration variable z, or if the integrand involves non-rational algebraic functions of z, the algorithm does not converge and returns an unevaluated expression. We emphasise that this is not a limitation of our implementation, but in these cases the space of MPLs may not be enough to perform all the integrals and more general classes of functions, e.g. of elliptic type, may be required.

In applications one is usually not directly interested in the primitive of a function, but in its integral over a certain range. One can easily evaluate definite integrals of MPL expressions by first computing a primitive and then evaluating the primitive at the end-points of the integration range. Doing so often results in spurious singularities that cancel

in the final result. For example, consider the integral

$$J = \int_0^b \frac{dz}{z-b}\,\left[G(b,a;z) - G(a;b)\,G(b;z) + G(a,b;b)\right] . \tag{8.8}$$

We can formally do the integral by computing a primitive and evaluating it at $z = b$ and $z = 0$,

$$J = G(b,b,a;b) - G(a;b)\,G(b,b;b) + G(a,b;b)\,G(b;b) . \tag{8.9}$$

This last expression is ill-defined because $G(b,\ldots;b)$ is divergent. In Section 4 we have seen how we can use the shuffle algebra properties to replace divergent MPLs by their shuffle regularised versions,

$$\begin{aligned} G(b;b) &= -\zeta_1\,, \\ G(b,b;b) &= \frac{1}{2}\,\zeta_1^2\,, \\ G(b,b,a;b) &= \frac{1}{2}\,\zeta_1^2\,G(a;b) - \zeta_1\,G(a;b) + G(a,b,b;b)\,. \end{aligned} \tag{8.10}$$

It is easy to check that if the previous relations are inserted into eq. (8.9) all powers of the regulator ζ_1 cancel, leaving the finite result

$$J = G(a,b,b;b) . \tag{8.11}$$

All these steps are automated in PolyLogTools, as shown in the following example,

```
 In[6]:=    J = GIntegrate[ (G[b,a,z] - G[b,z] G[a,b] +
            G[a,b,b])/(z-b), z ] /.  z -> b
 In[7]:=    ShuffleRegulate[ J ]
-----------------------------------------------------------
Out[6]:=    G[b,b,a,b] - G[a,b]*G[b,b,b] + G[b,z]*G[a,b,b]
Out[7]:=    G[a,b,b,b]
```

In cases where the integral has not only spurious logarithmic end-point singularities but also poles, replacing MPLs by their shuffle regularised version is not sufficient, and one needs to carefully expand the MPLs around the pole. This can for example be done using the `ExpandPolyLogs` function described in the Section 8.2.

8.4 Numerical evaluation of MPLs

It is important to be able to evaluate MPL expressions, and for this reason a considerable effort has been put by the community in developing fast and reliable computer libraries for the numerical evaluation of (some classes of) MPLs, see for example refs. [67, 68, 83, 88–94].

Given the vast amount of publicly available codes for the evaluation of MPLs, PolyLogTools does not have its own numerical routines for the evaluation of MPLs, but it relies on the Hpl and GiNaC [95] packages. Since the Hpl package is loaded together with PolyLogTools, `HPL` functions can be evaluated numerically as described in the manual of that package [67, 68], and we will not discuss it here any further.

GiNaC is a C++ library for numerical and symbolic computations in high-energy physics. It contains an implementation of the algorithms of ref. [83] for the numerical evaluation of MPLs. The command-line utility `ginsh` allows the user to start an interactive shell-version of GiNaC. We refer to the GiNaC manual for more details [96].

PolyLogTools contains an interface that allows the user to evaluate valid MPL expressions using the interactive `ginsh` environment from within Mathematica. For example,

```
 In[1]:=   T = 1/(1-x)*G[1/(1-x),0,0,1,1,1/(1+z)];
 In[2]:=   Ginsh[ T, {x->0.3, z->0.45} ]
-----------------------------------------------------------------
Out[2]:=   -0.0294179470484662503367597193416603238279
```

The input expression is transformed into a string that is a valid GiNaC expression. This string is written to a temporary file that is then piped through the `ginsh` shell command via the `Run` function in Mathematica. The result returned by `ginsh` is written to a temporary file. The content of this file is imported back into PolyLogTools and returned by the `Ginsh` function. The temporary files are then deleted from the disc. It is possible to instruct PolyLogTools not to delete the temporary files by setting the option `Debug` to `True` (the default is `False`) as shown in the following example:

```
 In[3]:=   Ginsh[ T, {x->0.3, z->0.45}, Debug -> True ]
-----------------------------------------------------------------
Out[3]:=   -0.0294179470484662503367597193416603238279
```

GiNaC allows the user to evaluate MPLs with arbitrary precision, and the user can choose the target number of digits by setting an appropriate flag. The target precision for GiNaC can be chosen dynamically by the user when calling the `Ginsh` function by setting the option `PrecisionGoal`. The default value is 30. For example, the user may increase or decrease the requested precision as shown in the following example:

```
 In[4]:=   Ginsh[ T, {x->0.3, z->0.45}, PrecisionGoal -> 10]
 In[5]:=   Ginsh[ T, {x->0.3, z->0.45}, PrecisionGoal -> 100]
-----------------------------------------------------------------
Out[4]:=   -0.02941794704846625
Out[5]:=   -0.02941794704846625033675971934166032382890897
           19017828790593790231936752156076091770442309841
           11624061172247650989277119
```

Let us make an important comment at this point. MPLs are multi-valued functions, and so care is needed when evaluating MPLs to obtain the correct numerical value. When calling GiNaC, PolyLogTools does not make any assumption on the branch cuts of the functions and it solely relies on the choices made by GiNaC. PolyLogTools merely translates an expression into a format that can be processed by GiNaC, and it is up to the

user to make sure that he or she understands the branch cut structures of the functions that are evaluated before calling GiNaC.

Closely related to numerical evaluation is fitting numerical constants using the PSLQ algorithm [97]. PolyLogTools provides the user with two possible ways to fit transcendental constants. The function `RunPSLQ` is based on the implementation of the PSLQ algorithm in Mathematica by P. Bertok. The source code of this implementation is publicly available from the Wolfram Mathematica library [98] and is shipped together with PolyLogTools. In more recent versions, Mathematica provides a built-in routine to find integer relations via the `FindIntegerNullVector` function. The PolyLogTools function `PSLQFit` relies on this implementation.

Let us illustrate the use of these functions with an example. The transcendental constant $G(0,1;1/2) = -\text{Li}_2(1/2)$ can be expressed as a linear combination of $\log^2 2$ and π^2. The coefficients of this linear combination can be found using the PSLQ algorithm from the numerical value of $G(0,1;1/2)$,

```
In[6]:=   num = Ginsh[ G[0,1,1/2], {} ]
In[7]:=   RunPSLQ[ num, {Log[2]^2, Pi^2}, 20 ]
In[8]:=   PSLQFit[ num, {Log[2]^2, Pi^2}, 20 ]
```

```
Out[6]:=  -0.582240526465012505902656320159680108746
Out[7]:=  -Pi^2/12 + Log[2]^2/2
Out[8]:=  -Pi^2/12 + Log[2]^2/2
```

The last argument of the `RunPSLQ` and `PSLQFit` functions is the number of digits that should be taken into account in the fit. It is usually advisable to evaluate the fit with several different numbers of digits, in order to ensure that the results are stable and that the identified rational coefficients are not accidental.

8.5 Fibration bases

An important part of applying MPLs to the computation of Feynman integrals is determining a combination of MPLs whose symbol matches a given (integrable) symbol tensor. While in general this is a highly complicated task for which no algorithmic solution is known, under certain conditions on the symbol alphabet it is possible to determine an MPL expression with the given symbol in an algorithmic way. In this section we describe such a criterion, and the corresponding algorithm implemented into PolyLogTools.

Consider a symbol alphabet $\mathcal{A}$. We assume that all letters are non-constant rational functions in a set of variables $x_1, \ldots, x_m$. Without loss of generality we may assume that all letters are irreducible polynomials p_i over $\mathbb{Z}$.

Next, we assume that all letters are linear in one of the variables, which we choose to be x_1. Then all the letters in $\mathcal{A}$ can be written in the form

$$p_i(x_1, \ldots, x_m) = a_i(x_2, \ldots, x_m)\, x_1 + b_i(x_2, \ldots, x_m)\,. \tag{8.12}$$

Following ref. [99], we define the set $\mathcal{A}_{x_1}$ as the set consisting of all irreducible non-constant polynomial factors in a_i, b_i and $a_i b_j - a_j b_i$. If all the polynomials in $\mathcal{A}_{x_1}$ are linear in one

of the variables, say x_2, we can iterate the procedure and construct the set $\mathcal{A}_{x_1,x_2}$. We say that a symbol is *linearly reducible* [99] if we can find an ordering of the variables (for example the natural ordering $(x_1,\ldots,x_m)$), which allows us to iterate this procedure until the end, i.e., we can find at every step a variable x_{k+1} such that all polynomials in $\mathcal{A}_{x_1,\ldots,x_k}$ are linear in x_{k+1}. Given an integrable symbol tensor S that is linearly reducible with respect to the ordering $(x_1,\ldots,x_m)$, one can find a function of the form

$$F(x_1,\ldots,x_m) = \sum_{I=(i_1,\ldots,i_m)} c_I\, G(\vec{a}_{1,i_1};x_1)\cdots G(\vec{a}_{m,i_m};x_m)\,, \tag{8.13}$$

such that $\mathcal{S}(F) = S$ and the weight vector $\vec{a}_{k,i_k}$ only involves rational functions in the variables $(x_{k+1},\ldots,x_m)$ [99]. We call the MPLs that appear on the right-hand side of eq. (8.13) a *fibration basis* for the alphabet $\mathcal{A}$ and the ordering $(x_1,\ldots x_m)$. The function F can be constructed in an algorithmic way, cf. [30, 43, 45, 46, 99, 100]. The algorithm implemented in PolyLogTools via the function `FiberSymbol` follows closely the one described in ref. [30]. The function `FiberSymbol` takes two arguments. The first one is an integrable and linearly reducible symbol tensor, and the second one is the list of variables with the chosen ordering. This is illustrated in the following example:

```
In[1]:=    S = CiTi[x,x-y]-CiTi[x,y]+CiTi[x-y,x]-CiTi[y,x];
In[2]:=    FiberSymbol[ S, {x,y} ]
--------------------------------------------------------------
Out[2]:=   G[0,y,x] + G[y,0,x]
```

We stress that the output of `FiberSymbol` depends on the ordering of the variables. For example, if we change the ordering of the variables in the previous example, we obtain

```
In[3]:=    FiberSymbol[ S, {y,x} ]
--------------------------------------------------------------
Out[3]:=   -G[0,x]*G[0,y] + G[0,x]*G[x,y] + 2*G[0,0,x]
```

If the input symbol tensor is not integrable or not linearly reducible with respect to the ordering in the second argument, then the algorithm fails and a warning will be shown.

So far we have only described how to use fibration bases to find a function whose symbol matches a given symbol tensor. In applications it is often useful to write a given MPL expression in terms of a fibration basis. More precisely, consider an MPL expression F with the property that all MPLs have a linearly reducible symbol alphabet with respect to a same ordering of the variables. Then the user can use PolyLogTools to write F in terms of the fibration basis with respect to that ordering. We illustrate this on the following example:

```
In[4]:=    F = G[0,1,1+x,1-y];
In[5]:=    ToFibrationBasis[ F, {x,y} ]
--------------------------------------------------------------
Out[5]:=   Pi^2*G[-1,x]/6+G[0,y]*G[0,-1,x]+G[0,-1,-1,x]-
           G[-1,x]*G[1,0,y]-G[0,-1,-y,x]+G[1,0,0,y]+Zeta[3]
```

Unlike the output of `FiberSymbol`, the output of `ToFibrationBasis` involves also terms proportional to MZVs that are not detected by the symbol. These terms are reconstructed using the coproduct combined with the numerical fitting technique described in ref. [30], where the MZVs are reconstructed by evaluating constant combinations of MPLs numerically at a single point. So far this fitting technique has been implemented through weight six, and therefore the use of `ToFibrationBasis` is limited to expression of weight up to five. Since MPLs are multi-valued functions, the result of the fit can depend crucially on the numerical value used in the fit. By default, each variable is assigned a random value in the range $[0, 1]$. The user can change the numerical values used in the fit through the option `FitValue`:

```
In[6]:=    ToFibrationBasis[ F, {x,y},
               FitValue -> { x-> 0.1, y->0.2} ]
----------------------------------------------------------
Out[6]:=   Pi^2*G[-1,x]/6+G[0,y]*G[0,-1,x]+G[0,-1,-1,x]-
           G[-1,x]*G[1,0,y]-G[0,-1,-y,x]+G[1,0,0,y]+Zeta[3]
```

It is strongly recommended that the user always chooses the numerical value in a way which reflects the applications which he or she has in mind (e.g., in applications the variables are often related to kinematic variables, which take values in a certain range). This is particularly important when the symbol contains letters such as $x - y$ that could lead to polylogarithms that can develop imaginary parts depending on the relative values used for fitting the constants. For large expressions, the reduction to a fibration basis can take a considerable amount of time. It it possible to save intermediate results to the disc by setting the option `Save -> "file.m"`. Intermediate results can later be read in via the option `Input -> "file.m"`.

The function `ToFibrationBasis` is one of the most important features of PolyLogTools. In particular, fibration bases play an important role in the computation of Feynman parameter integrals via direct integration, using ideas similar to those described in refs. [30, 43, 45, 46, 99, 100]. Indeed, if the set of polynomials in the denominator of an integrand is linearly reducible with respect to a given ordering, then we can perform all the integrations one-by-one in that order. At each step one can use the function `ToFibrationBasis` to write all MPLs in the integrand in a form where the integration variable appears in the last argument. Once that is achieved, the integral can be performed easily in an algorithmic way using the `GIntegrate` function.

Another important application of fibration bases is the derivation of transformation formulæ for MPLs. As an example, consider the function $G(-1, -y; x)$, and imagine we want to obtain its series expansion around $y = 1/2$ with $0 < x < 1$. In Section 8.2 we have seen how to use the function `ExpandPolyLogs` to expand MPLs of the form $G(\vec{a}; z)$ around $z = 0$. Hence, if we let $y = 1/2 - z$ and we pass to a fibration basis with respect to the ordering $\{z, x\}$, we can easily obtain the desired expansion. The corresponding piece of code is shown below:

```
In[7]:=   F = G[-1,-y,x];
In[8]:=   Ffiber = ToFibrationBasis[ F/. y->1/2-z, {z,x},
              FitValue -> {x->0.2,z->0.01}];
In[9]:=   ExpandPolyLogs[ Ffiber, {z,0,0} ]
-----------------------------------------------------------
Out[9]:=  G[-1,-1/2,x] + z*(4*G[-1,x]-2*G[-1/2,x])+
          z^2*(-2+2/(1+2*x)+2*G[-1/2,x])
```

Let us conclude by mentioning that the criterion of linear reducibility described above is only a sufficient, but not necessary, condition to find a function that matches a given symbol tensor. The sets $\mathcal{A}_{x_1,\ldots,x_k}$ provide an upper bound on the singularities that can appear after the variables $x_1,\ldots,x_k$ have been integrated out. There are more refined criteria that allow one to obtain a more refined bound on the singularities, and therefore to find fibration bases for larger classes of functions [5, 99]. These more refined criteria, however, have not yet been implemented into PolyLogTools.

9 Single-valued MPLs

Just like the logarithm function, MPLs are multi-valued functions. It is possible to define a variant of MPLs that are real-analytic and single-valued, while preserving most of their algebraic properties. The price to pay is that these functions are no longer holomorphic, i.e., they depend explicitly on the complex conjugated variables. More precisely, *single-valued MPLs* are combinations of MPLs and their complex conjugates such that all discontinuities cancel. The simplest example of such a function is the single-valued version of the logarithm, which is simply given by the logarithm of the absolute value of its argument,

$$\log|z|^2 = \log z + \log\bar{z}\,. \tag{9.1}$$

It is possible to determine the combinations that lead to single-valued MPLs in an algorithmic way. This was done for the first time in ref. [101], where the single-valued versions of HPLs whose weight vectors only contain 0's and 1's have been defined by constructing single-valued solutions to a unipotent differential equation. This construction was extended in ref. [102] to general classes of hyperlogarithms. In refs. [8, 86] it was shown how to define general single-valued MPLs and also MZVs.

Single-valued MPLs are not just of interest in pure mathematics, but they also appear in loop computations. In particular, they show up in the computation of Feynman integrals with massless propagators and three off-shell external legs [29], as well as in the computation of conformal four-point functions in four dimensions [59, 103–106]. In addition, single-valued versions of MPLs are known to describe the multi-Regge limit of scattering amplitudes in planar $\mathcal{N}=4$ Super Yang Mills [55, 107] and the high-energy limit of the dijet cross section in QCD [56, 108]. They also appear in the analytic result for the three-loop corrections to the soft anomalous dimension [109, 110].

In the remainder of this section we present the implementation of single-valued MPLs in PolyLogTools.

Table 6: Manipulating expressions (2)

`ExpandPolyLogs[expr,{x,0,n}]`	If `expr` involves only MPLs of the form $G(\vec{a};x)$, with $\vec{a}$ independent of x, then `expr` is expanded into a series around $x=0$ up to order n.
`Ginsh[expr, list]`	Uses GiNaC to evaluate `expr` numerically. `list` is a replacement list that specifies the numerical values that should be assigned to all the variables in `expr`. `Ginsh` has an option `Debug`. If set to `True` (which is the default), the temporary files containing the GiNaC code are not deleted from the disc.
`RunPSLQ[num, list, prec]`	Uses the PSLQ implementation by P. Bertok with precision `prec` to express `num` as a rational linear combination of the quantities in `list`. The argument `num` must be real.
`PSLQFit[num, list, prec]`	Uses the `FindIntegerNullVector` function to express `num` as a rational linear combination of the quantities in `list`. The argument `num` can be either real or complex. The integer `prec` specifies that all floats should be interpreted as having a precision of `prec` digits.
`DG[expr, x]`	Computes the partial derivative of `expr` with respect to `x`.
`GIntegrate[expr, x]`	Computes the primitive of `expr` with respect to `x`. Only expressions that contain rational functions of `x` and MPLs of the form $G(\vec{a};x)$, with $\vec{a}$ independent of x are allowed inside `expr`.
`ShuffleRegulate[expr]`	Replaces all divergent MPLs in `expr` by their shuffle-regularised version.

9.1 Single-valued MPLs in PolyLogTools

In refs. [8, 86, 101, 102] (see also ref. [55]) a map $\mathbf{s}$ was constructed which associates to $G(\vec{a};z)$ its single-valued version $\mathcal{G}(\vec{a};z)$. This map can be given explicitly in terms of the coproduct and the antipode on MPLs (see Section 6),

$$\mathbf{s}(x) = m(\tilde{S}\otimes \mathrm{id})\Delta(x)\,, \tag{9.2}$$

where $\tilde{S}$ is related to the the antipode of the complex conjugate of x, up to a sign,

$$\tilde{S}(x) = (-1)^{|x|}\,S(\bar{x})\,, \tag{9.3}$$

and $|x|$ is the weight of x. The map $\mathbf{s}$ is obviously linear, and it also preserves the multiplication,

$$\mathbf{s}(x_1\cdot x_2) = \mathbf{s}(x_1)\cdot\mathbf{s}(x_2)\,. \tag{9.4}$$

The image of any MPL expression under $\mathbf{s}$ is single-valued. Note that $\mathbf{s}$ does not only act on functions, but also on numbers. In particular, it sends to zero all powers of π, and acts

Table 8: Manipulating expressions (3)

`FiberSymbol[sym, list]`	Returns a combination of MPLs in a fibration basis with respect to the variables and the ordering specified in list whose symbol matches `sym`. The symbol `sym` is assumed integrable and linearly reducible.
`ToFibrationBasis[expr, list]`	Returns `expr` in a fibration basis with respect to the variables and the ordering specified in list. This function has several options decribed below.
`FitValue`	Option of `ToFibrationBasis`. A list of replacements which specify the numerical values of the all the variables used in the numerical fit. The default is `Automatic`, which assigns random values between 0 and 1 to each variable.
`Save`	Option of `ToFibrationBasis`. The value is a string (default: the empty string). If the string is non-empty, then intermediate results of `ToFibrationBasis` are written to the file whose name is the specified string.
`Input`	Option of `ToFibrationBasis`. The value is a string (default: the empty string). If the string is non-empty, then the file specified by the string is read in.
`ProgressIndicator`	Option of `ToFibrationBasis`. If set to `True`, a dynamically updated text indicates how many MPLs still need to be converted to the fibration basis.

on odd MZVs of depth one in a simple way,

$$\mathbf{s}(\pi) = 0 \quad \text{and} \quad \mathbf{s}(\zeta_{2n+1}) = 2\zeta_{2n+1}\,. \tag{9.5}$$

The single-valued MPLs $\mathcal{G}(a_1,\ldots,a_n;z)$ are represented in PolyLogTools by the symbols `cG[a1,...,an,z]`. The map $\mathbf{s}$ is implemented as the function `SV`, which can act on any MPL expression and returns its single-valued version. We illustrate this with the following example,

```
In[1]:=   T = G[0,0,1,z] + 4*Pi^2*G[1,z] + 3*Zeta[3];
In[2]:=   SVT = SV[ T ]

Out[2]:=  cG[0,0,1,z] + 6*Zeta[3]
```

Via eq. (9.2), single-valued MPLs can be expressed as linear combinations of products of MPLs and their complex conjugates. It is possible to replace all `cG` objects by these combinations as shown in the following example,

```
In[3]:=    cGToG[ SVT ]

Out[3]:=   G[0,0,1,z]+G[1,0,0,Conjugate[z]]+G[1,Conjugate[z]]*G[0,0,z]+
           G[0,z]*G[1,0,Conjugate[z]]+6*Zeta[3]
```

It is possible to work with the `cG` functions in very much the same way in PolyLogTools as with ordinary MPLs. All functions to manipulate MPL expressions described in Sections 4 and 8.1 can be used in the same way with `G` and `cG` functions. In particular, since the map $\mathbf{s}$ respects the multiplication (cf. eq. (9.4)) single-valued MPLs form a shuffle algebra, and we can decompose them into a Lyndon word basis, extract trailing zeroes, etc., just like for the `G` functions.

It is also possible to differentiate and integrate single-valued MPLs. The map $\mathbf{s}$ commutes with holomorphic differentiation [86, 101, 102], $\partial_z \mathbf{s} = \mathbf{s}\partial_z$, so that we can compute derivatives in the same way as for ordinary MPLs. In particular, the function `DG` can be used just like for `G` functions to compute the holomorphic derivatives of single-valued MPLs. It is also possible to evaluate single-valued MPLs by means of the `Ginsh` function, for example,

```
In[4]:=    Ginsh[ SVT, {z -> 0.1 + 0.2*I} ]

Out[4]:=   6.3826710439675724574308460148363187949 +
           1.6091612363338346436180592017136727938I*I
```

While the map $\mathbf{s}$ commutes with holomorphic differentiation, the situation is slightly more subtle when it comes to integration. Indeed, let us compute the following (holomorphic) primitive, and use the fact that single-valued MPLs can be written as ordinary MPLs for which we know how to compute primitives. We find

$$\begin{aligned}
\int \frac{dz}{z-1}\,\mathcal{G}(0;z) &= \int \frac{dz}{z-1}\,[G(0;z)+G(0;\bar{z})] \\
&= G(1,0;z)+G(0;\bar{z})\,G(1;z) \\
&\neq \mathcal{G}(1,0;z) = G(1,0;z)+G(0;\bar{z})\,G(1;z)+G(0,1;\bar{z})\,.
\end{aligned} \tag{9.6}$$

We see that the primitive of a single-valued function, if computed in a naive way, is not single-valued, and the difference is precisely the antiholomorphic function $G(0,1;\bar{z})$. We should keep in mind, however, that the primitive of a function is not unique. In particular, we can add to a holomorphic primitive any antiholomorphic function, and so we can find a single-valued primitive, albeit not the one that we would have naively constructed. This is a general fact: similarly to ordinary MPLs, the space of single-valued MPLs (multiplied by rational functions) is closed under taking primitives. The single-valued primitive of a single-valued MPL expression can be computed in PolyLogTools via the `cGIntegrate` function. This function works in the same way as its non-single-valued analogue `GIntegrate` described in Section 8.3, and so we do not describe it here any further.

In Section 6.2 we have seen that ordinary MPLs modulo their discontinuities form a Hopf algebra. Since we have a map that assigns to an MPL its single-valued version, it is

natural to ask if the single-valued MPLs themselves form a Hopf algebra. This is indeed the case, and single-valued MPLs form a graded and connected Hopf algebra with a coproduct Δ^{sv} and antipode S^{sv} given by the same formulas as the coproduct Δ and the antipode S on MPLs of Section 6.2, except that all MPLs need to be replaced by their single-valued versions. The coproduct Δ^{sv} and the antipode S^{sv} are implemented in PolyLogTools via the functions `DeltaSV` and `AntipodeSV`. These functions are completely analogous to the functions `Delta` and `Antipode` of Section 6.2.

Since both coproducts Δ and Δ^{sv} are computed via very similar formulas, we find it important to quickly compare the two coproducts. We do this on the example of the function

$$\mathcal{G}(0,1;z) = G(0,1;z) + G(1;\bar{z})\,G(0;z) + G(1,0;\bar{z})\,. \tag{9.7}$$

The coproduct Δ^{sv} acts on single-valued MPLs through the same formula as Δ on ordinary MPLs. We the find

$$\Delta^{\text{sv}}(\mathcal{G}(0,1;z)) = \mathcal{G}(0,1;z)\otimes 1 + 1\otimes\mathcal{G}(0,1;z) + \mathcal{G}(1;z)\otimes\mathcal{G}(0;z)\,. \tag{9.8}$$

In particular, by construction, Δ^{sv} only involves single-valued MPLs. The coproduct Δ, instead, acts on the MPLs on the right-hand side of eq. (9.7). We find

$$\begin{aligned}\Delta(\mathcal{G}(0,1;z)) &= \mathcal{G}(0,1;z)\otimes 1 + 1\otimes\mathcal{G}(0,1;z) + \mathcal{G}(1;z)\otimes G(0;z)\\ &\quad + \mathcal{G}(0;z)\otimes G(1;\bar{z})\,.\end{aligned} \tag{9.9}$$

We see that for Δ only the first entries in the coproduct are single-valued. The two coproducts are thus genuinely different.

Let us now discuss single-valued fibration bases. Since the map $\mathbf{s}$ preserves the algebra structure, single-valued MPLs satisfy the same relations as their non-single-valued analogues (with all factors of π removed). This implies in particular that single-valued MPLs can be expressed in terms of a fibration basis under the same conditions and in the same way as ordinary MPLs, simply by acting with $\mathbf{s}$ on the corresponding relation among ordinary MPLs. For this reason, the function `ToFibrationBasis` can be applied in the same way to single-valued MPLs as to ordinary ones as described in Section 8.5.

We conclude by mentioning that more general classes of single-valued MPLs show up in Feynman integral computations, where the symbol letters are neither holomorphic nor anti-holomorphic [29, 56, 59, 104, 106]. While it is also possible to use PolyLogTools to construct these functions using the algorithm of ref. [106] (cf., e.g., ref. [29, 56, 104]), this algorithm is not implemented in PolyLogTools in an automated way. We mention, however, that these functions can be obtained in an automated way from the Maple package HyperlogProcedures [105].

9.2 The Lie coalgebra of clean single-valued functions

Since single-valued MPLs form a graded and connected Hopf algebra, we can use the results of Section 6.3 and construct a projector P^{sv} to the Lie coalgebra of indecomposables of

Table 8: Manipulating expressions (3)

`cG[a1,...,an,z]`	The single-valued MPL $\mathcal{G}(a_1,\ldots,a_n;z)$.
`cC[a1,...,an,z]`	The clean single-valued MPL $\mathcal{C}(a_1,\ldots,a_n;z)$.
`SV[expr]`	Applies the map `s` to `expr`.
`cGToG[expr]`	Replaces all `cG` functions in `expr` by ordinary MPLs and their complex conjugates.
`cCTocG[expr]`	Replaces all `cC` functions in `expr` by single-valued MPLs.
`cCToG[expr]`	Equivalent to `cGToG[cCTocG[expr]]`.
`cGIntegrate[expr, x]`	Computes the single-valued primitive of `expr` with respect to `x`. Only expressions that contain rational functions `x` and MPLs of the form $\mathcal{G}(\vec{a};x)$, with $\vec{a}$ independent of x are allowed inside `expr`.
`DeltaSV[expr]`	Applies the coproduct Δ^{sv} to `expr`.
`AnitpodeSV[expr]`	Applies the antipode S^{sv} to `expr`.
`ProductProjectorSV[expr]`	Applies the projector P^{sv} to `expr`.
`CobracketSV[expr]`	Applies the cobracket δ^{sv} to `expr`.

this Hopf algebra. The projector is constructed recursively following eq. (6.15) using the coproduct. We first define the map

$$R^{\text{sv}}(x) = n\,x - m(\text{id}\otimes R^{\text{sv}})\Delta^{\text{sv}\prime}(x)\,, \tag{9.10}$$

where x has weight n and we let $P^{\text{sv}}(x) = \frac{1}{n}\,R^{\text{sv}}$.

In ref. [71] the following clean single-valued MPLs have been defined,

$$\mathcal{C}(a_1,\ldots,a_n;z) = P^{\text{sv}}(\mathcal{G}(a_1,\ldots,a_n;z))\,. \tag{9.11}$$

The images of single-valued MPLs under the projector P^{sv} have very interesting properties. In particular, they satisfy clean functional relations, i.e. the same relations as the $\mathcal{G}$ functions, but with all product terms removed [71]. The clean single-valued functions are represented by the symbols `cC[a1,...,an,z]`, and the projector P^{sv} is called `ProductProjectorSV`. The clean single-valued functions can be expressed in terms of single-valued MPLs. This can be achieved via the function `cCTocG`. Moreover, all the functions from Sections 4, 8.1 and 8.5 can also be applied to the clean versions. Finally, there is a cobracket $\delta^{\text{sv}} = (P^{\text{sv}}\otimes P^{\text{sv}})(1-\tau)\Delta^{\text{sv}}$ acting on these functions. The cobracket is implemented via the function `CobracketSV`.

10 Validation

PolyLogTools has already been applied to many computations involving MPLs that have led to publications in peer-reviewed journals, both in physics and in mathematics. In this section we review these applications. The reason for doing this is twofold. First, these applications show the versatility of the code, and they can serve as examples to the variety

of problems to which PolyLogTools can be applied. Second, these examples serve at the same time as validation of the code.

PolyLogTools was applied for the first time in ref. [28], where the coproduct on MPLs was used to simplify the two-loop amplitudes for the production of a Higgs boson in association with three partons [111].

The most prominent application of PolyLogTools is the computation of the N^{3}LO corrections to Higgs production in gluon-fusion [33, 35], where the package was used extensively to compute boundary conditions for differential equations [30, 112] and to evaluate phase space integrals for single-emission contributions [31, 113]. It was also used in ref. [114] to evaluate convolution integrals that appear in the mass-factorisation formulas beyond LO. Further phenomenological applications to Higgs physics include the computation of the two-loop amplitudes for Higgs production in the Standard Model Effective Field Theory [115] and the top-Yukawa contributions to bbH production [116].

PolyLogTools was successfully used to compute integrated counterterms for the CoLoRFulNNLO subtraction method [117–119]. It was also used in the context of Lattice QCD in the computation of certain one-loop integrals that appear in the massive momentum-subtraction scheme [120]. More recently, the package was used to manipulate MPLs that appear in the computation of four and five-gluon scattering using the numerical unitarity approach [121–124].

The package was not only applied to higher-order computations relevant to collider physics, but it has also seen various more formal applications. In particular, it has played a crucial role in the analytic computation of the three-loop soft anomalous dimension matrix [109]. It has also been used to compute certain two-loop triangles [29] and conformal four-point functions [104] in terms of single-valued MPLs. Single-valued MPLs are known to show up in multi-Regge kinematics, and PolyLogTools was also successfully applied in that context, both in planar $\mathcal{N} = 4$ SYM [55, 57, 58, 107] and in QCD [56, 108]. Recently, it was used to manipulate analytic expressions for two-loop form factors in $\mathcal{N} = 4$ SYM [125, 126]. The package was also used to compute and analyse the cuts of one and two-loop integrals [61, 127, 128] and to study their relationship to the coaction of Feynman integrals [62, 63]. Finally, (a private extension of) PolyLogTools was used to evaluate various Feynman integrals that evaluate to elliptic generalisations of MPLs [129–132].

PolyLogTools was not only used to perform research in theoretical and mathematical physics, but it has led to new results in pure mathematics. It was applied in refs. [11, 12] to study reduction identities for MPLs to lower depth, and in refs. [9, 10] to study some properties of MPLs of weight four, in particular to derive a novel functional relation for MPLs of weight four involving more than 100 terms. Finally, it was used in ref. [71] to define and study some of the clean single-valued MPLs reviewed in Section 9.2.

11 An example calculation

In the following we present a sample computation using PolyLogTools that illustrates the most important features of the package. We compute the one-loop four-mass scalar box integral shown in fig. 11. This integral is finite and furthermore one of the simplest and most

prominent examples of dual-conformal integrals appearing for example in $\mathcal{N}=4$ SYM. We stress that these properties are by no means prerequisites for the use of PolyLogTools, in particular the package has been used to compute many infrared-divergent and non-dual-conformal integrals as pointed out in the previous section. However, this integral allows us to focus on the features of PolyLogTools and elaborate on some potential obstructions in the calculation and how to avoid them using PolyLogTools. This example is also described in the notebook `integration_manual.nb` distributed with the source code of the package.

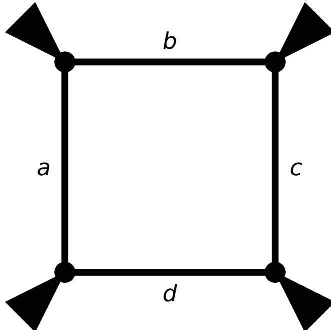

Figure 1. The four-mass box integral.

The four-mass box integral can be written as

$$I^{4m}(u,v) = \int d^4\ell \frac{(a,c)(b,d)\Delta_6[u,v]}{(\ell,a)(\ell,b)(\ell,c)(\ell,d)}\,. \tag{11.1}$$

Here we have used the notation $(a,b)=(x_b-x_a)^2$ that is inspired by the embedding space formalism. The four dual points a,b,c,d are non-light-like separated, so that we can define the two finite dual-conformal cross ratios

$$u = \frac{(a,b)(c,d)}{(a,c)(b,d)} \quad \text{and} \quad v = \frac{(a,d)(b,c)}{(a,c)(b,d)}\,, \tag{11.2}$$

as well as the Gram determinant

$$\Delta_6 = \sqrt{(1-u-v)^2-4uv}\,. \tag{11.3}$$

While it is possible to linearize the Gram determinant by introducing new variables, we will refrain from doing so here, in order to illustrate how to handle the appearance of square roots in the calculation. The starting point for our calculation is the Feynman parametrization of the loop integral, that we write as [50],

```
In[1]:=   f1 = a[1]*u + a[2] + a[3]*v;
In[2]:=   f2 = a[1]*a[2] + a[1]*a[3] + a[2]*a[3];
In[3]:=   F = d6/2/f1/f2;
```

Here the variable `d6` denotes the Gram determinant Δ_6. The integral over the Feynman parameters a_i is over $\mathbb{P}^3$, and we can choose a chart by setting any one of the a_i to one and

integrating the remaining ones over the range $[0,\infty)$. We can then compute the primitive in the first variable as

```
In[4]:=    P = GIntegrate[F/.a[3]->1,a[1]];
```

which yields

$$P = -\frac{\Delta_6\Big[G\left(-\frac{\alpha_2+v}{u};\alpha_1\right) - G\left(-\frac{\alpha_2}{\alpha_2+1};\alpha_1\right)\Big]}{2\left(\alpha_2^2+\alpha_2-\alpha_2 u+\alpha_2 v+v\right)}\,. \tag{11.4}$$

Since none of the resulting polylogarithms have a 0 in their last index, we can immediately set α_1 to zero to see that the integral vanishes on the lower boundary. To derive the result at the upper bound, we need to map $+\infty$ to zero, which we can do by letting $\alpha_1 \to 1/t$. Now we can instruct PolyLogTools to derive the functional identities that are required to express the primitive as a function of t using

```
In[5]:=    Pp = ToFibrationBasis[P/.a[1]->1/t,{t,a[2],u,v}];
```

which yields

$$\begin{aligned}\frac{\Delta_6}{2\left(\alpha_2^2+(1-u+v)\alpha_2+v\right)}\Big[&G\left(-1;\alpha_2\right)-G\left(0;\alpha_2\right)+G\left(-v;\alpha_2\right)\\ &-G\left(0;u\right)+G\left(0;v\right)+G\left(-\tfrac{1+\alpha_2}{\alpha_2};t\right)-G\left(-\tfrac{u}{v+\alpha_2};t\right)\Big]\,.\end{aligned} \tag{11.5}$$

We see that there are no logarithmically divergent polylogarithms with argument t and so we could just set $t=0$ to obtain the value of the integral at the upper bound. It is usually advisable to instead expand the polylogarithms to zeroth order around the desired point. This is more robust as it automatically enforces the cancellation of spurious logarithmic divergences or poles. This can be achieved using

```
In[6]:=    F2 = ExpandPolyLogs[Pp, {t,0,0}];
```

yielding the integrand for the last remaining integration

$$\frac{\Delta_6\Big[G(-1;\alpha_2)-G(0;u)+G(0;v)-G(0;\alpha_2)+G(-v;\alpha_2)\Big]}{2\left(\alpha_2^2+(1-u+v)\alpha_2+v\right)} \tag{11.6}$$

Here we see now a potential obstruction that is not automatically resolved by PolyLogTools: The denominator is quadratic in the next integration variable. In general this can be a fundamental obstruction, in the sense that that it may not be possible to evaluate the integral in terms of MPLs alone. In this case, however, there is only one integration left to do, so the result should be expressible as polylogarithms with algebraic arguments. PolyLogTools cannot automatically resolve this integral though, as it expects to be able to partial fraction the input into terms with only linear denominators. We can however manually solve the quadratic equation in the denominator

$$\alpha_2 = \tfrac{1}{2}\left(u-v-1\pm\sqrt{(1-u-v)^2-4uv}\right)\,, \tag{11.7}$$

and use it to reexpress the denominator in factorized form. At this point we are in principle done and could perform the next integral. We have to be careful to hide the square root from `Mathematica` as the routines for partial fractioning will otherwise restore the original quadratic form. Here we can just identify the root with the gram determinant Δ_6 that we introduced before and thus define the new denominator as

```
In[7]:=   De = 2*(1/2*(1-u+v-d6)+a[2]) *(1/2*(1-u+v+d6)+a[2]);
```

Now the next integration can be performed by invoking

```
In[8]:=   P = GIntegrate[Numerator[F2]/De,a[2]];
```

The resulting primitive is a pure function of weight two that we refrain from spelling out here. Again we need to evaluate the primitive at the boundaries of integration. The lower boundary is immediately found using:

```
In[9]:=   at0 = ExpandPolyLogs[P,a[2],0,0]
--------------------------------------------------
Out[9]:=  0
```

In order to obtain the value at infinity, we once again need to invert the argument of the polylogarithms and use `ToFibrationBasis` to derive the required functional equations. Rather than calling `ToFibrationBasis` on the entire expression, we can also call it on a list of polylogarithms to obtain the individual functional equations. This can be particularly useful when we have a basis of functions for which we want to derive functional equations for that can then be stored. It is also useful if we want to inspect the individual functional equations to make sure that our choice of `FitValue` has not introduced any spurious $i\pi$ terms. In the present case there is the potential for such terms, as the polylogarithms contain spurious branchpoints for $u - v = 0$ and as such the ordering of the variables that is employed when numerically fixing the branches of the logarithms becomes relevant. When we derive the functional equations in the following way we can easily verify that the obtain expressions maintain manifest reality:

```
In[10]:=   Pin = P/.a[2]->1/t;
In[11]:=   R = ToFibrationBasis[GetGs[Pin], {t,d6,u,v},
             FitValue->{t->.1, d6->.12, u->.23, v->.45}];
```

We can then obtain the final result of the integral using

```
In[12]:=   Pin = Pin //.  Dispatch[Thread[GetGs[Pin]->R]];
In[13]:=   result = ExpandPolyLogs[Pin,{t,0,0}]-at0;
```

The four mass box is well known of course and its literature result is

```
In[14]:=   literature = PolyLog[2, ut] + PolyLog[2, vt]
           + 1/2*Log[u]*Log[v] - Log[ut]*Log[vt] - Zeta[2];
```

with `ut = 1/2*(1-u+v+d6)` and `vt = 1/2(1+u-v+d6)`.

We can use the capability of PolyLogTools to compute numerical values for arbitrary polylogarithms to compare our result against the literature reference numerically:

```
In[15]:=    Ginsh[literature - result /.d6->Sqrt[(1-u-v)^2-4*u*v],
            {u->0.1,v->0.23}]
Out[15]:=   0
```

12 Conclusion

In this article we have reviewed the mathematical foundations of multiple polylogarithms and have documented their implementation in the Mathematica package PolyLogTools. Multiple polylogarithms have become ubiquitous in many areas of high-energy physics and the algorithms presented in this paper and implemented in PolyLogTools provide a powerful and flexible way to handle expressions involving multiple polylogarithms. PolyLogTools has already served as the backbone of many recent calculations in high energy physics. Its public release accompanying this paper will enable even more studies of the mathematical structure of scattering amplitudes and calculations of multi-loop amplitudes and cross sections. It also serves as a well-tested reference implementation for more specialized and optimized implementations that might be developed to handle particular situations.

Acknowledgments

We would like to thank Herbert Gangl and Steven Charlton for collaboration on some of the mathematical aspects implemented into the code and for correspondence on the cobracket on MPLs. We also thank Lorenzo Tancredi and Brenda Penante for a careful reading of the manuscript. We would like to thank Enrico Herrmann and Andrew McLeod for suggesting the title and Enrico Hermann for testing the release version of the package. This research was supported by the the ERC grant 637019 “MathAm”, and the U.S. Department of Energy (DOE) under contract DE-AC02-76SF00515.

References

[1] A. Goncharov, *Volumes of hyperbolic manifolds and mixed tate motives*, *Journal of the American Mathematical Society* **12** (1999) 569.

[2] A. B. Goncharov, *Multiple polylogarithms, cyclotomy and modular complexes*, *Math.Res.Lett.* **5** (1998) 497 [1105.2076].

[3] A. B. Goncharov, *Multiple polylogarithms and mixed Tate motives*, *arXiv Mathematics e-prints* (2001) math/0103059 [math/0103059].

[4] A. Goncharov, *Galois symmetries of fundamental groupoids and noncommutative geometry*, *Duke Math.J.* **128** (2005) 209 [math/0208144].

[5] F. C. S. Brown, *On the periods of some feynman integrals*, *arXiv* (2009) [0910.0114].

[6] F. C. Brown, *Multiple zeta values and periods of moduli spaces $\mathfrak{M}_{0,n}$*, *Annales Sci.Ecole Norm.Sup.* **42** (2009) 371 [math/0606419].

[7] F. Brown, *On the decomposition of motivic multiple zeta values*, in *Galois-Teichmüller theory and arithmetic geometry*, vol. 68 of *Adv. Studies in Pure Math.*, pp. 31–58, Math. Soc. Japan, 2012, 1102.1310.

[8] F. Brown, *Single-valued Motivic Periods and Multiple Zeta Values*, *SIGMA* **2** (2014) e25 [1309.5309].

[9] H. Gangl, *Multiple Polylogarithms in Weight 4*, *arXiv* (2016) [1609.05557].

[10] H. Gangl, *The Grassmannian Complex and Goncharov's Motivic Complex in Weight 4*, *arXiv* (2018) [1801.07816].

[11] S. Charlton, *A review of Dan's reduction method for multiple polylogarithms*, *arXiv* (2017) [1703.03961].

[12] S. Charlton, *Identities arising from coproducts on multiple zeta values and multiple polylogarithms*, Ph.D. thesis, University of Durham, 2016.

[13] C. R. Mafra, O. Schlotterer and S. Stieberger, *Complete N-Point Superstring Disk Amplitude II. Amplitude and Hypergeometric Function Structure*, *Nucl. Phys.* **B873** (2013) 461 [1106.2646].

[14] O. Schlotterer and S. Stieberger, *Motivic Multiple Zeta Values and Superstring Amplitudes*, *J. Phys.* **A46** (2013) 475401 [1205.1516].

[15] J. Broedel, O. Schlotterer and S. Stieberger, *Polylogarithms, Multiple Zeta Values and Superstring Amplitudes*, *Fortsch. Phys.* **61** (2013) 812 [1304.7267].

[16] S. Abreu, L. J. Dixon, E. Herrmann, B. Page and M. Zeng, *The two-loop five-point amplitude in $\mathcal{N} = 8$ supergravity*, *JHEP* **03** (2019) 123 [1901.08563].

[17] D. Chicherin, T. Gehrmann, J. M. Henn, P. Wasser, Y. Zhang and S. Zoia, *The two-loop five-particle amplitude in $\mathcal{N} = 8$ supergravity*, *JHEP* **03** (2019) 115 [1901.05932].

[18] A. B. Goncharov, M. Spradlin, C. Vergu and A. Volovich, *Classical Polylogarithms for Amplitudes and Wilson Loops*, *Phys. Rev. Lett.* **105** (2010) 151605 [1006.5703].

[19] J. Golden, A. B. Goncharov, M. Spradlin, C. Vergu and A. Volovich, *Motivic Amplitudes and Cluster Coordinates*, *JHEP* **01** (2014) 091 [1305.1617].

[20] M. Spradlin and A. Volovich, *Symbols of One-Loop Integrals From Mixed Tate Motives*, *JHEP* **11** (2011) 084 [1105.2024].

[21] L. J. Dixon, J. M. Drummond, C. Duhr and J. Pennington, *The four-loop remainder function and multi-Regge behavior at NNLLA in planar $N = 4$ super-Yang-Mills theory*, *JHEP* **06** (2014) 116 [1402.3300].

[22] L. J. Dixon and M. von Hippel, *Bootstrapping an NMHV amplitude through three loops*, *JHEP* **10** (2014) 065 [1408.1505].

[23] L. J. Dixon, M. von Hippel and A. J. McLeod, *The four-loop six-gluon NMHV ratio function*, *JHEP* **01** (2016) 053 [1509.08127].

[24] L. J. Dixon, J. Drummond, T. Harrington, A. J. McLeod, G. Papathanasiou and M. Spradlin, *Heptagons from the Steinmann Cluster Bootstrap*, *JHEP* **02** (2017) 137 [1612.08976].

[25] L. J. Dixon, M. von Hippel, A. J. McLeod and J. Trnka, *Multi-loop positivity of the planar $\mathcal{N} = 4$ SYM six-point amplitude*, *JHEP* **02** (2017) 112 [1611.08325].

[26] S. Caron-Huot, L. J. Dixon, A. McLeod and M. von Hippel, *Bootstrapping a Five-Loop Amplitude Using Steinmann Relations*, *Phys. Rev. Lett.* **117** (2016) 241601 [1609.00669].

[27] S. Abreu, L. J. Dixon, E. Herrmann, B. Page and M. Zeng, *The two-loop five-point amplitude in $\mathcal{N} = 4$ super-Yang-Mills theory*, *Phys. Rev. Lett.* **122** (2019) 121603 [1812.08941].

[28] C. Duhr, *Hopf algebras, coproducts and symbols: an application to Higgs boson amplitudes*, *JHEP* **08** (2012) 043 [1203.0454].

[29] F. Chavez and C. Duhr, *Three-mass triangle integrals and single-valued polylogarithms*, *JHEP* **1211** (2012) 114 [1209.2722].

[30] C. Anastasiou, C. Duhr, F. Dulat and B. Mistlberger, *Soft triple-real radiation for Higgs production at N3LO*, *JHEP* **07** (2013) 003 [1302.4379].

[31] C. Anastasiou, C. Duhr, F. Dulat, F. Herzog and B. Mistlberger, *Real-virtual contributions to the inclusive Higgs cross-section at N^3LO*, *JHEP* **12** (2013) 088 [1311.1425].

[32] F. Dulat and B. Mistlberger, *Real-Virtual-Virtual contributions to the inclusive Higgs cross section at N3LO*, 1411.3586.

[33] C. Anastasiou, C. Duhr, F. Dulat, E. Furlan, T. Gehrmann, F. Herzog et al., *Higgs boson gluon–fusion production at threshold in N^3LO QCD*, *Phys. Lett.* **B737** (2014) 325 [1403.4616].

[34] C. Anastasiou, C. Duhr, F. Dulat, E. Furlan, T. Gehrmann, F. Herzog et al., *Higgs Boson Gluon-fusion Production Beyond Threshold in N^3LO QCD*, *JHEP* **03** (2015) 091 [1411.3584].

[35] C. Anastasiou, C. Duhr, F. Dulat, F. Herzog and B. Mistlberger, *Higgs Boson Gluon-Fusion Production in QCD at Three Loops*, *Phys. Rev. Lett.* **114** (2015) 212001 [1503.06056].

[36] B. Mistlberger, *Higgs boson production at hadron colliders at N^3LO in QCD*, *JHEP* **05** (2018) 028 [1802.00833].

[37] F. Dulat, S. Lionetti, B. Mistlberger, A. Pelloni and C. Specchia, *Higgs-differential cross section at NNLO in dimensional regularisation*, *JHEP* **07** (2017) 017 [1704.08220].

[38] F. Dulat, B. Mistlberger and A. Pelloni, *Differential Higgs production at N^3LO beyond threshold*, *JHEP* **01** (2018) 145 [1710.03016].

[39] F. Dulat, B. Mistlberger and A. Pelloni, *Precision predictions at N^3LO for the Higgs boson rapidity distribution at the LHC*, *Phys. Rev.* **D99** (2019) 034004 [1810.09462].

[40] D. Chicherin, T. Gehrmann, J. M. Henn, P. Wasser, Y. Zhang and S. Zoia, *All master integrals for three-jet production at NNLO*, 1812.11160.

[41] D. Chicherin, J. M. Henn, P. Wasser, T. Gehrmann, Y. Zhang and S. Zoia, *Analytic result for a two-loop five-particle amplitude*, *Phys. Rev. Lett.* **122** (2019) 121602 [1812.11057].

[42] E. Remiddi and J. A. M. Vermaseren, *Harmonic polylogarithms*, *Int. J. Mod. Phys.* **A15** (2000) 725 [hep-ph/9905237].

[43] J. Ablinger, J. Blümlein, C. Raab, C. Schneider and F. Wißbrock, *Calculating Massive 3-loop Graphs for Operator Matrix Elements by the Method of Hyperlogarithms*, *Nucl. Phys.* **B885** (2014) 409 [1403.1137].

[44] E. Panzer, *Feynman integrals and hyperlogarithms*, Ph.D. thesis, Humboldt U., Berlin, Inst. Math., 2015. 1506.07243.

[45] E. Panzer, *Algorithms for the symbolic integration of hyperlogarithms with applications to Feynman integrals*, Comput. Phys. Commun. **188** (2015) 148 [1403.3385].

[46] C. Bogner, *MPL–A program for computations with iterated integrals on moduli spaces of curves of genus zero*, Comput. Phys. Commun. **203** (2016) 339 [1510.04562].

[47] J. L. Bourjaily, A. J. McLeod, M. Spradlin, M. von Hippel and M. Wilhelm, *Elliptic Double-Box Integrals: Massless Scattering Amplitudes beyond Polylogarithms*, Phys. Rev. Lett. **120** (2018) 121603 [1712.02785].

[48] J. L. Bourjaily, A. J. McLeod, M. von Hippel and M. Wilhelm, *Rationalizing Loop Integration*, JHEP **08** (2018) 184 [1805.10281].

[49] J. L. Bourjaily, A. J. McLeod, M. von Hippel and M. Wilhelm, *A (Bounded) Bestiary of Feynman Integral Calabi-Yau Geometries*, Phys. Rev. Lett. **122** (2019) 031601 [1810.07689].

[50] J. L. Bourjaily, F. Dulat and E. Panzer, *Manifestly Dual-Conformal Loop Integration*, 1901.02887.

[51] L. J. Dixon, J. M. Drummond, M. von Hippel and J. Pennington, *Hexagon functions and the three-loop remainder function*, JHEP **1312** (2013) 049 [1308.2276].

[52] J. M. Drummond, G. Papathanasiou and M. Spradlin, *A Symbol of Uniqueness: The Cluster Bootstrap for the 3-Loop MHV Heptagon*, JHEP **03** (2015) 072 [1412.3763].

[53] J. Drummond, J. Foster, Ö. Gürdoğn and G. Papathanasiou, *Cluster adjacency and the four-loop NMHV heptagon*, JHEP **03** (2019) 087 [1812.04640].

[54] F. C. S. Brown, *Polylogarithmes multiples uniformes en une variable*, Compt. Rend. Math. **338** (2004) 527.

[55] V. Del Duca, S. Druc, J. Drummond, C. Duhr, F. Dulat, R. Marzucca et al., *Multi-Regge kinematics and the moduli space of Riemann spheres with marked points*, JHEP **08** (2016) 152 [1606.08807].

[56] V. Del Duca, C. Duhr, R. Marzucca and B. Verbeek, *The analytic structure and the transcendental weight of the BFKL ladder at NLL accuracy*, JHEP **10** (2017) 001 [1705.10163].

[57] V. Del Duca, S. Druc, J. Drummond, C. Duhr, F. Dulat, R. Marzucca et al., *The seven-gluon amplitude in multi-Regge kinematics beyond leading logarithmic accuracy*, JHEP **06** (2018) 116 [1801.10605].

[58] R. Marzucca and B. Verbeek, *The Multi-Regge Limit of the Eight-Particle Amplitude Beyond Leading Logarithmic Accuracy*, 1811.10570.

[59] O. Schnetz, *Graphical functions and single-valued multiple polylogarithms*, Commun. Num. Theor. Phys. **08** (2014) 589 [1302.6445].

[60] F. Brown, *Feynman Amplitudes and Cosmic Galois group*, 1512.06409.

[61] S. Abreu, R. Britto, C. Duhr and E. Gardi, *Cuts from residues: the one-loop case*, JHEP **06** (2017) 114 [1702.03163].

[62] S. Abreu, R. Britto, C. Duhr and E. Gardi, *Algebraic Structure of Cut Feynman Integrals and the Diagrammatic Coaction*, Phys. Rev. Lett. **119** (2017) 051601 [1703.05064].

[63] S. Abreu, R. Britto, C. Duhr and E. Gardi, *Diagrammatic Hopf algebra of cut Feynman integrals: the one-loop case*, *JHEP* **12** (2017) 090 [1704.07931].

[64] D. E. Radford, *A natural ring basis for the shuffle algebra and an application to group schemes*, *Jounal of Algebra* **58** (1979) 432.

[65] W. A. Stein et al., *Sage Mathematics Software (Version 8.6).* The Sage Development Team, 2019.

[66] M. E. Hoffman, *Quasi-shuffle products*, *[arXiv:math/9907173]* (1999) .

[67] D. Maitre, *HPL, a mathematica implementation of the harmonic polylogarithms*, *Comput. Phys. Commun.* **174** (2006) 222 [hep-ph/0507152].

[68] D. Maitre, *Extension of HPL to complex arguments*, *Comput. Phys. Commun.* **183** (2012) 846 [hep-ph/0703052].

[69] K. Ihara, M. Kaneko and D. Zagier, *Derivation and double shuffle relations for multiple zeta values*, *Compositio Math.* **142** (2006) 307.

[70] C. Duhr, *Mathematical aspects of scattering amplitudes*, in *Proceedings, Theoretical Advanced Study Institute in Elementary Particle Physics: Journeys Through the Precision Frontier: Amplitudes for Colliders (TASI 2014): Boulder, Colorado, June 2-27, 2014*, pp. 419–476, 2015, 1411.7538, DOI.

[71] S. Charlton, C. Duhr, F. Dulat and H. Gangl, *To appear.*, .

[72] J. Golden and A. J. Mcleod, *Cluster Algebras and the Subalgebra Constructibility of the Seven-Particle Remainder Function*, *JHEP* **01** (2019) 017 [1810.12181].

[73] K. T. Chen, *Iterated path integrals*, *Bull. Amer. Math. Soc.* **83** (1977) 831.

[74] C. Duhr, H. Gangl and J. R. Rhodes, *From polygons and symbols to polylogarithmic functions*, *JHEP* **1210** (2012) 075 [1110.0458].

[75] R. Ree, *Lie Elements and an Algebra Associated With Shuffles*, *Annals of Mathematics* **68** (1958) 210.

[76] G. Griffing, *Dual Lie Elements and a Derivation for the Cofree Coassociative Coalgebra*, *Proceeding of the American Mathematical Society* **123** (1995) 3269.

[77] V. Del Duca, C. Duhr and V. A. Smirnov, *An Analytic Result for the Two-Loop Hexagon Wilson Loop in $N = 4$ SYM*, *JHEP* **03** (2010) 099 [0911.5332].

[78] V. Del Duca, C. Duhr and V. A. Smirnov, *The Two-Loop Hexagon Wilson Loop in $N = 4$ SYM*, *JHEP* **05** (2010) 084 [1003.1702].

[79] N. Nielsen, *Der Eulersche Dilogarithmus und seine Verallgemeinerungen*, *Nova Acta Leopoldina (Halle)* **90** (1909) .

[80] F. Brown, *unpublished*, .

[81] L. Lewin, *Polylogarithms and associated functions*. North Holland, New York, 1982.

[82] R. Kellerhals, *Volumes in hyperbolic 5–space*, *GAFA* **5** (1995) 640.

[83] J. Vollinga and S. Weinzierl, *Numerical evaluation of multiple polylogarithms*, *Comput. Phys. Commun.* **167** (2005) 177 [hep-ph/0410259].

[84] S. Moch, P. Uwer and S. Weinzierl, *Nested sums, expansion of transcendental functions and multiscale multiloop integrals*, *J. Math. Phys.* **43** (2002) 3363 [hep-ph/0110083].

[85] J. A. M. Vermaseren, *Harmonic sums, Mellin transforms and integrals*, *Int. J. Mod. Phys.* **A14** (1999) 2037 [hep-ph/9806280].

[86] F. Brown, *Notes on Motivic Periods*, *Commun. Num. Theor Phys.* **11** (2017) 557 [1512.06410].

[87] J. Broedel, C. Duhr, F. Dulat and L. Tancredi, *Elliptic polylogarithms and iterated integrals on elliptic curves. Part I: general formalism*, *JHEP* **05** (2018) 093 [1712.07089].

[88] T. Gehrmann and E. Remiddi, *Numerical evaluation of two-dimensional harmonic polylogarithms*, *Comput. Phys. Commun.* **144** (2002) 200 [hep-ph/0111255].

[89] T. Gehrmann and E. Remiddi, *Numerical evaluation of harmonic polylogarithms*, *Comput. Phys. Commun.* **141** (2001) 296 [hep-ph/0107173].

[90] M. Yu. Kalmykov and A. Sheplyakov, *lsjk - a C++ library for arbitrary-precision numeric evaluation of the generalized log-sine functions*, *Comput. Phys. Commun.* **172** (2005) 45 [hep-ph/0411100].

[91] S. Buehler and C. Duhr, *CHAPLIN - Complex Harmonic Polylogarithms in Fortran*, *Comput. Phys. Commun.* **185** (2014) 2703 [1106.5739].

[92] H. Frellesvig, D. Tommasini and C. Wever, *On the reduction of generalized polylogarithms to Li_n and $Li_{2,2}$ and on the evaluation thereof*, *JHEP* **03** (2016) 189 [1601.02649].

[93] H. Frellesvig, *Generalized Polylogarithms in Maple*, 1806.02883.

[94] J. Ablinger, J. Blümlein, M. Round and C. Schneider, *Numerical Implementation of Harmonic Polylogarithms to Weight $w = 8$*, 1809.07084.

[95] C. W. Bauer, A. Frink and R. Kreckel, *Introduction to the GiNaC framework for symbolic computation within the C++ programming language*, *J. Symb. Comput.* **33** (2000) 1 [cs/0004015].

[96] *GiNaC*, *http://www.ginac.de* (2019) .

[97] H. R. P. Ferguson and D. H. Bailey, *A Polynomial Time, Numerically Stable Integer Relation Algorithm*, *RNR Technical Report RNR-91-032* (1992) .

[98] P. Bertok, *Pslq integer relation algorithm implementation*, *http://library.wolfram.com/infocenter/MathSource/4263/* (2004) .

[99] F. Brown, *The Massless higher-loop two-point function*, *Commun. Math. Phys.* **287** (2009) 925 [0804.1660].

[100] C. Bogner and F. Brown, *Feynman integrals and iterated integrals on moduli spaces of curves of genus zero*, *Commun. Num. Theor. Phys.* **09** (2015) 189 [1408.1862].

[101] F. C. Brown, *Single-valued multiple polylogarithms in one variable*, *C. R. Acad. Sci. Paris, Ser. I* **338** (2004) 527.

[102] F. C. S. Brown, *Single-valued hyperlogarithms and unipotent differential equations*, *IHES notes* (2004) .

[103] J. Drummond, *Generalised ladders and single-valued polylogarithms*, *JHEP* **1302** (2013) 092 [1207.3824].

[104] J. Drummond, C. Duhr, B. Eden, P. Heslop, J. Pennington and V. A. Smirnov, *Leading singularities and off-shell conformal integrals*, *JHEP* **08** (2013) 133 [1303.6909].

[105] O. Schnetz, *Hyperlog procedures*, *http://www.algeo.math.uni-erlangen.de/?2297* (2018) .

[106] O. Schnetz, *Numbers and Functions in Quantum Field Theory*, *Phys. Rev.* **D97** (2018) 085018 [1606.08598].

[107] L. J. Dixon, C. Duhr and J. Pennington, *Single-valued harmonic polylogarithms and the multi-Regge limit*, *JHEP* **1210** (2012) 074 [1207.0186].

[108] V. Del Duca, L. J. Dixon, C. Duhr and J. Pennington, *The BFKL equation, Mueller-Navelet jets and single-valued harmonic polylogarithms*, *JHEP* **02** (2014) 086 [1309.6647].

[109] Ø. Almelid, C. Duhr and E. Gardi, *Three-loop corrections to the soft anomalous dimension in multileg scattering*, *Phys. Rev. Lett.* **117** (2016) 172002 [1507.00047].

[110] Ø. Almelid, C. Duhr, E. Gardi, A. McLeod and C. D. White, *Bootstrapping the QCD soft anomalous dimension*, *JHEP* **09** (2017) 073 [1706.10162].

[111] T. Gehrmann, M. Jaquier, E. W. N. Glover and A. Koukoutsakis, *Two-Loop QCD Corrections to the Helicity Amplitudes for $H \to 3$ partons*, *JHEP* **02** (2012) 056 [1112.3554].

[112] C. Anastasiou, C. Duhr, F. Dulat, E. Furlan, F. Herzog and B. Mistlberger, *Soft expansion of double-real-virtual corrections to Higgs production at N^3LO*, *JHEP* **08** (2015) 051 [1505.04110].

[113] C. Duhr, T. Gehrmann and M. Jaquier, *Two-loop splitting amplitudes and the single-real contribution to inclusive Higgs production at N^3LO*, *JHEP* **02** (2015) 077 [1411.3587].

[114] S. Buehler and A. Lazopoulos, *Scale dependence and collinear subtraction terms for Higgs production in gluon fusion at N3LO*, *JHEP* **10** (2013) 096 [1306.2223].

[115] N. Deutschmann, C. Duhr, F. Maltoni and E. Vryonidou, *Gluon-fusion Higgs production in the Standard Model Effective Field Theory*, *JHEP* **12** (2017) 063 [1708.00460].

[116] N. Deutschmann, F. Maltoni, M. Wiesemann and M. Zaro, *Top-Yukawa contributions to bbH production at the LHC*, *Submitted to: JHEP* (2018) [1808.01660].

[117] V. Del Duca, C. Duhr, G. Somogyi, F. Tramontano and Z. Trócsányi, *Higgs boson decay into b-quarks at NNLO accuracy*, *JHEP* **04** (2015) 036 [1501.07226].

[118] V. Del Duca, C. Duhr, A. Kardos, G. Somogyi and Z. Trócsányi, *Three-Jet Production in Electron-Positron Collisions at Next-to-Next-to-Leading Order Accuracy*, *Phys. Rev. Lett.* **117** (2016) 152004 [1603.08927].

[119] V. Del Duca, C. Duhr, A. Kardos, G. Somogyi, Z. Szőr, Z. Trócsányi et al., *Jet production in the CoLoRFulNNLO method: event shapes in electron-positron collisions*, *Phys. Rev.* **D94** (2016) 074019 [1606.03453].

[120] P. Boyle, L. Del Debbio and A. Khamseh, *Massive momentum-subtraction scheme*, *Phys. Rev.* **D95** (2017) 054505 [1611.06908].

[121] S. Abreu, F. Febres Cordero, H. Ita, M. Jaquier, B. Page and M. Zeng, *Two-Loop Four-Gluon Amplitudes from Numerical Unitarity*, *Phys. Rev. Lett.* **119** (2017) 142001 [1703.05273].

[122] S. Abreu, F. Febres Cordero, H. Ita, B. Page and M. Zeng, *Planar Two-Loop Five-Gluon Amplitudes from Numerical Unitarity*, *Phys. Rev.* **D97** (2018) 116014 [1712.03946].

[123] S. Abreu, F. Febres Cordero, H. Ita, B. Page and V. Sotnikov, *Planar Two-Loop Five-Parton Amplitudes from Numerical Unitarity*, *JHEP* **11** (2018) 116 [1809.09067].

[124] S. Abreu, J. Dormans, F. Febres Cordero, H. Ita and B. Page, *Analytic Form of Planar Two-Loop Five-Gluon Scattering Amplitudes in QCD*, Phys. Rev. Lett. **122** (2019) 082002 [1812.04586].

[125] A. Brandhuber, M. Kostacinska, B. Penante and G. Travaglini, *Tr(F^3) supersymmetric form factors and maximal transcendentality Part I: $\mathcal{N} = 4$ super Yang-Mills*, JHEP **12** (2018) 076 [1804.05703].

[126] A. Brandhuber, M. Kostacinska, B. Penante and G. Travaglini, *Tr(F^3) supersymmetric form factors and maximal transcendentality Part II: $0 < \mathcal{N} < 4$ super Yang-Mills*, JHEP **12** (2018) 077 [1804.05828].

[127] S. Abreu, R. Britto, C. Duhr and E. Gardi, *From multiple unitarity cuts to the coproduct of Feynman integrals*, JHEP **1410** (2014) 125 [1401.3546].

[128] S. Abreu, R. Britto and H. Grönqvist, *Cuts and coproducts of massive triangle diagrams*, JHEP **07** (2015) 111 [1504.00206].

[129] J. Broedel, C. Duhr, F. Dulat and L. Tancredi, *Elliptic polylogarithms and iterated integrals on elliptic curves II: an application to the sunrise integral*, Phys. Rev. **D97** (2018) 116009 [1712.07095].

[130] J. Broedel, C. Duhr, F. Dulat, B. Penante and L. Tancredi, *Elliptic symbol calculus: from elliptic polylogarithms to iterated integrals of Eisenstein series*, JHEP **08** (2018) 014 [1803.10256].

[131] J. Broedel, C. Duhr, F. Dulat, B. Penante and L. Tancredi, *Elliptic Feynman integrals and pure functions*, JHEP **01** (2019) 023 [1809.10698].

[132] J. Broedel, C. Duhr, F. Dulat, B. Penante and L. Tancredi, *Elliptic polylogarithms and Feynman parameter integrals*, 1902.09971.