LiSK - A C++ Library for Evaluating Classical Polylogarithms and Li$_{22}$

Sebastian Kirchner

Institut für Theoretische Teilchenphysik und Kosmologie
RWTH Aachen University
Aachen, Germany

Abstract

I present a lightweight C++ library for the evaluation of classical polylogarithms $Li_n$ and the special function $Li_{22}$ for arbitrary complex arguments. The evaluation is possible in arbitrary precision arithmetic and features also an explicit double precision implementation for a much faster numerical evaluation. The implementation is based on Ref [1].

Keywords: Classical polylogarithms, $Li_{22}$, C++

1. Introduction

Classical polylogarithms and their multi-variable generalisations to generalised polylogarithms (GPLs) play an important role in modern quantum field theory. These functions and specialisations thereof have been studied in early works by Poincaré and Kummer, much later implicitly by Chen in his works on iterated integrals [2] and by Goncharov [3, 4, 5]. The importance lies in the fact that a large class of dimensionally regulated loop Feynman integrals in $d = 4 - 2\epsilon$ dimensions have been found to be expressible in terms of these GPLs (Please refer to Refs [6, 7, 8] and references therein for more detailed discussions). It is known that all one-loop integrals up to $\epsilon^0$ can be written in terms of the logarithm and the dilogarithm $Li_2$ [4]. The situation at higher orders in the $\epsilon$ expansion or, similar, at higher orders in perturbation theory, is already more involved. However, it was in Ref [1] recently proven that all GPLs up to weight four can be expressed in terms of classical polylogarithms and the special function $Li_{22}$. Hence the evaluation of Feynman integrals, fulfilling the

---

Email address: kirchner@physik.rwth-aachen.de (Sebastian Kirchner)

URL: https://bitbucket.org/SebastianKirchner/lisk (Sebastian Kirchner)
upper restrictions, and GPLs, in general, reduce then to two fundamental steps.
The first is expressing all GPLs up to weight four by \(L_i\) with \(n \leq 4\) and \(L_{i2}\). Although this step is far from trivial because divergent intermediate expressions may occur which have to be suitably regulated, a handful of methods have been proposed which, in principle, allow for a complete automatisation of this step. See e.g. Refs [1, 5, 6, 10, 11, 12, 13]. The second step is the fast and numerically stable evaluation of the resulting \(L_i\) and \(L_{i2}\). I want to address the latter in this work.

Efficient algorithms for the numerical evaluation of classical polylogarithms and specific subclasses of GPLs have been developed and studied in the literature [1, 14, 15, 16, 17, 18]. The evaluation of GPLs, and \(L_{i2}\) for that matter, with arbitrary complex arguments, however, is only publicly available in GiNaC [19, 20]. That specific implementation allows to evaluate GPLs without restrictions on the weight in arbitrary precision via the CLN library. Using the arbitrary precision implementation without exceptions obviously yields sincere performance penalties if the user is solely interested in standard floating point formats as e.g. double precision. The authors of Ref. [1] provide a new algorithm and also some C++ routines for the evaluation of the classical polylogarithms and the special function \(L_{i2}\). However, their implementation is limited to weights \(n \leq 6\) of the \(L_i\), and is exclusively given in double precision. Experience shows that higher precision results are often required in the computation of multi-loop Feynman integrals, such as in Ref [21, 22].

I believe that a combination of both worlds is desirable. The template features of the modern C++ language allows to easily incorporate multiple supported types without duplicating the entire code. The current version of LiSK supports a very fast double precision implementation as well as an arbitrary precision implementation utilising the CLN library. LiSK implements the computation of \(L_i\) and \(L_{i2}\) for arbitrary complex arguments using the algorithms from Ref [1]; but extends the computation of \(L_i\) to arbitrary weights.

LiSK has been tested under SUSE Linux and is known to work with the compiler versions g++-4.8.3, clang++-3.5 and icc 14.0.2 using the GNU C++ standard library. LiSK can be downloaded from

https://bitbucket.org/SebastianKirchner/lisk

This work is structured as follows. I will explain the user interface and show an example how to use LiSK in Sec. 2. In Sec. 3 I will validate the results obtained by LiSK and conclude in Sec. 4.

2. Usage

LiSK is a complete header-only library, i.e. the library does not have to be built separately but only the header file must be included in the user’s code via the usual

```
#include "lisk.hpp"
```
Please note: The implementation has been put into the lisk.cpp file due to readability. This source file must be present in the same directory as the header file.

LiSK is completely encapsulated in the namespace LiSK and relies on several C++-11 features. A central part is the CLN library by Bruno Haible and has to be linked to all programs using LiSK; also if using only double precision. A simple example of how to use LiSK is given in example/example.cpp, which can be compiled via

```
g++ -O2 -std=c++11 example.cpp -o example -I/cln/include/path
-I../LiSK -L/cln/library/path -lcln
```

Alternatively, if cmake is available, simply by running

```
cmake .
```

in the example directory. If CLN is not found automatically the `CLN_INCLUDE_DIR` and `CLN_LIB` path can easily be set via

```
cmake .
```

I will illustrate the usage of LiSK on this simple example. In the following T denotes one of the two currently supported types `std::complex<double>` and `cln::clN`. The first action should be to create a LiSK object of type T and precision p via

```
LiSK::LiSK<T> lisk(n, p);
```

During object creation LiSK is initialised and all constants required for the computation of the \(Li_n(x)\) and \(Li_{22}(x, y)\) are pre-computed. This reflects the main idea of LiSK; prepare and save all needed constants during its initialisation phase and use them during the actual computation. The constructor \(\text{LiSK}(n = 4, \text{prec} = 34)\) features two optional arguments.

The first argument \(n\) defines the weight of the \(Li_n(x)\) up to which the constants are computed during LiSKs initialisation phase. It is not mandatory but advised to set \(n\) to a value which resembles the highest expected weights. If higher weights are encountered during the computation the constants will be adapted dynamically. This, obviously, leads to longer evaluation times for the polylogarithms for which the higher weights have been encountered. This situation should be avoided as much as possible.

The second argument of the constructor sets the desired precision if \(T = \text{cln::clN}\) is chosen. This argument is superfluous in the double precision case. E.g. set \(p = 34\) to obtain results with 34 digit precision. Internally all floating point values are set to this precision. The user has to ensure that the supplied input values match the requested precision.

\(^1\)CLN can be downloaded from [http://www.ginac.de/CLN/](http://www.ginac.de/CLN/)
Calling the public wrapper functions for the computation of \( \text{Li}_m(x) \) for positive integer weights \( m \) and \( \text{Li}_{22}(x,y) \) at given points \( x \) and \( y \) is given by

\[
\text{lisk} \cdot \text{Li}(m,x);
\text{lisk} \cdot \text{Li22}(x,y);
\]

where \( m \) should be smaller than \( n \). One might also call

\[
\text{LiSK} :: \text{LiSK}<\mathbb{T}>(n,p) \cdot \text{Li}(m,x);
\text{LiSK} :: \text{LiSK}<\mathbb{T}>(n,p) \cdot \text{Li22}(x,y);
\]

but this is strongly not recommended due to the above mentioned reasons. However, special wrapper functions exist for the classical polylogarithms with weights \( n \leq 4 \)

\[
\text{lisk} \cdot \text{Li1}(x);
\text{lisk} \cdot \text{Li2}(x);
\text{lisk} \cdot \text{Li3}(x);
\text{lisk} \cdot \text{Li4}(x);
\]

In case some error is encountered \text{LiSK} will throw a \text{std::runtime_error}. Hence, it is advised to put all calls to \text{LiSK} into a \text{try}-block like

\[
\text{try}
\{
    \text{ /* some code */}
\}
\text{catch}(\text{std::runtime\_error } \& e )
\{
    \text{std::cout} \ll e.\text{what()} \ll \text{std::endl};
\}
\]

Last but not least it must be ensured that all expressions, initial and intermediate, are well defined. To this end a small positive imaginary part is added to the initial arguments \( x \) and \( y \) of \( \text{Li}_n(x) \) and \( \text{Li}_{22}(x,y) \), i.e. \( x \rightarrow x - i\epsilon \). The sign of the imaginary part is chosen according to the widely used convention for mathematical software\(^2\) which agrees for example with the convention used in \text{Mathematica} and \text{GiNaC}. The value of \( \epsilon \) is set to \( 10^{-(p-\text{offset})} \) during initialisation. Hereby defines \( p \) the requested precision in the constructor (\( p = 17 \) in \text{double} precision mode). The default value of \( \text{offset} \) is 2. The user can change this value at the top of the \text{LiSK} header\(^3\).

3. Checks

In this section I discuss some checks of the \text{LiSK} library by its comparison against \text{GiNaC}.

\(^2\)\text{[\ldots] implementations shall map a cut so the function is continuous as the cut is approached coming around the finite endpoint of the cut in a counter clockwise direction} \cite{22}

\(^3\)A handful of options to change the behaviour of \text{LiSK} can be set in the header. However, these are of rather technical (and experimental) nature and will not be discussed here. The user is advised to proceed with caution.
Various tests in different parameter regions have been performed. Similar to Ref [1] I have checked the level of precision by evaluating $\text{Li}_n$ (for multiple values of $n$) and $\text{Li}_{22}$ at $10^4$ random parameter points, where the absolute value of each point is exponentially distributed between $10^{-10}$ and $10^{10}$. Multiple tests for real input parameters and parameters in vicinity of zero and one have been performed. In case of the double precision implementation all points had a relative deviation $(2|a - b|/|a + b|)$ smaller than $10^{-13}$. The arbitrary precision implementation is in perfect agreement with GiNaC, provided that a consistent $\epsilon$ prescription is given in both programs.

From the performance point of view LiSK evaluates the $\text{Li}_n$ and $\text{Li}_{22}$ in comparable time and orders of magnitudes faster than the current GiNaC implementation; depending on the parameter values. The exact timing depends on the used architecture, compiler, program setup, etc. For brevity of this work I encourage the interested user to try and compare both implementations for his specific problem. The double precision implementation is, as expected, much faster than using the arbitrary precision arithmetic in CLN and might lead to a sizeable speed-up in computations where double precision is sufficient.

4. Conclusions

In this work I have presented the first version of LiSK, an independent implementation for the numerical evaluation of $\text{Li}_n$ and $\text{Li}_{22}$ at arbitrary complex arguments. LiSK evaluates $\text{Li}_n$ and $\text{Li}_{22}$ in arbitrary precision but also features an explicit double precision implementation. The latter provides a much faster evaluation of the polylogarithms, which might be desirable for certain users. The complete template C++ implementation provides a basis to easily extend the supported types, e.g. to the GNU Quad-Precision Math Library.

LiSK utilises many of the algorithms from Ref [1]. However not all of these algorithms are currently in use. Especially the numerical evaluation of $\text{Li}_{22}$ with arguments in vicinity of one leaves room for performance improvements in upcoming versions.

5. Acknowledgements

This work was supported by the Deutsche Forschungsgemeinschaft through Graduiertenkolleg GRK 1675.

References

[1] H. Frellesvig, D. Tommasini, C. Wever, On the reduction of generalized polylogarithms to $\text{Li}_n$ and $\text{Li}_{22}$ and on the evaluation thereof, JHEP 03 (2016) 189. arXiv:1601.02849 doi:10.1007/JHEP03(2016)189

[2] K.-T. Chen, Iterated path integrals, Bull. Am. Math. Soc. 83 (1977) 831–879. doi:10.1090/S0002-9904-1977-14320-6
[3] A. B. Goncharov, Multiple polylogarithms, cyclotomy and modular complexes, Math. Res. Lett. 5 (1998) 497–516. arXiv:1105.2076 doi:10.4310/MRL.1998.v5.n4.a7

[4] A. B. Goncharov, Multiple polylogarithms and mixed Tate motives arXiv:math/0103059

[5] A. B. Goncharov, M. Spradlin, C. Vergu, A. Volovich, Classical Polylogarithms for Amplitudes and Wilson Loops, Phys. Rev. Lett. 105 (2010) 151605. arXiv:1006.5703 doi:10.1103/PhysRevLett.105.151605

[6] C. Duhr, Mathematical aspects of scattering amplitudes, in: Theoretical Advanced Study Institute in Elementary Particle Physics: Journeys Through the Precision Frontier: Amplitudes for Colliders (TASI 2014) Boulder, Colorado, June 2-27, 2014, 2014. arXiv:1411.7538

[7] J. M. Henn, Multiloop integrals made simple: applications to QCD processes, in: Proceedings, 49th Rencontres de Moriond on QCD and High Energy Interactions, 2014, pp. 289–292. arXiv:1405.3683

[8] J. M. Henn, Lectures on differential equations for Feynman integrals, J. Phys. A48 (2015) 153001. arXiv:1412.2298 doi:10.1088/1751-8113/48/15/153001

[9] R. K. Ellis, G. Zanderighi, Scalar one-loop integrals for QCD, JHEP 02 (2008) 002. arXiv:0712.1851 doi:10.1088/1126-6708/2008/02/002

[10] A. B. Goncharov, A simple construction of Grassmannian polylogarithms arXiv:0908.2238.

[11] C. Duhr, Hopf algebras, coproducts and symbols: an application to Higgs boson amplitudes, JHEP 08 (2012) 043. arXiv:1203.0454 doi:10.1007/JHEP08(2012)043

[12] C. Bogner, Generalizations of polylogarithms for Feynman integrals, in: 17th International workshop on Advanced Computing and Analysis Techniques in physics research (ACAT 2016) Valparaiso, Chile, January 18-22, 2016. arXiv:1603.00420

[13] C. Bogner, F. Brown, Symbolic integration and multiple polylogarithms PoSLL2012,053(2012). arXiv:1209.6524

[14] K. S. Kolbig, J. A. Mignoco, E. Remiddi, ON NIELSEN’S GENERALIZED POLYLOGARITHMS AND THEIR NUMERICAL CALCULATION.

[15] G. ’t Hooft, M. J. G. Veltman, Scalar One Loop Integrals, Nucl. Phys. B153 (1979) 365–401. doi:10.1016/0550-3213(79)90605-9

[16] J. M. Borwein, D. M. Bradley, D. J. Broadhurst, P. Lisonek, Special values of multiple polylogarithms, Trans. Am. Math. Soc. 353 (2001) 907–941. arXiv:math/9910045 doi:10.1090/S0002-9947-00-02616-7
[17] T. Gehrmann, E. Remiddi, Numerical evaluation of harmonic polylogarithms, Comput. Phys. Commun. 141 (2001) 296–312. arXiv:hep-ph/0107173 doi:10.1016/S0010-4655(01)00411-8

[18] T. Gehrmann, E. Remiddi, Numerical evaluation of two-dimensional harmonic polylogarithms, Comput. Phys. Commun. 144 (2002) 200–223. arXiv:hep-ph/0111255 doi:10.1016/S0010-4655(02)00139-X

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

[20] J. Vollinga, S. Weinzierl, Numerical evaluation of multiple polylogarithms, Comput. Phys. Commun. 167 (2005) 177. arXiv:hep-ph/0410259 doi:10.1016/j.cpc.2004.12.009

[21] T. Gehrmann, A. von Manteuffel, L. Tancredi, The two-loop helicity amplitudes for $q\bar{q} \rightarrow V_1V_2 \rightarrow 4$ leptons, JHEP 09 (2015) 128. arXiv:1503.04812 doi:10.1007/JHEP09(2015)128

[22] A. von Manteuffel, L. Tancredi, The two-loop helicity amplitudes for $gg \rightarrow V_1V_2 \rightarrow 4$ leptons, JHEP 06 (2015) 197. arXiv:1503.08835 doi:10.1007/JHEP06(2015)197

[23] B. S. Institution, The C standard, John Wiley, 2003. URL https://books.google.de/books?id=e4whAQAIAAAJ