handyG

Release v0.1.5

L. Naterop, A. Signer, Y. Ulrich

Oct 09, 2023

CONTENTS:

1 Getting started **3**
- 1.1 Obtaining the code . . . 3
- 1.2 Installation using meson and ninja (recommended) . . . 3
- 1.3 Installing using make . . . 4
- 1.4 Usage in Fortran . . . 5
- 1.5 Usage in Mathematica . . . 6

2 Notation **7**
- 2.1 Multiple polylogarithms . . . 7
- 2.2 Convergence properties . . . 8
- 2.3 Shuffle algebra and trailing zeros . . . 8

3 The algorithm **9**
- 3.1 Removal of trailing zeros . . . 9
- 3.2 Making GPLs convergent . . . 9
- 3.3 Evaluation of pending integrals . . . 11
- 3.4 Increase rate of convergence . . . 13
- 3.5 An example reduction . . . 13

4 Fortran reference guide **15**
- 4.1 User-facing functions . . . 15
- 4.2 Internal functions . . . 16
- 4.3 Cache system . . . 26

5 Known Issues **27**
- 5.1 Segmentation fault for arguments on the complex unit circle . . . 27
- 5.2 GPLs with arguments close to one are not precise . . . 28
- 5.3 Parallel builds are not supported . . . 28

6 Bibliography **29**

Bibliography **31**

Index **33**

Generalised polylogarithms naturally appear in higher-order calculations of quantum field theories. We present handyG [6], a Fortran 90 library for the evaluation of such functions, by implementing the algorithm proposed by Vollinga and Weinzierl. This allows fast numerical evaluation of generalised polylogarithms with currently relevant weights, suitable for Monte Carlo integration.

CHAPTER

ONE

GETTING STARTED

We provide a pre-compiled Mathematica interface for most Linux systems, both for double precision and the quad precision. Once downloaded, just make the file executable through

```
# for double precision
$ chmod +x handyG-double
# for quad precision
$ chmod +x handyG-quad
```

and load it into Mathematica

```
(* for double precision *)
Install["handyG-double"]
(* for quad precision *)
Install["handyG-quad"]
```

1.1 Obtaining the code

The code can be downloaded from this page in compressed form or cloned using the *git* command

```
$ git clone https://gitlab.com/mule-tools/handyG.git
```

This will download `handyG` into a subfolder called `handyg`. Within this folder

```
$ git pull
```

can be used to update `handyG`.

1.2 Installation using meson and ninja (recommended)

`handyG` can most easily be build with meson and ninja. You can install these through your system's package manager or pip if you have not already

```
$ pip install meson ninja
```

Once you have these tools, you can run

```
$ meson setup build        # Configures handyG
$ ninja -C build           # Compiles the library
$ ninja -C build test      # Performs checks (optional)
$ ninja -C build install   # Installs library into prefix (optional)
```

This will compile handyG in the subfolder build (you can choose any name).

During the configuration step (meson setup) you can provide a number of options

- install handyG to a non-standard path (recommended)

  ```
  $ meson setup build --prefix /path/to/installation/folder
  ```

- perform dynamic linking (produces libhandyg.so rather than libhandyg.a)

  ```
  $ meson setup build --default-library shared
  ```

- Use quadruple precision (128 bits) rather than double precision (64 bits)

  ```
  $ meson setup build -Dreal=128
  ```

- Compile Mathematica interface (requires mathematica to be installed or mocked)

  ```
  $ meson setup build -Dmcc=true
  ```

- Compile GiNaC interface (testing only, requires GiNaC to be installed)

  ```
  $ meson setup build -Dginac=true
  ```

- Build handyG with debug symbols (testing and debugging only)

  ```
  $ meson setup build --buildtype=debug
  ```

You can of course mix and match these options. For further details, see the meson manual

1.3 Installing using make

The code follows the conventional installation scheme

```
./configure  # Look for compilers and make a guess at
             # necessary flags
make all     # Compiles the library
make check   # Performs a variety of checks (optional)
make install # Installs library into prefix (optional)
```

handyG has a Mathematica interface (activate with --with-mcc) and a GiNaC interface (activate with --with-ginac) that can be activated by supplying the necessary flags to ./configure. The latter is only used for testing purposes and is not actually required for running. Another important flag is --quad which enables quadruple precision in Fortran. Note that this will slow down handyG, so that it should only be used if double-precision is indeed not enough.

The compilation process creates the following results

- libhandyg.a the handyG library
- handyg.mod the module files for Fortran 90

- `geval` a binary file for quick-and-dirty evaluation
- `handyG` the Mathematica interface

1.4 Usage in Fortran

`handyG` is written with Fortran in mind. We provide a module `handyg.mod` containing the following objects

- **`prec`**
  the working precision as a Fortran `kind`. This is read-only, the code needs to be reconfigured for a change to take effect. Note that this does not necessarily increase the result's precision without also changing the next options.
- `inum`
  a datatype to handle $^{\pm}$-prescription (see Section 3.4).
- `clearcache`
  `handyG` caches a certain number of classical polylogarithms (see Section 3.5). This resets the cache (in a Monte Carlo this should be called at every phase space point).
- `G`
  the main interface for generalised polylogarithms.

The following code calculates five GPLs (see paper for details)

```
PROGRAM gtest
  use handyG
  complex(kind=prec) :: res(5), x, weights(4)
  call clearcache

  x = 0.3 ! the parameter

  ! flat form with integers
  res(1) = G((/ 1, 2, 1 /))

  ! very flat form for real numbers using F2003 arrays
  res(2) = G([ 1., 0., 0.5, real(x)])
  ! this is equivalent to the flat expression
  res(2) = G([ 1., 0., 0.5 ], real(x))
  ! or in condesed form
  res(2) = G((/1, 2/), (/ 1., 0.5 /), real(x))

  ! flat form with complex arguments
  weights = [(1.,0.), (0.,0.), (0.5,0.), (!.,1.) ]
  res(3) = G(weights, x)

  ! flat form with explicit i0-prescription
  res(4) = G([inum(1.,+1),inum(0,+1),inum(5,+1)], &
                    inum(1/x,di0))
  res(5) = G([inum(1.,-1),inum(0,+1),inum(5,+1)],&
                    inum(1/x,di0))
  ! this is equivalent to
  res(5) = G((/1,2/),[inum(1.,-1),inum(5,+1)], &
```

(continues on next page)

(continued from previous page)

```
                    inum(1/x,+1))

  do i =1,5
    write(*,900) i, real(res(i)), aimag(res(i))
  enddo
900 FORMAT("res(",I1,") = ",F9.6,"+",F9.6,"i")
END PROGRAM gtest
```

The easiest way to compile code is with `pkg-config`. Assuming `handyG` has been installed with `make install`, the example program `example.f90` can be compiled as (assuming you are using GFortran)

```
$ gfortran -o example example.f90 \
       `pkg-config --cflags --libs handyg`
$ ./example
res(1) = -0.822467+ 0.000000i
res(2) =  0.128388+ 0.000000i
res(3) = -0.003748+ 0.003980i
res(4) = -0.961279+-0.662888i
res(5) = -0.961279+ 0.662888i
```

If *pkg-config* is not avaible and/or for non-standard installations it might be necessary to specify the search paths

```
$ gfortran -o example example.f90 \
>     -I/absolute/path/to/handyg -fdefault-real-8 \
>     -L/absolute/path/to/handyg -lhandyg
```

1.5 Usage in Mathematica

We have interfaced our code to Mathematica using Wolfram's MathLink interface. Below we show how to calculate the functions above in Mathematica, again, assuming that the code was installed with `make install`

```
Install["handyg"];
x=0.3;
res[1] = G[1,2,1]
res[2] = G[1,0,1/2,x]
res[3] = G[1,0,1/2,1+I,x]
res[4] = G[SubPlus[1],5,1/x]
res[5] = G[SubMinus[1],5,1/x]
```

Using `SubPlus` and `SubMinus` the side of the branch cut can be specified. In Mathematica, this can be entered using `ctrl _` followed by `+` or `-`. When using `handyG` in Mathematica, keep in mind that it uses Fortran which means that computations are performed with fixed precision.

CHAPTER

TWO

NOTATION

GPLs are complex-valued functions that depend on m complex parameters $z_1, ..., z_m$ as well as an argument y. We can define a GPL as a nested integral with $z_m \neq 0$

$$G(z_1, ..., z_m\,; y) \equiv \int_0^y \frac{t_1}{t_1 - z_1} \int_0^{t_1} \frac{t_2}{t_2 - z_2} \cdots \int_0^{t_{m-1}} \frac{t_m}{t_m - z_m}\,. \tag{2.1}$$

Alternatively, they can also be defined in recursive form as

$$G(z_1, ..., z_m\,; y) = \int_0^y \frac{t_1}{t_1 - z_1} G(z_2, ..., z_m\,; t_1)\,,$$

where the base case of $m = 1$ is just a logarithm

$$G(z\,; y) = \log\left(1 - \frac{y}{z}\right).$$

To also cover the case of $z_m = 0$ we define

$$G(\underbrace{0, ..., 0}_{m}\,; y) \equiv G(0_m\,; y) = \frac{(\log y)^m}{m!}\,, \tag{2.2}$$

where we denote a string of m zeros as 0_m.

We call $G(z_1, ..., z_m; y)$ *flat* since all parameters are explicit. However, this notation can be cumbersome if many of the z_i are zero. In this case we introduce the *condensed* notation which uses partial weights m_i in order to keep track of the number of zeros in front of the parameter z_i

$$G_{m_1,...,m_k}\big(z_1, ..., z_k\,; y\big) \equiv G\big(0_{m_1-1}, z_1, ..., z_{k-1}, 0_{m_k-1}, z_k\,; y\big)\,. \tag{2.3}$$

Both notations will be used interchangeably. We say that this GPL is of depth k as it has k non-zero parameters (not counting y). Its total weight is $m = \sum m_i$.

2.1 Multiple polylogarithms

Multiple polylogarithms (MPLs) are a related class of functions that also generalise logarithms. They are defined as an infinite nested series

$${}_{m_1,...,m_k}(x_1, ..., x_k) \equiv \sum_{i_1 > \cdots > i_k}^{\infty} \frac{x_1^{i_1}}{i_1^{m_1}} \cdots \frac{x_k^{i_k}}{i_k^{m_k}}\,, \tag{2.4}$$

where $m_1, ..., m_k$ are integer weights. If there is only one argument present, they reduce to classical polylogarithms ${}_m(x)$.

MPLs are closely related to GPLs through

$$_{m_1,...,m_k}(x_1,...,x_k) = (-1)^k G_{m_1,...,m_k}\left(\frac{1}{x_1}, \frac{1}{x_1 x_2}, ..., \frac{1}{x_1 \cdots x_k}\,;1\right).$$

This can be inverted by performing an iterated substitution

$$u_1 = \frac{1}{x_1}\,, \quad u_2 = \frac{1}{x_1 x_2} = \frac{u_1}{x_1}\,, \quad ... \qquad u_k = \frac{1}{x_1 ... x_k} = \frac{u_{k-1}}{x_k}\,,$$

allowing us to write the GPLs in terms of MPLs

$$G_{m_1,...,m_k}(u_1,...,u_k\,;1) = (-1)^k_{m_1,...,m_k}\left(\frac{1}{u_1}, \frac{u_1}{u_2}, ..., \frac{u_{k-1}}{u_k}\right). \tag{2.5}$$

In (2.5), the left-hand side is an integral representation whereas the right-hand side is a series representation.

GPLs with arbitrary parameters satisfy the scaling relation

$$G(z_1,...,z_m\,;y) = G(\kappa z_1,...,\kappa z_m\,;\kappa y) \tag{2.6}$$

for any complex number $\kappa \neq 0$. (2.5) assumes the argument of G is equal to one. Using the scaling relation we can normalise $G(z_1,...,z_m;y)$ with $\kappa = 1/y$ to guarantee that the argument is indeed one.

For the numerical evaluation the main idea will be to compute G-functions by reducing them to their corresponding series representation (2.5).

2.2 Convergence properties

If we want to use an infinite series for numerical evaluation of GPLs, the series needs to be convergent. It can be shown [7] that an MPL $_{m_1,...,m_k}(x_1,...,x_k)$ is convergent if the conditions

$$|x_1 \cdots x_k| < 1 \qquad \text{and} \qquad (m_1, x_1) \neq (1,1)$$

are satisfied. Using the relation (2.5), this translates to a sufficient convergence criterion for the integral representation. We find that if

$$|y| < |z_i| \quad \forall i = 1,...,k \quad \text{and} \quad (m_1, y/z_1) \neq (1,1)\,, \tag{2.7}$$

$G_{m_1,...,m_k}(z_1,...,z_k\,;y)$ is convergent.

In Section *The algorithm* we will review the algorithm developed by [7] to transform any GPL into this form.

2.3 Shuffle algebra and trailing zeros

If the last parameter z_k of a GPL $G_{m_1,...,m_k}(z_1,...,z_k\,;y)$ vanishes, the convergence criterion (2.7) is not fulfilled. Hence, any algorithm that intents to exploit (2.4) for numerical evaluation needs to remove trailing zeros.

We can exploit the fact that GPLs satisfy two Hopf algebras: a shuffle algebra and a stuffle algebra [3, 5, 7]. Here, we will only be needing the former. It allows us to write the product of two GPLs with parameters $\vec{a}$ and $\vec{b}$ as

$$G(\vec{a}\,;y) \cdot G(\vec{b}\,;y) = \sum_{\vec{c}=\vec{a}\;\vec{b}} G(\vec{c}\,;y)\,. \tag{2.8}$$

The sum in the right-hand side of (2.8) runs over all elements of the shuffle product of the list $\vec{a}$ with $\vec{b}$. This shuffle product gives the set of all permutations of the elements in $\vec{a}$ and $\vec{b}$ that preserve the respective orderings of $\vec{a}$ and $\vec{b}$. For practical implementations, a recursive algorithm exists [4].

CHAPTER

THREE

THE ALGORITHM

The central idea to numerically evaluate GPLs is to first map their parameters to the domain where the corresponding series representation is convergent (2.7) and to then use the series expansion up to some finite order. Thus, we will first look at how to remove trailing zeros in Section *Removal of trailing zeros*, and then how to make a GPL without trailing zeros convergent in Section *Making GPLs convergent* as presented in [7]. In Section *Increase rate of convergence*, we comment on accelerating the convergence of already convergent GPLs. Finally, in Section *An example reduction* we apply the algorithm to an explicit example.

3.1 Removal of trailing zeros

Consider a GPL of weight m with $m-j$ trailing zeros

$$G(z_1, ..., z_j, 0_{m-j}\,; y)\,.$$

We now shuffle $\vec{a} = (z_1, ..., z_j, 0_{m-j-1})$ with $\vec{b} = (0)$. This results in $m-j$ times the original GPL as well as terms with less trailing zeros

$$\begin{aligned} G(0\,; y) \cdot G(z_1, ..., z_j, 0_{m-j-1}\,; y) = (m-j) G(z_1, ..., z_j, 0_{m-j}\,; y) \\ + \sum_{\vec{s}} G(s_1, ..., s_j, z_j, 0_{m-j-1}\,; y)\,, \end{aligned} \tag{3.1}$$

where the sum runs over all shuffle $\vec{s} = (z_1, ..., z_{j-1})\ (0)$. We now solve (3.1) for $G(z_1, ..., z_j, 0_{m-j}; y)$ and obtain an expression with fewer trailing zeros. By applying this strategy recursively, we can remove all trailing zeros.

3.2 Making GPLs convergent

3.2.1 Reduction to pending integrals

Consider a GPL of the form

$$G(a_1, ..., a_{i-1}, s_r, a_{i+1}, ..., a_m\,; y) \tag{3.2}$$

where $s_r (= a_i)$ has the smallest absolute value among all the non-zero parameters in G. If $|s_r| < |y|$, (3.2) has no convergent series expansion. In order to remove the smallest weight s_r, we apply the fundamental theorem of calculus to generate terms where s_r is either integrated over or not present anymore

$$\begin{aligned} G(a_1, ..., a_{i-1}, s_r, a_{i+1}, ..., a_m\,; y) = G(a_1, ..., a_{i-1}, 0, a_{i+1}, ..., a_m\,; y) \\ + \int_0^{s_r} s_{r+1} \frac{\partial}{\partial s_{r+1}} G(a_1, ..., a_{i-1}, s_{r+1}, a_{i+1}, ..., a_m\,; y)\,. \end{aligned}$$

For the second term we use partial fraction decomposition and integration by parts. Then we obtain different results depending on where s_r is in the parameter list:

- *If s_r appears first* in the list (i.e. $i = 1$ and $s_r = a_1$) we find

$$\begin{aligned} G(s_r, a_{i+1}, ..., a_m\,; y) &= G(0, a_{i+1}, ..., a_m\,; y) + \underbrace{\int_0^{s_r} \frac{s_{r+1}}{s_{r+1} - y}}_{G(y\,;s_r)} G(a_{i+1}, ..., a_m\,; y) \\ &+ \underbrace{\int_0^{s_r} \frac{s_{r+1}}{s_{r+1} - a_{i+1}} G(s_{r+1}, a_{i+2}, .., a_m\,; y)}_{\text{pending integral}} - \underbrace{\int_0^{s_r} \frac{s_{r+1}}{s_{r+1} - a_{i+1}}}_{G(a_2\,;s_r)} G(a_{i+1}, ..., a_m\,; y)\,. \end{aligned} \tag{3.3}$$

  In the first term on the right-hand side, s_r is absent. Therefore the resulting GPL is simpler. It might still be non-convergent, but we can use this method recursively on the resulting GPLs until we end up with convergent GPLs.

  In the second and fourth terms the integration variable s_{r+1} does not appear in the parameters of the GPL, so that the integral can be solved (we write the solution as a GPL instead of a logarithm to be able to continue recursively).

  The third term does have the integration variable s_{r+1} among the weights and therefore yields what we refer to as a pending integral. This object can be written as a linear combination of simpler GPLs as we will see in Section *Evaluation of pending integrals*.

  Note that all GPLs on the right-hand side have depth reduced by one.

- *If s_r appears in the middle* of the list, i.e. $1 < i < m$, we find

$$\begin{aligned} G(a_1, ..., a_{i-1}, s_r, a_{i+1}, ..., a_m\,; y) = \\ &+ G(a_1, ..., a_{i-1}, 0, a_{i+1}, ..., a_m\,; y) \\ &- \int_0^{s_r} \frac{s_{r+1}}{s_{r+1} - a_{i-1}} G(a_1, ..., a_{i-2}, s_{r+1}, a_{i+1}, ..., a_m\,; y) \\ &+ \underbrace{\int_0^{s_r} \frac{s_{r+1}}{s_{r+1} - a_{i-1}}}_{G(a_{i-1}\,;s_r)} G(a_1, ..., a_{i-1}, a_{i+1}, ..., a_m\,; y) \\ &+ \int_0^{s_r} \frac{s_{r+1}}{s_{r+1} - a_{i+1}} G(a_1, ..., a_{i-1}, s_{r+1}, a_{i+2}, ..., a_m\,; y) \\ &- \underbrace{\int_0^{s_r} \frac{s_{r+1}}{s_{r+1} - a_{i+1}}}_{G(a_{i+1}\,;s_r)} G(a_1, ..., a_{i-1}, a_{i+1}, ..., a_m\,; y)\,. \end{aligned} \tag{3.4}$$

  Again we obtain simpler GPLs (without s_r or lower depth) as well as pending integrals.

- *If s_r appears last* in the list, i.e. $i = m$, we use the shuffle algebra to remove s_r from the last place, just as we have done to remove trailing zeros.

We repeat these steps also for GPLs that are already under a pending integral.

3.3 Evaluation of pending integrals

The most general term created by the procedure of the last section is of the form

$$\Big(\vec{p}=(y',\vec{b}),i,\vec{g}=(\vec{a},y)\Big) \equiv \int_0^{y'} \frac{s_1}{s_1-b_1}\int_0^{s_1}\frac{s_2}{s_2-b_2}\cdots\int_0^{s_{r-1}}\frac{s_r}{s_r-b_r} G(a_1,...,a_{i-1},s_r,a_{i+1},...,a_m\,;y)\,. \tag{3.5}$$

Here we have adopted the convention that $i=0$ implies that the integration variable does not appear inside the GPL. For example

$$\Big(\vec{p}=(1,2,3),0,(4,5)\Big) = \int_0^1 \frac{s_1}{s_1-2}\int_0^{s_1}\frac{s_2}{s_2-3}G(4;5)$$
$$\Big(\vec{p}=(1,2,3),2,(4,5)\Big) = \int_0^1 \frac{s_1}{s_1-2}\int_0^{s_1}\frac{s_2}{s_2-3}G(4,s_2;5)\,.$$

As we use the algorithm, we need a way to collapse the pending integrals back down again. As an example, consider the case $i=1$

$$\begin{aligned}
&\Big(\vec{p}=(y',\vec{b}),1,\vec{g}=(\vec{a},y)\Big) = \int_0^{y'} \frac{s_1}{s_1-b_1}\cdots\int_0^{s_{r-1}}\frac{s_r}{s_r-b_r}G(s_r,a_{i+1},...,a_m\,;y) = \\
&\underbrace{\int_0^{y'} \frac{s_1}{s_1-b_1}\cdots\int_0^{s_{r-1}}\frac{s_r}{s_r-b_r}}_{(\vec{p},0,())} G(0,a_{i+1},...,a_m\,;y) \\
&+\underbrace{\int_0^{y'} \frac{s_1}{s_1-b_1}\cdots\int_0^{s_{r-1}}\frac{s_r}{s_r-b_r}\int_0^{s_r}\frac{s_{r+1}}{s_{r+1}-y}}_{\Big((\vec{p},y),0,()\Big)} G(a_{i+1},...,a_m\,;y) \\
&+\underbrace{\int_0^{y'} \frac{s_1}{s_1-b_1}\cdots\int_0^{s_{r-1}}\frac{s_r}{s_r-b_r}\int_0^{s_r}\frac{s_{r+1}}{s_{r+1}-a_{i+1}}G(s_{r+1},a_{i+2},...,a_m\,;y)}_{\Big((\vec{p},a_{i+1}),1,(a_{i+2},...,a_m;y)\Big)} \\
&-\underbrace{\int_0^{y'} \frac{s_1}{s_1-b_1}\cdots\int_0^{s_{r-1}}\frac{s_r}{s_r-b_r}\int_0^{s_r}\frac{s_{r+1}}{s_{r+1}-a_{i+1}}}_{\Big((\vec{p},a_{i+1}),0,()\Big)} G(a_{i+1},...,a_m\,;y) \\
&=\Big(\vec{p},0,()\Big)G(0,a_{i+1},...,a_m\,;y)+\Big((\vec{p},y),0,()\Big)G(a_{i+1},....,a_m\,;y) \\
&+\Big((\vec{p},a_{i+1}),1,(a_{i+2},...,a_m\,;y)\Big)-\Big((\vec{p},a_{i+1}),0,()\Big)G(a_{i+1},...,a_m\,;y)\,.
\end{aligned}$$

The other combinations follow similarly

$$\begin{aligned}
\Big(\vec{p},i,(\vec{a};y)\Big) = &+\Big(\vec{p},0,()\Big)\,G(a_1,...,a_{i-1},0,a_{i+1},...,a_m\,;y) \\
&-\Big((\vec{p},a_{i-1}),i-1,(a_{i+1},...,a_m;y)\Big) \\
&+\Big((\vec{p},a_{i-1}),0,()\Big)\,G(a_1,...,a_{i-1},a_{i+1},...,a_m\,;y) \\
&+\Big((\vec{p},a_{i+1}),i,(a_1,...,a_{i-1},a_{i+2},...,a_m;y)\Big) \\
&-\Big((\vec{p},a_{i+1}),1,()\Big)\,G(a_1,...,a_{i-1},a_{i+1},...,a_m\,;y)\,.
\end{aligned}$$

As we recursively apply the algorithm, we increase the number of pending integrals in front but decrease the depth of the G-functions by one unit in every recursion step. We do this until

1. the only GPLs remaining under pending integrals are of depth one, i.e. $G_m(s_r\ y)$,
2. s_r is the argument, i.e. $G(...\,;s_r)$, or
3. there are no GPLs under pending integrals.

We now discuss all these cases in turn:

1. For GPLs of depth one, i.e. $G_m(s_{r\pm};y)$, we will be working with explicit logarithms. Hence, we need to indicate the infinitesimal imaginary part. We have to distinguish two cases: $m=1$ and $m>1$. For $m=1$ we have

$$G_1(s_{r\pm};y) = G_1(y_{2\mp};s_r) - G(0;\ s_r) + \log(-y)\,.$$

   Note that we will most likely have pending integrals in front, thus each term gives again a simpler pending integral

$$\Big(\vec{p}=(y'_{\pm},\vec{b}),1,(y)\Big) = G(\vec{b},y_{\mp}\,;y') - G(\vec{b},0\,;y') + \log(-y_{\mp})G\big(\vec{b},y'\big)$$

   The first and second terms have been reduced to case 2. and the third term to case 3.

   For $m>1$, we note

$$G_m(s_{r\pm}\,;y) = -\zeta(m) + \int_0^y \frac{t}{t} G_{m-1}(t_{\pm}\,;y) - \int_0^{s_r} \frac{t}{t} G_{m-1}(t_{\pm}\,;y)\,. \tag{3.6}$$

   The second and third terms are now longer pending integrals, albeit with reduced weight

$$\begin{aligned}\Big(\vec{p},m,(0_{m-1},y)\Big) = -\zeta(m)&\Big(\vec{p},0,()\Big)\\ &+\Big((y,0),m-1,(0_{m-2};y)\Big)\Big(\vec{p},0,()\Big)\\ &-\Big((\vec{p},0),m-1,(0_{m-2};y)\Big)\,.\end{aligned}$$

2. In this case we end up simply with one large GPL

$$\int_0^{y'} \frac{s_1}{s_1-b_1}\cdots\int_0^{s_{r-1}} \frac{s_r}{s_r-b_r}\,G(\vec{a}\,;s_r) = G((\vec{b},\vec{a})\,;y')\,.$$

   In terms of pending integrals this is written as

$$\Big(\vec{p}=(y',\vec{b}),m+1,\vec{g}\Big) = G(\vec{b},\vec{g}\,;y')\,.$$

3. If there is no GPL under the pending integral, the integral evaluates to a GPL

$$\int_0^{y'} \frac{s_1}{s_1-b_1}\cdots\int_0^{s_{r-1}} \frac{s_r}{s_r-b_r} = G(b_1,...,b_r\,;y')\,.$$

In each case we end up with GPLs that are simpler in the sense that s_r has been eliminated. These might still be non-convergent due to other (non-zero) z_i elements being smaller in absolute value than y. But applying the removal of s_r recursively we can eliminate all z_i for which $|z_i|<|y|$. Therefore in the end we always obtain convergent GPLs.

3.4 Increase rate of convergence

Even though we have now only convergent GPLs, that does not imply that the convergence is fast enough for numerical applications. From now on we will only consider $y = 1$, as we can normalise any convergent GPL using (2.6). Convergence of such a GPL is slow if some z_i is close to the unit circle, i.e.

$$1 \leq |z_i| \leq \lambda < 2\,,$$

where λ is a parameter to be chosen.

Only for such z_i we apply the following strategy: to increase the rate of convergence we can use the fact that GPLs satisfy the convolution equation [1]

$$G(z_1,...,z_k\,;1) = \sum_{j=0}^{k}(-1)^j G\Big(1-z_j,...,1-z_1\,;1-\tfrac{1}{p}\Big)G\Big(z_{j+1},...,z_k\,;\tfrac{1}{p}\Big)\,,$$

where p is an arbitrary non-zero complex number. Separating the first and the last term of this sum we obtain for $p = 2$ and again normalising the GPLs on the right-hand side

$$\begin{aligned} G(z_1,...,z_k\,;1) = {}& G\big(2z_1,...,2z_k\,;1\big) + (-1)^k G\big(2(1-z_k),...,2(1-z_1)\,;1\big) \\ & + \sum_{j=1}^{k-1}(-1)^j G\Big(2(1-z_j),...,2(1-z_1)\,;1\Big)G\Big(2z_{j+1},...,2z_k\,;1\Big)\,. \end{aligned}$$

The first term has now better convergence as all parameters are twice as big. The GPL appearing in the sum all have reduced weight and are therefore not relevant for the present discussion.

The second term may or may not be convergent. If not, we repeat the algorithm outlined in Section *Making GPLs convergent*, including if necessary, convolution. At this stage it is not obvious why this recipe does indeed lead to a final answer and not to an infinite recursion. This can be shown by noting that the algorithm does only replace parameters with zero or permutes them; it does not introduce new non-trivial parameters. By carefully considering all possible behaviours under transformation $z \mapsto 2(1-z)$, [7] proved that this method indeed works.

The choice of λ is a trade-off between accuracy and speed. A typical choice would be $\lambda = 1.1$ which is the default in `handyG`. λ can be changed using the `hCircle` option in `set_options`.

3.5 An example reduction

To illustrate the various aspects discussed so far, we include here an example of how the algorithm works in practice. For this purpose we reduce $G(1,0,3;2)$ according to this algorithm until we end up with logarithms, polylogarithms and convergent MPLs. In our notation of a non-convergent GPL we have

$$G(\underbrace{1}_{s_r},\underbrace{0}_{a_2},\underbrace{3}_{a_3}\,;\underbrace{2}_{y}) = G(0,0,3;2) + \int_0^1 s_1 \frac{\partial}{\partial s_1} G(s_1,0,3;2)\,. \tag{3.7}$$

The first term corresponds to $G_3(3;2)$ and therefore it is a convergent trilogarithm. The second term has s_r appearing at the first place. Using (3.3) we obtain for the second term

$$\begin{aligned} \int_0^1 s_1 \frac{\partial}{\partial s_1} G(s_1,0,3\,;2) = {}& \int_0^1 \frac{s_1}{s_1-2} G(0,3\,;2) + \int_0^1 \frac{s_1}{s_1-0} G(s_1,3\,;2) \\ & - \int_0^1 \frac{s_1}{s_1-0} G(0,3\,;2)\,. \end{aligned} \tag{3.8}$$

The first and last terms are both conventional functions. Hence, we only need to worry about the second term which involves a pending integral. In order to evaluate it, we apply again (3.3) to the GPL under the pending integral to find

$$G(s_1,3\,;2) = G(0,3\,;2) + \int_0^{s_1} \frac{s_2}{s_2-2} G(3\,;2) + \int_0^{s_1} \frac{s_2}{s_2-3} G(s_2\,;2) - \int_0^{s_1} \frac{s_2}{s_2-3} G(3\,;2)\,. \tag{3.9}$$

Substituting this back into (3.8) gives

$$\int_0^1 \frac{s_1}{s_1-0} G(s_1,3\,;2) = \int_0^1 \frac{s_1}{s_1} G(0,3\,;2) + \int_0^1 \frac{s_1}{s_1} \int_0^{s_1} \frac{s_2}{s_2-2} G(3\,;2) + \int_0^1 \frac{s_1}{s_1} \int_0^{s_1} \frac{s_2}{s_2-3} G(s_2\,;2) - \int_0^1 \frac{s_1}{s_1} \int_0^{s_1} \frac{s_2}{s_2-3} G(3\,;2)\,. \tag{3.10}$$

Here only the third term is interesting, as the others are (poly)logarithms. The third term is a pending integral over a GPL of depth one. Thus,

$$\int_0^1 \frac{s_1}{s_1} \int_0^{s_1} \frac{s_2}{s_2-3} G(s_2\,;2) = \int_0^1 \frac{s_1}{s_1} \int_0^{s_1} \frac{s_2}{s_2-3} \Big(G(2\,;s_2) - G(0\,;s_2) + \log(-2) \Big) \tag{3.11}$$

The first two terms have s_r as the argument and hence they are GPLs. The last term is independent of s_r, making the integration trivial. Unfortunately, the second term $G(0,3,0;1)$ has a trailing zero. To remove it, we shuffle $G(0,3;1)$ with $G(0;1)$ to find

$$G(0,3\,;1)G(0\,;1) = \sum_{\vec{c}=(0,3)(0)} G(\vec{c}\,;1) = G(0,3,0\,;1) + 2 \times G(0,0,3\,;1)\,, \tag{3.12}$$

which we solve for $G(0,3,0\,;1)$.

Gathering all terms we obtain with $G(0;\,1) = \log 1 = 0$

$$\begin{aligned} G(1,0,3\,;2) = {} & \underbrace{G(0,0,3\,;2)}_{-3(2/3)} + \underbrace{G(2\,;1)}_{\log(1/2)} \underbrace{G(0,3\,;2)}_{-2(2/3)} - G(0\,;1)G(0,3\,;2) \\ & + G(0\,;1)G(0,3\,;2) + \underbrace{G(0,2\,;1)}_{-2(1/3)} \underbrace{G(3\,;2)}_{\log(1/3)} - \underbrace{G(0,3\,;1)}_{-2(2/3)} \underbrace{G(3\,;2)}_{\log(1/3)} \\ & + \underbrace{G(0,3,2\,;1)}_{2,1(1/3,3/2)} + \underbrace{G(0,3\,;1)}_{-2(1/3)} \log(-2) - G(0\,;1)G(0,3\,;1) \\ & + 2\underbrace{G(0,0,3;1)}_{-3(1/3)} = -0.81809 - 1.15049\,. \end{aligned}$$

CHAPTER

FOUR

FORTRAN REFERENCE GUIDE

Here we will list the functions of `handyG`

4.1 User-facing functions

`type real` (kind=prec) *[fixed]*

The real number type used in `handyG`. This cannot be changed at runtime by the user but should be used for all interactions with the code. It usually refers to double precision

`type inum`

The data type used for the prescription. This implements `abs`, `real`, and `aimag`.

Type fields

- **`% c`** *[complex(kind=prec)]* :: the complex number
- **`% i0`** *[integer(1)]* :: the prescription

`di0` *[type(inum),fixed]*

The default sign of the prescription

`subroutine GPLopts`(*mpldel*, *lidel*, *hcircle*)

a subroutine to set runtime parameters of `handyG`

Parameters

- **MPLdel** *[real(kind=prec),optional]* :: difference between two successive terms at which the series expansion (2.4) is truncated. Defaults to .
- **LiInf** *[*integer*,optional]* :: number of terms in the expansion of classical polylogarithms. Defaults to 1000.
- **hcircle** *[real(kind=prec),optional]* :: the size of the circle λ (see Section *Increase rate of convergence*). Defaults to 1.1

`function toinum`(*x*, *s*)

a function to convert one or more numbers to the *`inum`* type.

Parameters

- **x** *[in]* :: a scalar or vector of real, complex, or integer numbers
- **s** *[integer(1),optional]* :: the sign of the . Defaults to the value of *`di0`* if omitted.

function G(z, y)

the main GPL function in flat notation

Parameters

- **z** (*) *[in]* :: a list of the weights z_i of (2.1), either `real`, *complex*`, or *inum*.
- **y** *[in]* :: a argument y of (2.1), either `real`, *complex*`, or *inum*.

function G(m, z, y)

the main GPL function in condensed notation

Parameters

- **m** *[integer(:),in]* :: a list of the partial weights m_i of (2.3)
- **z** (*) *[in]* :: a list of the weights z_i of (2.3), either `real`, *complex*`, or *inum*.
- **y** *[in]* :: a argument y of (2.3), either `real`, *complex*`, or *inum*.

subroutine clearcache()

handyG caches a certain number of classical polylogarithms (see Section *Cache system*). This resets the cache (in a Monte Carlo this should be called at every phase space point).

4.2 Internal functions

4.2.1 Globals

This contains `real(kind=prec)`, *GPLopts()* (as `set_options`)

type integer (kind=ikin) *[fixed]*

The integer type using in `mpl_module` for the evaluation of multiple polylogarithms

zero *[real(kind=prec),parameter=*1e-15*]*

Values smaller than this are considered to be zero

MPLdelta *[real(kind=prec),protected]*

If the MPL sum changes less then this, it is truncated

Lidelta *[real(kind=prec),protected]*

If the polylog sum changes less then this, it is truncated

HoelderCircle *[real(kind=prec),protected]*

the size of the circle λ (see Section *Increase rate of convergence*)

PolyLogCacheSize *[integer(2),parameter=(/5,100/)]*

an array of two elements `(/ mmax, n /)`. At most `n` polylogs with weight `mmax` will be cached

i_ *[complex(kind=prec),parameter=*(0,*1)]*

the imaginary unit

verb *[*integer*]*

the verbosity of handyG

4.2.2 prescription

This contains *inum*, *di0*, *toinum()*

izero *[*inum,*paramter=0]*

the number $0+^{\pm}$ with the default prescription *di0*

marker *[*inum,*parameter*=opaque*]*

a marker used in *find_marker()*

function abs(*v*)

Parameters
v *[type(inum),in]* :: a scalar or vector *inum* value

Return
abs *[real(kind=prec)]* :: the absolute value of **v**

function real(*v*)

Parameters
v *[type(inum),in]* :: a scalar or vector *inum* value

Return
real *[real(kind=prec)]* :: the real part of **v**

function aimag(*v*)

Parameters
v *[type(inum),in]* :: a scalar or vector *inum* value

Return
aimag *[real(kind=prec)]* :: the imaginary part of **v**

4.2.3 Utilities

function get_condensed_m(*z*)

Parameters
z (*) *[type(inum),in]* :: the GPL weights

Return
m *[integer(size(z))]* :: condensed m where the ones not needed are filled with 0

function get_condensed_z(*m*, *z_in*)

Parameters

- **m** (*) *[*integer,*in]* :: the m vector
- **z_in** (*) *[type(inum),in]* :: the original flat GPL weights

Return
z *[type(inum) (size(m))]* :: the condesed z vector

function get_flattened_z(*m*, *z_in*)

Parameters

- **m** (*) *[*integer,*in]* :: the m vector
- **z_in** (*) *[type(inum),in]* :: the condensed GPL weights

Return
z *[type(inum) (sum(m))]* :: the flattened GPL weights

`function find_amount_trailing_zeros(z)`

Parameters
z (*) *[type(inum),in]* :: the GPL weights

Return
n *[integer]* :: the number of trailing zeroes

`function find_marker(z)`

Parameters
z (*) *[type(inum),in]* :: a list of GPL weights including a *marker*

Return
n *[integer]* :: the location of the *marker* (indexed at 1)

`function find_first_zero(v)`

Parameters
v (*) *[integer,in]* :: a list of integers

Return
n *[integer]* :: the location of the first zero or `-1` if no zero is found

`function min_index(r)`

Parameters
v (*) *[real(kind=prec),in]* :: a list of real numbers

Return
n *[integer]* :: the location of the smallest element

`function zeroes(n)`

Parameters
n *[integer,in]* :: the length of the resulting vector, can be zero

Return
z *[integer(n)]* :: a list of zeroes, potentially empty

`function factorial(n)`

calculates $n!$ iteratively

> **Warning:** This may return an incorrect result if `n` is too large to fit into the integer datatype. For 32 bit integers, this means `n<=12`.

Parameters
n *[integer,in]*

Return
res *[integer]* :: the factorial of `n`

`function binom`(n, r)

This implementation of the binomial coefficient is adapted from Rosetta Code which is published under the GNU Free Documentation License 1.2. It requires approximately $(1.55n - 2.5)$ bit integers. This means that we can go up to $n \approx 83$ for 128 bit and $n \approx 42$ on 64 bit compilers. While this could be restrictive the Bernoulli numbers this is used for are already $\mathcal{O}(10^{19})$ and $\mathcal{O}(10^{60})$. It is possible to extend this further by adding more prime numbers in the implementation

Parameters

- **n** *[integer,in]* :: the upper index
- **r** *[integer,in]* :: the lower index

Return

binom *[integer]* :: the binomial coefficient $\binom{n}{r}$

4.2.4 Shuffle algebra

The shuffle algebra is implemented recursively

$$\{a_1, a_2, \cdots\}\{b_1, b_2, \cdots\} = \begin{pmatrix} \{a_2, \cdots\}\{b_1, b_2, \cdots\} \oplus a_1 \\ \{a_1, a_2, \cdots\}\{b_2, \cdots\} \oplus b_1 \end{pmatrix}$$

where $\vec{a} \oplus b$ appends b to the vector $\vec{a}$.

`function append_to_each_row`(a, m)

Parameters

- **a** *[type(inum),in]* :: a scalar
- **m** (*,*) *[type(inum),in]* :: a list of vectors

Return

res *[type(inum)(size(m,1),size(m,2)+1)]* :: the list of vectors with `a` appended to each row

`function stack_matrices_vertically`(*m1*, *m2*)

Parameters

- **m1** (*,*) *[type(inum),in]* :: a list of vectors
- **m2** (*,*) *[type(inum),in]* :: a list of vectors

Return

res *[type(inum)(size(m1,1)+size(m2,1), size(m1,2))]* :: the matrix `m1` with the rows of `m2` appended

`function shuffle_product`(*v1*, *v2*)

Parameters

- **v1** (*) *[type(inum),in]* :: a list of numbers
- **v2** (*) *[type(inum),in]* :: a list of numbers

Return

res *[type(inum)(:, size(v1)+size(v2))]* :: a list of lists containing the shuffle product $v_1 v_2$

function shuffle_with_zero(a)

Parameters
a (*) *[type(inum),in]* :: a list of numbers

Return
res *[type(inum)(size(a)+1, size(a)+1)]* :: the list $a\{0\}$

4.2.5 Mathematical tools

zeta *[real(kind=prec),parameter=*Zeta[2..10]*]*

The Riemann function for integer values between 2 and 10

DirichletBeta *[real(kind=prec),parameter=DirichletBeta*[2..10]*]*

The Dirichlet function for integer values between 2 and 10

type el

The data type used for the polylog cache containing the complex argument and the result. The weight is addressed using the index in `cache`

Type fields

- **% c** *[complex(kind=prec)]* :: the complex argument
- **% ans** *[complex(kind=prec)]* :: the result of $_m(c)$

cache *[type(el)(PolyLogCacheSize(1),PolyLogCacheSize(2))]*

The polylogarithm cache, the size is controlled using `PolyLogCacheSize`. The first index tracks the weight m and the second the pair $\{c,_m(c)\}$

plcachesize *[integer(PolyLogCacheSize(1))]*

The number of occupied `cache` elements.

function naive_polylog(m, x)

A naive series implementation of the classical polylogarithm until i^m rolls over or the new term is less than `LiDelta`

$$_m(z) = \sum_{n=1}^{\infty} \frac{x^i}{i^m}$$

This function is not meant to be called directly

Parameters

- **m** *[*integer*]* :: the weight
- **x** *[complex(kind=prec)]* :: the argument

Return
res *[complex(kind=prec)]* :: the resulting $_m(x)$

function bernoullinumber(n)

This returns the *n*-th Bernoulli number by computing all Bernoulli numbers up to the *n*-th recursively using the relation

$$\sum_{k=0}^{m} \binom{m+1}{k} B_k = 0$$

for $m > 0$. Solving this for B_m results in

$$B_m = -\sum_{k=0}^{m-1} \binom{m}{k} \frac{B_k}{m-k+1}$$

for $m > 0$ and $B_0 = 1$. Care is taken to avoid multiple computation by using a dynamic cache.

Warning: The implementation of *binom()* limits this to roughly 42 when working with 32 bit integers.

Parameters
n *[integer]* :: the index of the Bernoulli number

Return
res *[real(kind=prec)]* :: the resulting B_n.

`function harmonicnumber(n)`

The harmonic number

$$H_n = \sum_{i=1}^{n} \frac{1}{n}$$

for $n \leq 40$.

Parameters
n *[integer]* :: the index of the harmonic number

Return
res *[real(kind=prec)]* :: the resulting H_n

`function logz_polylog(n, z)`

Computes the classical polylogarithm ${}_n(z)$ using series representation in $\log z < 2\pi$. The algorithm works by using (1.4) of [2]

$${}_m(z) = \sum_{i=0}^{\infty}{}^* \zeta_{m-i} \frac{\log^i z}{i!} + \log^{m-1} z \frac{H_{m-1} - \log(-\log z)}{(n-1)!}$$

$\sum^*$ excludes the singular ζ_1 term at $m = n-1$. In Fortran, we split this in a sum from $0, \cdots, n-2$ with positive arguments in the Zeta function. The next term $m = n$ we do manually to not have to implement $\zeta_0 = -1/2$ and then we use $\zeta_{n-m} = (-1)^{m-n} B_{1+m-n}/(1+m-n)$ for the remaining terms.

Parameters

- **n** *[integer]* :: the weight
- **z** *[complex(kind=prec)]* :: the argument

Return
res *[complex(kind=prec)]* :: the resulting ${}_m(x)$

`function Li2(x)`

The real dilogarithm using Chebyshev interpolation and the Clenshaw algorithm as done in CERNLib `C332`

Parameters
x *[real(kind=prec)]* :: the argument x < 1

Return
res *[real(kind=prec)]* :: the result ${}_2(x) \in \mathbb{R}$

`function dilog`(x)

An optimised evaluation for ${}_2(z)$ for $|z| < 1$ using either *Li2()*, *logz_polylog()*, or *naive_polylog()*

Parameters
z *[complex(kind=prec)]* :: the argument

Return
res *[complex(kind=prec)]* :: the resulting ${}_2(x)$

`function Li3`(x)

The real trilogarithm using Chebyshev interpolation and the Clenshaw algorithm as done in CERNLib `C332`

Parameters
x *[real(kind=prec)]* :: the argument x < 1

Return
res *[real(kind=prec)]* :: the result ${}_3(x) \in \mathbb{R}$

`function trilog`(x)

An optimised evaluation for ${}_3(z)$ for $|z| < 1$ using either *Li3()*, *logz_polylog()*, or *naive_polylog()*

Parameters
z *[complex(kind=prec)]* :: the argument

Return
res *[complex(kind=prec)]* :: the resulting ${}_3(x)$

`function BERNOULLI_POLYNOMIAL`(n, x)

Calculate the n-th Bernoulli polynomial up to $n = 15$ using hard-coded coefficients

Parameters

- **n** *[integer]* :: the weight $n \leq 15$
- **x** *[complex(kind=prec)]* :: the argument

Return
res *[complex(kind=prec)]* :: the resulting $B_n(x)$

`function mylog`(x)

Calculates the logarithm of a complex number x taking care to have the correct imaginary part for small but non-zero $\Im x$.

Parameters
x *[complex(kind=prec)]*

Return
res *[complex(kind=prec)]* :: the result $\log(x)$

`function polylog`(m, x)

Calculates and cache the polylogarithm of x by recursively mapping x into a region where the result can be easily obtained. For $x = \pm 1$, we use the function and for for $x = \pm$ß, we use the function. For $|x| > 1$ we remap to $x \to 1/x$ and for $\frac{1}{2} < |x| < 2$ we use *logz_polylog()*.

Parameters

- **m** *[integer]* :: the weight
- **x** *[complex(kind=prec)]* :: the argument

Return
res *[complex(kind=prec)]* :: the result ${}_m(x)$

function polylog(m, x, y)

Calculates ${}_m(x/y)$ for two *inum*

Parameters

- **m** *[integer]* :: the weight
- **x** *[type(inum)]* :: the numerator
- **y** *[type(inum)]* :: the denominator

Return

res *[complex(kind=prec)]* :: the result ${}_m(x/y)$

function plog1(a, b)

Calculates $\log(1 - a/b)$ for two *inum*

Parameters

- **a** *[type(inum)]* :: the numerator
- **b** *[type(inum)]* :: the denominator

Return

res *[complex(kind=prec)]* :: the result $\log(1 - a/b)$

subroutine clearcache()

Clears the polylogarithm cache

4.2.6 Convergent multiple polylogarithms

underflowalert *[real(kind=prec),parameter=*1e-250*]*

A value to detect floating precision underflow in *MPL()*.

overflowalert *[real(kind=prec),parameter=*1e+250*]*

A value to detect floating precision overflow in *MPL()*.

cachesize *[integer(2),parameter=(/4,2500/)]*

The maximum weight and number of MPLs to cache

type el

The data type used for the MPL cache containing the complex arguments and the result. The weight is addressed using the index in *cache*

Type fields

- **% c** (*cachesize*(1) *[complex(kind=prec)]* :: the complex argument
- **% ans** *[complex(kind=prec)]* :: the result of the MPL

cache *[type(el)(cachesize(1),cachesize(2))]*

The MPL cache, the size is controlled using *cachesize*. The first index number of arguemnts and the index in the cache

function MPL_converges(m, x)

Checks whether an MPL of weight *m* with arguments *x* converges

Parameters

- **m** (*) *[integer,in]* :: the weight vector
- **x** (*) *[complex(kind=prec),in]* :: the argument vector

Return
converges *[logical]* :: `.true.` if the MPL converges without transformation, `.false.` otherwise.

`function check_cache`(*m*, *x*, *res*)

Performs a lookup in the MPL *cache()*.

Parameters

- **m** (*) *[integer,in]* :: the weight vector
- **x** (*) *[complex(kind=prec),in]* :: the argument vector

Return

- **res** *[complex(kind=prec)]* :: the result of the MPL if it is in the cache
- **cached** *[logical]* :: `.true.` is in the cache, `.false.` otherwise.

`function MPL`(*m*, *x*)

Calculates a multiple polylogarithm using the series expansion (2.4)

$$_{m_1,...,m_k}(x_1,...,x_k) = \sum_{i_1>\cdots>i_k}^{\infty} \frac{x_1^{i_1}}{i_1^{m_1}} \cdots \frac{x_k^{i_k}}{i_k^{m_k}} ,$$

The expansion aborts if either $x_j^i <$ `underflowalert`, $i^m <$ `overflowflowalert` or the difference between successive terms becomes smaller than *MPLdelta*.

Parameters

- **m** (*) *[integer]* :: the weight vector
- **x** (*) *[complex(kind=prec)]* :: the argument vector

Return
res *[complex(kind=prec)]* :: the resulting MPL

4.2.7 Generalised polylogarithms

`function GPL_zero_zi`(*l*, *y*)

computes the value of a GPL when all $z_i = 0$ using (2.2)

Parameters

- **l** *[integer]* :: the number of zeros
- **y** *[type(inum)]* :: the argument

Return
res *[complex(kind=prec)]* :: the resulting GPL

`function is_convergent`(*z*, *y*)

checks whether a given flat GPL has a convergent series representation using (2.7)

Parameters

- **z** (*) *[type(inum)]* :: the weight vector
- **y** *[type(inum)]* :: the argument

Return
is_convergent *[logical]* :: `.true.` if the GPL is convergent, `.false.` otherwise

function remove_sr_from_last_place_in_PI(*a*, *y2*, *m*, *p*, *srs*)

Similar to *remove_sr_from_last_place_in_G()*, this uses the shuffle algebra to remove the smallest element s_r from the last position of the GPL

Parameters

- **a** (*) *[type(inum)]* :: the weights up to s_r without trailing zeroes
- **y2** *[type(inum)]* :: the argument of the inner GPL
- **m** *[integer]* :: the number of weights
- **p** (*) *[type(inum)]* :: the $\vec{p} = (y', \vec{b})$ of (3.5) where the y' is the upper limit of the final integration and $\vec{b}$ the weight vector of the pending integral
- **srs** *[integer(1)]* :: the prescription of the original s_r

Return

res *[complex(kind=prec)]* :: the result of the reduction

function pending_integral(*p*, *i*, *g*, *srs*)

evaluates a pending integral (3.5) by reducing it to simpler ones and pure GPLs

$$\left(\vec{p} = (y', \vec{b}), i, \vec{g} = (\vec{a}, y)\right) \equiv \int_0^{y'} \frac{s_1}{s_1 - b_1} \int_0^{s_1} \frac{s_2}{s_2 - b_2} \cdots \int_0^{s_{r-1}} \frac{s_r}{s_r - b_r} G(a_1, ..., a_{i-1}, s_r, a_{i+1}, ..., a_m \,; y) \,.$$

See Section *Evaluation of pending integrals* for further details

Parameters

- **p** (*) *[type(inum)]* :: the $\vec{p} = (y', \vec{b})$ of (3.5) where the y' is the upper limit of the final integration and $\vec{b}$ the weight vector of the pending integral
- **i** *[integer]* :: the position of the smallest element s_r that was removed in the original weight vector
- **g** (*) *[type(inum)]* :: the $\vec{g} = (\vec{a}, y)$ of (3.5) where the $\vec{a}$ are the weights of the original GPL (with the smallest element removed) and y is its argument.
- **srs** *[integer(1)]* :: the prescription of the original s_r

Return

res *[complex(kind=prec)]* :: the result of the reduction

function remove_sr_from_last_place_in_G(*a*, *y2*, *m*, *sr*)

This uses the shuffle algebra to remove the smallest element s_r from the last position of the GPL $G(a_1, ..., a_{m-1}, s_r, 0, 0, ..., 0; y)$

Parameters

- **a** (*) *[type(inum)]* :: the weights up to s_r
- **y2** *[type(inum)]* :: the argument of the GPL
- **m** *[integer]* :: the number of weights
- **sr** *[type(inum)]* :: the smallest non-zero element

Return

res *[complex(kind=prec)]* :: the result of the reduction

`function make_convergent`(a, $y2$)

This reduces a given GPL to more convergent or simpler objects by using the algorithm in Section *The algorithm*

Parameters

- **a** (*) *[type(inum)]* :: the weight vector
- **y2** *[type(inum)]* :: the argument

Return

res *[complex(kind=prec)]* :: the result of the reduction

`function improve_convergence`(z)

improves the convergence by applying the convolution to $G(z_1, ..., z_k; 1)$

> **Warning:** In the Hoelder expression, all the $(1 - z)$ are $-^+$. GiNaC does something different (and confusing). As we do, they usually would set `i0` to `-z%i0`. However, if $\Im z = 0$ and $\Re z \geq 1$, they just set it to `+i0`, be damned what it was before.

Parameters

z (*) *[type(inum)]* :: the normalised weights vector

Return

res *[complex(kind=prec)]* :: the result of the reduction

4.3 Cache system

handyG has a cache systems for classical polylogarithms and one for GPLs. It is controlled through the parameter

```
integer, parameter :: PolyLogCacheSize(2) = (/ n, mmax /)
```

This caches n polylogarithms of the form ${}_m(x)$ for $2 \leq m \leq m_{\max}$ each. The default values are $n = 100$ and $n_{\max} = 5$. This cache system consumes

$$n \times m_{\max} \times \big(2 \times \texttt{sizeof(complex(kind=prec))} + 1\text{byte} + \text{padding}\big) = 12\,\text{kB}$$

bytes of memory in the default settings. This is a very small price to pay for improving the evaluation speed considerably.

CHAPTER

FIVE

KNOWN ISSUES

Here we make a list of known issues that have occurred. If you experience a problem with handyG, please do not hesitate to contact us. Ideally, your bug report should contain

- The version of handyG you are using. This can be found at the end of the `./configure` procedure. Please understand that older releases or development versions are not fully supported and that you may be required to update the latest version.
- The logfile produced by the `./configure` process. This can be obtained by prepending the `./configure` call with for example `LOGFILE=log.txt`.
- If applicable, a short example program demonstrating your problem. For a timely response, please provide the simplest program that still causes your issue.
- Your issue may result in a new release or an addition on this page. By default, we will acknowledge you for your bug report and maybe publish parts of your example code. Please let us know if you object to this.
- If you already have investigated your issue, please share your results, though this *is not necessary*.

5.1 Segmentation fault for arguments on the complex unit circle

Thanks to F. Buccioni for reporting this issue

Sometimes GPLs with arguments on the unit circle, i.e. $G(z_1, ..., z_m\ ; y)$ with $y \in \{c \in \mathbb{C}\ :\ |c| = 1\}$ result in segmentation faults.

```
! compile with gfortran -o demo demo.f90 libhandyg.a
program handyGdemo
  use handyG
  implicit none
  real(kind=prec) :: z
  complex(kind=prec) :: y
  complex(kind=prec), parameter :: i_ = (0._prec, 1._prec)

  z = 0.99592549661823904_prec
  y = (1._prec - 2*z + i_*sqrt(4*z-1._prec))/(2*z)

  print*, G([(-1._prec,0._prec),(-1._prec,0._prec)],y)

end program handyGdemo
```

The above code may result in a segmentation fault due to an infinite loop. GPLs with complex arguments are first normalised. In the above case, the GPL evaluate is $G(-1/y, -1/y\ ; 1)$. Due a numerical issue in Fortran `abs(-1/`

y) may evaluate to a number slightly less than one. This problem can easily be circumvented by performing the normalisation analytically

```
moy = cmplx(1._prec-1/(2*z), sqrt(4*z-1._prec) / (2*z), kind=prec)
print*, G([moy, moy],(1._prec,0.))
```

5.2 GPLs with arguments close to one are not precise

Thanks to Xiofeng Xu for pointing out this issue

The function that calculates $_n(z)$ for $z \in \mathbb{C}$ is not precise for $z \sim 1$ because the series expansion converges too slow. This should be resolve in `v0.1.4`.

5.3 Parallel builds are not supported

Thanks to R. K. Eillis and J. Campbell for pointing out this issue

`make -j` fails because dependencies are not correctly implemented. This should be resolved on `master` and will be part of `v0.1.5`.

CHAPTER

SIX

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Jonathan M. Borwein, David M. Bradley, David J. Broadhurst, and Petr Lisonek. Special values of multiple polylogarithms. *Trans. Am. Math. Soc.*, 353:907–941, 2001. arXiv:math/9910045, doi:10.1090/S0002-9947-00-02616-7.

[2] R. Crandall. Note on fast polylogarithm computation. 01 2006. URL: https://www.reed.edu/physics/faculty/crandall/papers/Polylog.pdf.

[3] Claude Duhr. Mathematical aspects of scattering amplitudes. In *Theoretical Advanced Study Institute in Elementary Particle Physics: Journeys Through the Precision Frontier: Amplitudes for Colliders*, 419–476. 2015. arXiv:1411.7538, doi:10.1142/9789814678766_0010.

[4] Claude Duhr and Falko Dulat. PolyLogTools — polylogs for the masses. *JHEP*, 08:135, 2019. arXiv:1904.07279, doi:10.1007/JHEP08(2019)135.

[5] Hjalte Frellesvig, Damiano Tommasini, and Christopher Wever. On the reduction of generalized polylogarithms to $\text Li_n$ and $\text Li_2,2$ and on the evaluation thereof. *JHEP*, 03:189, 2016. arXiv:1601.02649, doi:10.1007/JHEP03(2016)189.

[6] L. Naterop, A. Signer, and Y. Ulrich. handyG —Rapid numerical evaluation of generalised polylogarithms in Fortran. *Comput. Phys. Commun.*, 253:107165, 2020. arXiv:1909.01656, doi:10.1016/j.cpc.2020.107165.

[7] Jens Vollinga and Stefan Weinzierl. Numerical evaluation of multiple polylogarithms. *Comput. Phys. Commun.*, 167:177, 2005. arXiv:hep-ph/0410259, doi:10.1016/j.cpc.2004.12.009.

INDEX

A

abs() (*fortran function*), 17
aimag() (*fortran function*), 17
append_to_each_row() (*fortran function*), 19

B

BERNOULLI_POLYNOMIAL() (*fortran function*), 22
bernoullinumber() (*fortran function*), 20
binom() (*fortran function*), 18

C

cache (*fortran variable*), 20, 23
cachesize (*fortran variable*), 23
check_cache() (*fortran function*), 24
clearcache() (*fortran subroutine*), 16, 23

D

di0 (*fortran variable*), 15
dilog() (*fortran function*), 21
DirichletBeta (*fortran variable*), 20

E

el (*fortran type*), 20, 23

F

factorial() (*fortran function*), 18
find_amount_trailing_zeros() (*fortran function*), 18
find_first_zero() (*fortran function*), 18
find_marker() (*fortran function*), 18

G

G() (*fortran function*), 15, 16
get_condensed_m() (*fortran function*), 17
get_condensed_z() (*fortran function*), 17
get_flattened_z() (*fortran function*), 17
GPL_zero_zi() (*fortran function*), 24
GPLopts() (*fortran subroutine*), 15

H

harmonicnumber() (*fortran function*), 21
HoelderCircle (*fortran variable*), 16

I

i_ (*fortran variable*), 16
improve_convergence() (*fortran function*), 26
integer (*fortran type*), 16
inum (*fortran type*), 15
is_convergent() (*fortran function*), 24
izero (*fortran variable*), 17

L

Li2() (*fortran function*), 21
Li3() (*fortran function*), 22
Lidelta (*fortran variable*), 16
logz_polylog() (*fortran function*), 21

M

make_convergent() (*fortran function*), 25
marker (*fortran variable*), 17
min_index() (*fortran function*), 18
MPL() (*fortran function*), 24
MPL_converges() (*fortran function*), 23
MPLdelta (*fortran variable*), 16
mylog() (*fortran function*), 22

N

naive_polylog() (*fortran function*), 20

O

overflowalert (*fortran variable*), 23

P

pending_integral() (*fortran function*), 25
plcachesize (*fortran variable*), 20
plog1() (*fortran function*), 23
polylog() (*fortran function*), 22
PolyLogCacheSize (*fortran variable*), 16

R

real (*fortran type*), 15
real() (*fortran function*), 17

remove_sr_from_last_place_in_G() (*fortran function*), 25
remove_sr_from_last_place_in_PI() (*fortran function*), 24

S

shuffle_product() (*fortran function*), 19
shuffle_with_zero() (*fortran function*), 19
stack_matrices_vertically() (*fortran function*), 19

T

toinum() (*fortran function*), 15
trilog() (*fortran function*), 22

U

underflowalert (*fortran variable*), 23

V

verb (*fortran variable*), 16

Z

zero (*fortran variable*), 16
zeroes() (*fortran function*), 18
zeta (*fortran variable*), 20