User manual for \texttt{bch}, a program for the fast computation of the Baker–Campbell–Hausdorff and similar series

Harald Hofstätter  
Reitschachersiedlung 4/6, 7100 Neusiedl am See, Austria  
hofi@harald-hofstaetter.at

Program version 1.2, October 11, 2021

Abstract
This manual describes \texttt{bch}, an efficient program written in the C programming language for the fast computation of the Baker–Campbell–Hausdorff (BCH) and similar Lie series. The Lie series can be represented in the Lyndon basis, in the classical Hall basis, or in the right-normed basis of E.S. Chibrikov. In the Lyndon basis, which proves to be particularly efficient for this purpose, the computation of 111,013 coefficients for the BCH series up to terms of degree 20 takes less than half a second on an ordinary personal computer and requires negligible 11 MB of memory. Up to terms of degree 30, which is the maximum degree the program can handle, the computation of 74,248,451 coefficients takes 55 hours but still requires only a modest 5.5 GB of memory.

1 Introduction
We consider the element
\begin{equation}
H = \log(e^A e^B) = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} (e^A e^B - 1)^k = \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} \left( \sum_{i+j>0} \frac{1}{i! j!} A^i B^j \right)^k
\end{equation}
in the ring $\mathbb{Q}\langle\langle A, B \rangle\rangle$ of formal power series in the non-commuting variables $A$ and $B$ with rational coefficients. This element $H$ is commonly called the Baker–Campbell–Hausdorff (BCH) series. A classical result known as the Baker–Campbell–Hausdorff theorem (see, e.g., [7]) states that $H$ is a Lie series, which means that the homogeneous components $H_n$ of degree $n = 1, 2, \ldots$ in
\begin{equation}
H = \sum_{n=1}^{\infty} H_n,
\end{equation}
can be written as linear combinations of $A$ and $B$ and (possibly nested) commutator terms in $A$ and $B$, i.e., they are elements of the free Lie algebra $\mathcal{L}_Q(A, B)$ generated by $A$ and $B$.

The program \texttt{bch} computes the terms of the Lie series $H$ up to terms of a given maximal degree $N$, where $H$ (or more precisely, each component $H_n$, $n = 1, \ldots, N$) is optionally represented in the Lyndon basis, the right-normed basis of E.S. Chibrikov [3] or the classical

\footnote{The right-normed Chibrikov basis consists of right-normed (i.e., right-nested) commutators of the form $[a_1, [a_2, \ldots, [a_{n-1}, a_n] \ldots]]$ where $a_i \in \{A, B\}$. It seems that there are no reports in the literature on computations of the BCH series in terms of linearly independent sets of such right-normed commutators up to high degree. In [4] such computations are performed only up to degree $N = 10$, and the approach taken there does not seem to be as systematic, general and efficient as that of our \texttt{bch} program, which has computed the BCH series in the right-normed Chibrikov basis up to degree $N = 22$, see Section [5].}
Hall basis. Furthermore, the program can straightforwardly be extended to compute the representation in an arbitrary generalized Hall basis.

In addition to $H = \log(e^Ae^B)$ the program $\texttt{bch}$ can compute the Lie series for arbitrary expressions of the form

$$X = \log(e^{\Phi_1} \cdots e^{\Phi_n}),$$

(3)

where the $\Phi_i$ are Lie polynomials with rational coefficients in two or more non-commuting variables like the symmetric BCH series $\log(e^{A_1B_1}e^{A_2B_2})$, the BCH series with 3 generators $\log(e^Ae^Be^C)$, or the more complex expression $\log(e^{\frac{1}{3}A_1}e^{\frac{1}{3}A_2}e^{\frac{2}{3}B_1}e^{\frac{1}{3}[B_1,[A_1,B_1]]}e^{\frac{1}{2}A_2}e^{\frac{1}{2}B_1})$ with a commutator in an exponential.

2 Implementation

Here we only describe the main ideas on which the implementation of $\texttt{bch}$ is based, details will be available in [10].

The program $\texttt{bch}$ is implemented in the C programming language according to the C99 (or later) standard. The C code for parsing expressions of the form (3) has been generated by GNU Bison [5]. The program uses $\texttt{khash.h}$ from the $\texttt{klib}$ library [12], which is a very efficient implementation of hash tables. Apart from $\texttt{khash.h}$ it is self-contained, i.e., independent of external libraries except the standard C library.

In particular, the program does not require a library for multi-precision integer or rational arithmetic. Instead all calculations are carried out in pure integer arithmetic using the 128 bit integer type $\texttt{__int128_t}$, which is available for most current C compilers on current hardware. With this integer type computations of the BCH series up to degree $N = 30$ are possible. To avoid calculations with rational numbers, the program first determines a common denominator for all rational numbers that can occur during the computation (see [8], [9]), then all calculations are reorganized in such a way that they can actually be carried out in pure integer arithmetic.

In a preliminary step, the program computes the coefficients of all Lyndon words up to degree $N$ in the power series of the given expression $X$ of the form (3) using an adaptation of the algorithm from [11]. In the special case $X = \log(e^Ae^B)$ it alternatively uses the algorithm from the appendix of [9] and exploits the symmetry that the coefficient of the word $A^{q_1}B^{q_2} \cdots (A \lor B)^{q_m}$ is invariant under permutations of the exponents $q_1, \ldots, q_m$, see [6].

Next, the program transforms the coefficients of the Lyndon words into coefficients of Lie basis elements in the Lie series. For the Lyndon basis and the right-normed Chibrikov basis this is done by solving a linear system whose matrix consists of coefficients of Lyndon words in basis elements.

In the Lyndon basis case this matrix with integer entries is triangular with all diagonal elements equal 1, which is a consequence of [13, Theorem 5.1]. Thus, this integer system is readily solved in integer arithmetic with no further denominators introduced. Here the performance bottleneck is actually the computation of the coefficient matrix, for which a new efficient algorithm has been developed, for details see [10].

In the right-normed Chibrikov basis case, the coefficient matrix is no longer triangular, but it still has determinant $\pm 1$, and its leading principal minors are all equal $\pm 1$. It follows that the matrix has a $LU$-decomposition with triangular matrices $L$, $U$ whose entries are all integers and whose diagonal elements are equal $\pm 1$. This $LU$-decomposition can be computed in pure integer arithmetic using Gaussian elimination without any row or column permutations. Again solving such a linear system introduces no further denominators.

In the Hall basis case, the program first computes the representation of the Lie series in the Lyndon basis, and then uses a standard rewriting algorithm (see [13, Section 4.2])
applied to the Lyndon basis elements to obtain a representation in the Hall basis. Because this rewriting algorithm is carried out in integer arithmetic, again no further denominators are introduced.

The special case $X = \log(e^Ae^B)$ allows a significant optimization in the Lyndon basis or right-normed Chibrikov basis case, if $N$ is even: Here the Lie series is computed using the above method only up to terms of degree $N - 1$. The terms of degree $N$ can then efficiently be computed using a formula due to E. Eriksen, see [4, Section III.A].

As a final remark on the implementation we note that the computation can be organized in such a way that finely homogeneous components $H_{(a,b)}$ of $H$ are computed completely independently of each other. Here a finely homogeneous component $H_{(a,b)}$ of multi-degree $(a,b)$ is of degree $a$ in $A$ and of degree $b$ in $B$ such that

$$H_n = \sum_{a+b=n} H_{(a,b)}.$$

Thus, the computations of these components can be done in parallel, giving a simple but effective parallelization strategy, which has been implemented using OpenMP.

3 Installation

The source code can be downloaded from the GitHub repository

https://github.com/HaraldHofstaetter/BCH

On a Unix-like system with gcc compiler available just type

```bash
$ make
```

in a directory containing the source code, which causes the shared library libbch.so and the executable bch to be created. To use a different compiler, the Makefile has to be adapted accordingly.

The bch program has been compiled to WebAssembly [14]. So one can try it in the browser without installing it. For example, to compute the BCH series represented in the Lyndon basis up to terms of degree $N = 20$ just open the following link:

http://www.harald-hofstaetter.at/BCH/bch.html?verbosity_level=1&N=20&basis=0

Parameters are specified as shown. Their usage is (up to obvious modifications, e.g. ‘&’ instead of whitespace) exactly as described in Section 4.

4 Usage of the bch program

The program invoked without any arguments produces the following output:

```bash
$ ./bch
+1/1*A+1/1*B+1/2*[A,B]+1/12*[A,[A,B]]+1/12*[[A,B],B]+1/24*[A,[[A,B],B]]-1/720*[A,[[A,B],[A,B]]]+1/180*[A,[A,[A,B]]]+1/360*[[A,B],[A,B]]+1/180*[A,[[[A,B],[A,B]]],B]+1/120*[[[A,B],[A,B]],B]-1/720*[[[[A,B],[A,B]],B],B]
```

This is the BCH series (i.e., the Lie series for $\log(e^Ae^B)$) up to terms of degree $N = 5$ represented in the Lyndon basis.

Arguments have the form

```
parameter=value
```

Such arguments may be combined arbitrarily. The following parameters are available (with default values in square brackets):

```bash
3
```
N[=5]

The maximal degree up to which the Lie series is computed.

```
$ ./bch N=8
+1/1*A+1/1*B+1/2*[A,B]+1/12*[A,[A,B]]+1/12*[B,[A,B]]+1/24*[A,[A,[A,B],B]]+1/720*[A,[A,[A,[A,B],B],B]]+1/180*[A,[A,[A,[A,B],B],B],B]]+1/360*[A,[A,[A,B],[A,B]],[A,B]]+1/180*[A,[A,[A,B],[A,B],[A,B]]]+1/120*[A,[A,B],[A,B],[A,B]]+1/1440*[A,[A,[A,[A,B],B],B],B]-1+1/720*[A,[A,[A,[A,B],B],B],B]]+1/240*[A,[A,[A,B],[A,B]],[A,B]]+1/120*[A,[A,[A,B],[A,B]],[A,B]]+1/1440*[A,[A,B],[A,B],[A,B]]+1/30240*[A,[A,B],[A,B],[A,B]]
```

basis[=0]

With default value `basis=0` the result is represented in the Lyndon basis; with value `basis=1` in the right-normed Chibrikov basis:

```
$ ./bch basis=1
+1/1*A+1/1*B-1/2*[B,A]+1/12*[B,[A,B]]+1/24*[B,[A,[B,A]]]+1/720*[A,[A,[A,[A,B],B],B],B]]+1/180*[A,[A,[A,[A,B],B],B],[A,B]]+1/360*[A,[A,[A,B],[A,B]],[B,A]]+1/180*[A,[A,[A,B],[A,B]],[B,A]]+1/120*[A,[A,B],[A,B],[B,A]]+1/1440*[A,[A,B],[A,B],[B,A]]+1/30240*[A,[A,B],[A,B],[B,A]]
```

With value `basis=2` the result is represented in the classical Hall basis:

```
$ ./bch basis=2
+1/1*A+1/1*B+1/2*[B,A]+1/12*[B,[B,A]]+1/24*[B,[A,[B,A]]]+1/720*[A,[A,[A,[A,B],B],B],B]]+1/180*[A,[A,[A,[A,B],B],B],[A,B]]+1/360*[A,[A,[A,B],[A,B]],[B,A]]+1/180*[A,[A,[A,B],[A,B]],[B,A]]+1/120*[A,[A,B],[A,B],[B,A]]+1/1440*[A,[A,B],[A,B],[B,A]]+1/30240*[A,[A,B],[A,B],[B,A]]
```

generators[=ABC...]

This parameter enables alternative names for the generators:

```
$ ./bch generators=xy
+1/1*x+1/1*y+1/2*[x,y]+1/12*[x,[x,y]]+1/12*[y,[x,y]]+1/24*[x,[x,[x,y]]]+1/720*[x,[x,[x,[x,y]]]]+1/180*[x,[x,[x,[x,y]]]]+1/360*[x,[x,[x,[x,y]]],[x,y]]+1/180*[x,[x,[x,[x,y]]],[x,y]]+1/120*[x,[x,[x,[x,y]]],[x,y]]+1/720*[x,[x,[x,[x,y]]],[x,y]]
```

expression[=0]

The program can compute the Lie series of arbitrary Lie expressions of the form $e^{A}e^{B}$. Some expressions are predefined:

- `expression=0` – Compute Lie series for $log(e^{A}e^{B})$, the classical BCH formula.
• expression=1 – compute Lie series for log(e^{\frac{1}{2}A}e^{\frac{1}{2}B}), the symmetric BCH formula:

\$ ./bch expression=1
+1/1*A+1/1*B-1/12*[[A,B],B]+7/5760*[A,[[[A,B],B],B]]-7/1440*[A,[A,[[A,B],B]]]+1/180*[[A,[[[A,B],B],B]],B]+1/120*[[[[A,B],B],B],B]-1/720*[[[[[A,B],B],B],B],B] \$

• expression=2 – compute Lie series for log(e^{A}e^{B}e^{A}), another symmetric version of the BCH formula:

\$ ./bch expression=2
+2/1*A+1/1*B-1/6*[[A,B],B]+7/360*[A,[[[A,B],B],B]]-7/180*[A,[[[A,B],B],B]]+1/45*[[[A,B],B],B]+1/30*[[[A,B],B],B]-1/360*[[[[A,B],B],B],B] \$

• expression=3 – compute Lie series for log(e^{A}e^{B}e^{C}), a BCH formula with three different exponentials:

\$ ./bch expression=3 N=4
+1/1*A+1/1*B+1/1*C+1/2*[A,B]+1/2*[A,[A,B]]+1/12*[[A,B],B]+1/12*[[[A,B],B],B]+1/24*[[[A,B],B],B]+1/12*[[[[A,B],B],B],B] \$

• expression=4 – compute Lie series for log(e^{A}e^{B}e^{-A}e^{-B}),

\$ ./bch expression=4
+1/1*A+1/1*B-1/2*[[[A,B],B],B]+1/6*[A,[[A,B],B]]-1/4*[[A,[A,B],B],B]+1/6*[A,[[[[A,B],B],B],B]]-1/4*[[A,[[[A,B],B],B],B],B] \$

• expression=5 – compute Lie series for log(e^{\frac{1}{2}B}e^{\frac{1}{2}A}e^{\frac{1}{2}B+\frac{1}{12}[[B,A,B]]}e^{\frac{1}{2}B}e^{\frac{1}{2}A}e^{B}e^{A}), a more complicated version of the BCH formula, which is constructed in such a way that besides A + B all terms have degree ≥ 5:

\$ ./bch expression=5
+1/1*A+1/1*B+1/1*C+1/2*[[[A,B],B],B]+1/6*[A,[[A,B],B]]+1/24*[[[A,B],B],B]+1/12*[[[[A,B],B],B],B]+1/24*[[[[[A,B],B],B],B],B] \$

An arbitrary expression of the form (3) can directly be specified by its formula as

"expression= formula"

Here the double quotes are necessary because the formula usually contains special characters like ‘*’ or brackets that confuse the command line shell. For example the above Lie series for log(e^{\frac{1}{2}B}e^{\frac{1}{2}A}e^{\frac{1}{2}B+\frac{1}{12}[[B,A,B]]}e^{\frac{1}{2}B}e^{\frac{1}{2}A}e^{B}e^{A}) can also be obtained in the following way:

\$ ./bch "expression=log(exp(1/6*B)*exp(1/2*A)*exp(2/3*B+1/72*[B,[A,B]]))*exp(1/2*A)*exp(1/6*B))"

+1/1*A+1/1*B+1/1*C+1/2*[[[A,B],B],B]+1/6*[A,[[A,B],B]]+1/24*[[[A,B],B],B]+1/12*[[[[A,B],B],B],B]+1/24*[[[[[A,B],B],B],B],B] \$

table_output [=0 or =1 depending on size of result]}

With table_output=0 the result is displayed as a linear combination of commutators. This is the default if the result consists of less than 200 terms. With table_output=1, on the other hand, the result is displayed in tabular form. Each row of the table corresponds to a
basis element of the respective Lie basis, where by default the index, the indices of the left
and the right factors, and the coefficient of the basis element is displayed. This information
uniquely determines the basis element, which follows from the fact that for all the bases we
consider (i.e., Lyndon, Hall, and right-normed Chibrikov) the factors $h', h''$ of basis elements
$h = [h', h'']$ of degree $\geq 2$ are basis elements themselves.

```bash
$ ./bch table_output=1
0 1 0 0 1/1
1 1 1 0 1/1
2 2 0 1 1/2
3 3 0 2 1/12
4 3 2 1 1/12
5 4 0 3 0/1
6 4 0 4 1/24
7 4 4 1 0/1
8 5 0 5 -1/720
9 5 0 6 1/180
10 5 3 2 1/360
11 5 0 7 1/180
12 5 2 4 1/120
13 5 7 1 -1/720
```

The displayed items can be customized by specifying the following parameters which should
be self-explanatory:

- **print_index [=1]**
- **print_degree [=1]**
- **print_multi_degree [=0]**
- **print_factors [=1]**
- **print_foliage [=0]**
- **print_basis_element [=0]**
- **print_coefficient [=1]**

```bash
$ ./bch table_output=1 print_basis_element=1 print_multi_degree=1
0 1 (1,0) 0 0 A 1/1
1 1 (0,1) 1 0 B 1/1
2 2 (1,1) 0 1 [A,B] 1/2
3 3 (2,1) 0 2 [A,[A,B]] 1/12
4 3 (1,2) 2 1 [[A,B],B] 1/12
5 4 (3,1) 0 3 [A,[A,[A,B]]] 0/1
6 4 (2,2) 0 4 [A,[[A,B],B]] 1/24
7 4 (1,3) 4 1 [[[A,B],B],B] 0/1
8 5 (4,1) 0 5 [A,[A,[A,[A,B]]]] -1/720
9 5 (3,2) 0 6 [A,[A,[[A,B],B]]] 1/180
10 5 (3,2) 3 2 [[A,[A,B]],[A,B]] 1/360
11 5 (2,3) 0 7 [A,[[[A,B],B],B]] 1/180
12 5 (2,3) 2 4 [[A,B],[[A,B],B]] 1/120
13 5 (1,4) 7 1 [[[A,B],[[A,B],B]],B] -1/720
```

Note that if the parameter **verbosity_level** is set to a value $\geq 1$ (see below), then a header
line for the table is printed.
With `verbosity_level` set to a value \( \geq 1 \) some useful information about the performance of the computation is printed, for example the common denominator and timings of the main steps of the computation as described in Section 2. Furthermore a table showing the number of the Lie basis elements and the number of nonzero coefficients of the Lie series is printed. For easy identification, each line of this output has the character `#` at the leading position.

With `verbosity_level` set to a value \( \geq 2 \) this output is not buffered but flushed immediately. This is only recommended if long running times of the program are expected.

```
$ ./bch N=20 verbosity_level=1 | head -n 60
#number of Lyndon words of length\(\leq\)20 over set of 2 letters: 111013
#initialize Lyndon words: time=0.0344573 sec
#expression=log(exp(A)*exp(B))
#denominator=1021818843434188800000
#compute Goldberg coefficients: time=0.0137833 sec
#compute coefficients of Lyndon words: time=0.0226991 sec
#convert to Lie series: time=0.158603 sec
#compute terms of degree 20: time=0.000395872 sec
#total time=0.216212 sec
# degree  dim  #nonzero  dim(cum.)  #nz(cum.)
#  1   2    2   2   2   2
#  2   1    1   3   3
#  3   2    2   5   5
#  4   3    1   8   6
#  5   6    6   14  12
#  6   9    5   23  17
#  7  18   18  41  35
#  8  30   17  71  52
#  9  56   55  127 107
# 10  99   55  226 162
# 11 186   186 412 348
# 12 335   185 747 533
# 13 630   630 1377 1163
# 14 1161  629 2538 1792
# 15 2182 2181 4720 3973
# 16 4080 2181 8800 6154
# 17 7710 7710 16510 13864
# 18 14532 7709 31042 21573
# 19 27594 27594 58636 49167
# 20 52377 27593 111013 76760
#
# multi-degree  dim  #nonzero
#  (1,19) 1   0
#  (2,18) 9   1
#  (3,17) 57  9
#  (4,16) 240 51
#  (5,15) 775 204
#  (6,14) 1932 612
#  (7,13) 3876 1428
#  (8,12) 6288 2652
#  (9,11) 8398 3978
# (10,10) 9225 4862
# (11,9) 8398 4862
# (12,8) 6288 3978
# (13,7) 3876 2652
```
5 Performance of the \texttt{bch} program

We document the performance of typical runs of the \texttt{bch} program on an ordinary personal computer. More specifically, the computer system was a a 3.0 GHz Intel Core i5-2320 processor with 4 CPU cores and 8 GB of RAM running an Ubuntu Linux operating system. The C compiler was \texttt{gcc} version 9.3.0. The memory usage was measured with \texttt{/usr/bin/time}. For the test runs of the program we used different maximum degrees $N$ and different bases. The results are as follows: (Here, dimension is the number of basis elements of degree $\leq N$ and \#nonzero is the number of coefficients $\neq 0$.)

Lyndon basis:

\begin{center}
\begin{tabular}{cccccc}
\hline
$N$ & dimension & \#nonzero & time & memory \\
\hline
18 & 31042 & 21573 & 0.04 sec & 4628 kB \\
20 & 111013 & 76760 & 0.22 sec & 10932 kB \\
22 & 401428 & 276474 & 2.13 sec & 33008 kB \\
24 & 1465020 & 1005917 & 54.43 sec & 115 MB \\
26 & 5387991 & 3690268 & 16 min 16 sec & 427 MB \\
28 & 19945394 & 13632278 & 3 h 57 min & 1.4 GB \\
30 & 74248451 & 50657857 & 54 h 46 min & 5.5 GB \\
\hline
\end{tabular}
\end{center}

Classical Hall basis:

\begin{center}
\begin{tabular}{cccccc}
\hline
$N$ & dimension & \#nonzero & time & memory \\
\hline
18 & 31042 & 30477 & 0.35 sec & 81248 kB \\
20 & 111013 & 109697 & 2.12 sec & 428 MB \\
22 & 401428 & 398313 & 20 min 53 sec & 2.6 GB \\
\hline
\end{tabular}
\end{center}

Right-normed Chibrikov basis:

\begin{center}
\begin{tabular}{cccccc}
\hline
$N$ & dimension & \#nonzero & time & memory \\
\hline
18 & 31042 & 21561 & 0.76 sec & 58300 kB \\
20 & 111013 & 76748 & 21.52 sec & 637 MB \\
22 & 401428 & 276463 & 21 min 53 sec & 6.9 GB \\
\hline
\end{tabular}
\end{center}

For the Lyndon basis, the maximum degree $N = 30$ that the program can in principle handle could actually be achieved. For the other bases, $N = 22$ was the limit due to memory
constraints (the computer system had 8 GB of RAM).

It is natural to compare the performance of our program with that of other implementa-
tions of algorithms for computing the BCH series up to terms of high degree. Ref. [2] is the
only recently published report of such an implementation that we know of. Results obtained
with this implementation are available at

http://www.ehu.eus/ccwmuura/bch.html.

It should be acknowledged that these results served us well in verifying the computations of
our \texttt{bch} program.

For the implementation of [2] in \textsc{Mathematica} a completely different and more sophis-
ticated approach than ours was used. This approach, which is based on a Lie algebraic
structure of labeled rooted trees, has the obvious disadvantage that it is not easily adapted
to arbitrary expressions of the form [3]. Thus, only the classical and the symmetric BCH
series are handled in [2].

In [2] it is reported that the computation of the BCH series up to terms of degree \( N = 20 \)
in the classical Hall basis required less than 15 min CPU time and 1.5 GB of memory on a
2.4 GHz Intel Core 2 Duo processor with 2 GB of RAM. For the computation in the Lyndon
basis it required 3.6 GB of memory, but no specific information about the CPU time is given,
only that it required more time than the computation in the classical Hall basis. This has
to be compared with our implementation, which requires only 0.22 sec CPU time and 11 MB
of RAM for the Lyndon basis, and 2.12 sec CPU time and 428 MB of RAM for the classical
Hall basis. These numbers speak for themselves, even if direct comparisons of different
implementations in different programming languages that run on different hardware must be
interpreted with caution.

6 Usage of the \texttt{libbch} library

The \texttt{main()} function of the executable \texttt{bch}, which is defined in the source file \texttt{bch.c}, essen-
tially does nothing else than processing the command line arguments and then calling the
appropriate functions from the \texttt{libbch} library, which perform the actual computations. In
the same way, these functions can also be called from a user-defined program, which has to
be linked to the shared library \texttt{libbch.so} for this purpose. Each such program should in-
clude the header file \texttt{bch.h}, which contains declarations for the functions and data structures
provided by the library. The source file \texttt{bch.c} for the executable \texttt{bch} can serve as a model
for the usage of the \texttt{libbch} library.

In the following we give an excerpt from the header file \texttt{bch.h} covering essential decla-
rations together with some necessary explanations. It has been tried to use names for the
functions and their arguments that are as self-explanatory as possible, and that are consis-
tent with the names of the corresponding parameters for the \texttt{bch} program as described in
Section 3.

6.1 128 bit integer type

Essentially all computations of the \texttt{libbch} library are carried out in pure integer arithmetic,
a large part of them in 128 bit integer arithmetic (the other part can be done using integers
with fewer bits). In \texttt{bch.h} the 128 bit integer type \texttt{INTEGER} is defined as an alias for the
built-in type \texttt{__int128_t}:

\begin{verbatim}
typedef __int128_t INTEGER;
\end{verbatim}
Although such 128 bit integers are available for most modern C compilers, the C standard library does not provide functions for converting such integers to strings, or print them on standard output. The libbch library provides such functions and also functions for converting and printing rational numbers, which are specified by their numerators and denominators of type INTEGER. Note that the rational numbers are reduced to lowest terms first.

```c
int str_INTEGER(char *str, INTEGER x);
int str_RATIONAL(char *str, INTEGER num, INTEGER den);
void print_INTEGER(INTEGER x);
void print_RATIONAL(INTEGER num, INTEGER den);
```

The functions `str_INTEGER` and `str_RATIONAL` write to a character string `str` which has to be long enough to contain the output including a terminating '\0'. They return the number of characters written (excluding the terminating '\0'). If `str` is `NULL`, then these functions return the number of characters that would have been written in case of a proper output string `str`.

### 6.2 Lie series

The following two functions compute respectively the classical BCH series (i.e., the Lie series for \(\log(e^A e^B)\)) and the symmetric BCH series (i.e., the Lie series for \(\log(e^{\frac{1}{2} A} e^B e^{\frac{1}{2} A})\)):

```c
lie_series_t* BCH(int maximum_degree, int basis);
lie_series_t* symBCH(int maximum_degree, int basis);
```

Here the argument `maximum_degree` specifies the maximum degree up to which terms of the Lie series shall be computed. The argument `basis` specifies the basis in which the resulting Lie series shall be represented, where the values 0,1,2 respectively correspond to the Lyndon basis, the right-normed Chibrikov basis, and the classical Hall basis. These functions return a pointer to a structure of type `lie_series_t` which contains the resulting Lie series. This structure should be considered as an opaque data type for which access to its data is provided by data access functions (see below).

```c
void set_verbosity_level(int verbosity_level);
```

If the verbosity level is set to a value \(\geq 1\) before calling the functions `BCH` or `symBCH`, then these function print some useful information about the performance of the computation to the standard output.

### 6.3 Data access functions

```c
int dimension(lie_series_t *LS);
int maximum_degree(lie_series_t *LS);
int number_of_generators(lie_series_t *LS);
INTEGER denominator(lie_series_t *LS);
INTEGER numerator_of_coefficient(lie_series_t *LS, int i);
int degree(lie_series_t *LS, int i);
int degree_of_generator(lie_series_t *LS, int i, uint8_t g);
int left_factor(lie_series_t *LS, int i);
int right_factor(lie_series_t *LS, int i);
int str_foliage(char *str, lie_series_t *LS, int i, char *generators);
int str_basis_element(char *str, lie_series_t *LS, int i, char *generators);
```
int str_coefficient(char *str, lie_series_t *LS, int i);
void print_foliage(lie_series_t *LS, int i, char *generators);
void print_basis_element(lie_series_t *LS, int i, char *generators);
void print_coefficient(lie_series_t *LS, int i);

Here, the argument LS is a pointer which was previously returned by the functions BCH or symBCH (or lie_series, see below). The argument i is the index of the basis element, for which the respective information is requested. The argument g in degree_of_generator specifies the (index of the) generator for which the degree in the basis element with index i is requested. The argument generators should be a string like "ABCDEFG" containing at position 0 the name for generator 0, at position 1 the name for generator 1, etc.

The functions str_foliage, str_basis_element, and str_coefficient write to a character string str which has to be long enough to contain the output including a terminating '\0'. They return the number of characters written (excluding the terminating '\0'). If str is NULL, then these functions return the number of characters that would have been written in case of a proper output string str.

The functions left_factor and right_factor respectively return the index of the left factor $h'$ and the index of the right factor $h''$ of the basis element $h = [h', h'']$ with index i.

### 6.4 Cleaning up

void free_lie_series(lie_series_t *LS);

### 6.5 User-defined expressions

Besides Lie series for $\log(e^A e^B)$ and $\log(e^{\frac{1}{2}A} e^{\frac{1}{2}B})$, also Lie series for arbitrary expressions of the form (3) can be computed. Such expressions are represented as binary expression trees, for the generation of which the libbch library provides the following functions:

- `expr_t* identity(void);`
- `expr_t* generator(uint8_t g);`
- `expr_t* sum(expr_t* arg1, expr_t* arg2);`
- `expr_t* difference(expr_t* arg1, expr_t* arg2);`
- `expr_t* product(expr_t* arg1, expr_t* arg2);`
- `expr_t* negation(expr_t* arg);`
- `expr_t* term(int numerator, int denominator, expr_t* arg);`
- `expr_t* exponential(expr_t* arg);`
- `expr_t* logarithm(expr_t* arg);`
- `expr_t* commutator(expr_t* arg1, expr_t* arg2);`

These functions return a pointer to a structure of type `expr_t` which represents a node of the expression tree and contains data, depending on the type of the node, like pointers to subexpressions, the index of a generator, or numerator and denominator of a rational coefficient. For example, the following code generates the expression $\log(e^{\frac{1}{2}B} e^{\frac{1}{2}A} e^{\frac{1}{2}B} + \frac{1}{1728}[B;[A,B]] e^{\frac{1}{2}B} e^{\frac{1}{2}A})$:

```c
expr_t *A = generator(0);
expr_t *B = generator(1);
expr_t* expression =
    logarithm(product(product(product(product(
        exponential(term(1, 6, B)), exponential(term(1, 2, A))),
        exponential(sum(term(2, 3, B),
            term(1, 72, commutator(B, commutator(A, B)))))),
        exponential(term(1, 2, A))),
    exponential(term(1, 6, B)));
```
Alternatively to the explicit construction of expression trees using the above functions, such expressions can also be obtained by parsing an input string for which the `libbch` library provides the following function:

```c
expr_t* parse(char *input, char *generators, int *number_of_generators);
```

Here, the argument `input` is the input string and `generators` is an array of characters of length at least the number of different generators (noncommuting variables) that occur in `input`. On output the array `generators` contains the symbols for these generators in alphabetical order, and the variable pointed to by `number_of_generators` contains the number of these generators. For example, the following code likewise generates the expression

\[ \log(e^{\frac{1}{6}B}e^{\frac{1}{2}A}e^{\frac{2}{3}B+\frac{1}{72}[B,[A,B]]}e^{\frac{1}{2}A}e^{\frac{1}{6}B}) \];

```c
char gens[2];
int num_gen;
expr_t* expression = parse("log(exp(1/6*B)*exp(1/2*A) *exp(2/3*B+1/72*[B,[A,B]]) *exp(1/2*A)*exp(1/6*B))", gens, &num_gen);
```

After executing this code the array `gens` contains the characters `A` and `B` at positions 0 and 1, respectively, and the variable `num_gen` has the value 2.

It is possible to construct more general expressions than those of the form (3), which cannot necessarily be represented as a Lie series, so that the result of the function `lie_series` described below would not necessarily make any sense. Therefore it is recommended to use the following function for checking such expressions whether they can actually be represented as Lie series:

```c
int is_lie_element(expr_t* expression);
```

Finally, for an expression obtained in one of the above ways, the following function computes the corresponding Lie series:

```c
lie_series_t* lie_series(int number_of_generators, expr_t* expression, int maximum_degree, int basis);
```

Regarding arguments and return value, this function is analogous to the functions `BCH` and `symBCH`, but with additional arguments `number_of_generators` and `expression`. Here, `number_of_generators` is the number of generators occurring in the expression (or more precisely the highest index plus 1 of all generators occurring in the expression). In the above examples it should have the value 2.

For cleaning up, the following function frees all memory that was allocated during the construction of all expressions since the program was started (or since the last time this function was called):

```c
void free_all_expressions(void);
```

Usually it is not necessary to call this function explicitly because the `libbch` library registers it with the standard library function `atexit` so that it is automatically called when the program ends.

**References**

[1] A. Arnal, F. Casas, and C. Chiralt, A Note on the Baker–Campbell–Hausdorff series in terms of right-nested commutators, *Mediterr. J. Math.* 18, 53 (2021).
[2] F. Casas and A. Murua, An efficient algorithm for computing the Baker–Campbell–Hausdorff series and some of its applications, *J. Math. Phys.* 50, 033513 (2009).

[3] E.S. Chibrikov, A right normed basis for free Lie algebras and Lyndon–Shirshov words, *Journal of Algebra* 302 (2006), 593–612.

[4] E. Eriksen, Properties of higher-order commutator products and the Baker–Campbell–Hausdorff formula, *J. Math. Phys.* 9 (1968), 790–796.

[5] GNU Bison, [https://www.gnu.org/software/bison](https://www.gnu.org/software/bison).

[6] K. Goldberg, The formal power series for \(\log e^xe^y\), *Duke Math. J.* 23 (1956), 13–21.

[7] H. Hofstätter, A relatively short self-contained proof of the Baker–Campbell–Hausdorff theorem, *Expositiones Mathematicae*, to appear.

[8] H. Hofstätter, Denominators of coefficients of the Baker–Campbell–Hausdorff series, available at [https://arxiv.org/abs/2010.03440](https://arxiv.org/abs/2010.03440).

[9] H. Hofstätter, Smallest common denominators for the homogeneous components of the Baker–Campbell–Hausdorff series, available at [https://arxiv.org/abs/2012.03818](https://arxiv.org/abs/2012.03818).

[10] H. Hofstätter, Fast computation of the Baker–Campbell–Hausdorff and similar series, in preparation.

[11] H. Hofstätter, W. Auzinger, and O. Koch, An Algorithm for Computing Coefficients of Words in Expressions Involving Exponentials and Its Application to the Construction of Exponential Integrators, In: M. England, W. Koepf, T. Sadykov, W. Seiler, and E. Vorozhtsov (eds), *Computer Algebra in Scientific Computing, CASC 2019*, Lecture Notes in Computer Science 11661, Springer, Cham, 2019.

[12] Klib, A standalone and lightweight C library, available at [https://github.com/attractivechaos/klib](https://github.com/attractivechaos/klib).

[13] C. Reutenauer, Free Lie Algebras, Oxford University Press, Oxford 1993.

[14] WebAssembly, [https://webassembly.org](https://webassembly.org).