The package for computing a quasi-isomorphism $\mathsf{Ger}_\infty \to \mathsf{Br}$

V. A. Dolgushev and G.E. Schneider

Abstract

In this short addendum to [2], we describe our package for computing MC sprouts in $\mathrm{Conv}(\mathsf{Ger}^\vee, \mathsf{Br})$.

1 Brief Outline

This package allows us to compute MC-sprouts in

$$\mathrm{Conv}^{\oplus}(\mathsf{Ger}^\vee, \mathsf{Br}) \;\cong\; \bigoplus_{n\ge 2} \big(\mathsf{Br}(n) \otimes \Lambda^{-2}\mathsf{Ger}(n)\big)_{S_n}. \tag{1.1}$$

It consists of six Python files *Bases.py, BTCirc.py, Conv.py, Ger.py, LinCombGraphs.py and TwBT.py* and several "storage" files such as *al5bfile, al5cfile, ...* This package requires Python 3.5 (or a later version) and the library *SymPy* [5].

The file **LinCombGraphs.py** contains various functions for working with linear combinations, permutations, and graphs. Linear combinations are represented as nested lists $[[c_1, T_1], [c_2, T_2], \dots]$, where $T_1, T_2, \dots$ are some Python objects and $c_1, c_2, \dots$ are coefficients. For example, the list $[[-1, T_1], [5, T_2], [-3, T_3]]$ represents the linear combination

$$-T_1 + 5T_2 - 3T_3\,.$$

The coefficients $c_1, c_2, \dots$ are either Python integers or SymPy integers, or SymPy rationals.

The file *LinCombGraphs.py* also defines the function *Solve* whose input is a SymPy matrix M viewed as the augmented matrix of a linear system. If the linear system is inconsistent, the function *Solve* returns the tuple $(\mathsf{False}, (), ())$. If the linear system is consistent, the function *Solve* returns a tuple $(\mathsf{True}, x_0, NullMtx)$, where x_0 is a solution of this system (presented as a SymPy matrix with one column) and $NullMtx$ is the SymPy matrix whose columns form a basis of the null space for the corresponding coefficient matrix.

The file **Ger.py** contains various functions for working with $\Lambda\mathsf{Lie}$-words and Ger-words[1] $\Lambda\mathsf{Lie}$-words are represented as tuples. For example, the tuple $(2, 3, 1)$ represents the $\Lambda\mathsf{Lie}$-word $\{a_2, \{a_3, a_1\}\}$. Ger-words are represented as lists of $\Lambda\mathsf{Lie}$-words. For example, the list $[(2, 3), (1,), (4, 6, 5)]$ represents the Ger-word

$$\{a_2, a_3\}a_1\{a_4, \{a_6, a_5\}\}.$$

Note that the $\Lambda\mathsf{Lie}$-word $\{\{a_1, a_2\}, \{a_3, a_4\}\}$ cannot be written as a tuple because it is not of the form $\{a_{i_1}, \{a_{i_2}, \{a_{i_3}, a_{i_4}\}\}\}$. Due to the Jacobi identity,

$$\{\{a_1, a_2\}, \{a_3, a_4\}\} = -\{a_1, \{a_2, \{a_3, a_4\}\}\} - \{a_2, \{a_1, \{a_3, a_4\}\}\}.$$

[1] Notational conventions for vectors of Ger and $\Lambda^{-2}\mathsf{Ger}$ are borrowed from [1, Sections 3.2.2 and 11].

So the $\Lambda\mathsf{Lie}$-word $\{\{a_1, a_2\}, \{a_3, a_4\}\}$ can be represented as the "linear combination" of $\Lambda\mathsf{Lie}$-words

$$[\,[-1,(1,2,3,4)]\,,\ [-1,(2,1,3,4)]\,]$$

or as the "linear combination" of Ger-words

$$[\,[-1,[(1,2,3,4)]]\,,\ [-1,[(2,1,3,4)]]\,].$$

The tuple $(i_1, i_2, \dots, i_n)$ representing a $\Lambda\mathsf{Lie}$-word is called **standard** if i_n is maximal among $i_1, i_2, \dots, i_n$. For example $(3,5,1,4)$ is not standard but $(3,2,4)$ is[2]. It is easy to see that every vector in $\Lambda\mathsf{Lie}(n)$ is a linear combination of $\Lambda\mathsf{Lie}$-words represented by standard tuples.

A list

$$[(i_{11},\dots,i_{1k_1}),(i_{21},\dots,i_{2k_2}),\dots,(i_{r1},\dots,i_{rk_r})] \tag{1.2}$$

representing a Ger-word is called **standard** if

- i_{tk_t} is the biggest element of the tuple $(i_{t1},\dots,i_{tk_t})$ for every t, and
- it is sorted according to the standard Python 3 order for numerical tuples.

It is known [1, Section 3.3.2] that Ger-words in $\mathsf{Ger}(n)$ corresponding to standard lists form a basis of $\mathsf{Ger}(n)$. The function *toStanGer* (in the file *Ger.py*) takes a list representing a Ger-word W and returns the corresponding linear combination of standard lists.

Another important function is $InsGG$. It computes an elementary insertion of Ger-words. For example, after executing the file *Ger.py*, the command

In [2]: InsGG([(1, 4), (5, 2, 3)], 2, [(1,), (2,)])

produces

Out [2]: [[-1, [(1, 5), (2, 4, 6), (3,)]], [-1, [(1, 5), (3,), (4, 2, 6)]], [-1, [(1, 5), (2, 4), (3, 6)]], [-1, [(1, 5), (2,), (3, 4, 6)]], [-1, [(1, 5), (2,), (4, 3, 6)]], [1, [(1, 5), (2, 6), (3, 4)]]]

The output in **Out [2]:** shows that[3]

$$\begin{aligned}\{a_1,a_4\}\{a_5,\{a_2,a_3\}\} \circ_2 a_1a_2 \;=\;& -\{a_1,a_5\}\{a_2,\{a_4,a_6\}\}a_3 - \{a_1,a_5\}a_3\{a_4,\{a_2,a_6\}\}\\ &-\{a_1,a_5\}\{a_2,a_4\}\{a_3,a_6\} - \{a_1,a_5\}a_2\{a_3,\{a_4,a_6\}\}\\ &-\{a_1,a_5\}a_2\{a_4,\{a_3,a_6\}\} + \{a_1,a_5\}\{a_2,a_6\}\{a_3,a_4\}.\end{aligned}$$

Remark 1.1 Note that the degree of the monomial

$$\{b_{i_{11}}, b_{i_{12}}, \dots b_{i_{1k_1}}\}\{b_{i_{21}}, b_{i_{22}}, \dots b_{i_{2k_2}}\}\dots\{b_{i_{r1}}, b_{i_{r2}}, \dots b_{i_{rk_r}}\} \;\in\; \Lambda^{-2}\mathsf{Ger}(n) \tag{1.3}$$

differs from the degree of the monomial

$$\{a_{i_{11}}, a_{i_{12}}, \dots a_{i_{1k_1}}\}\{a_{i_{21}}, a_{i_{22}}, \dots a_{i_{2k_2}}\}\dots\{a_{i_{r1}}, a_{i_{r2}}, \dots a_{i_{rk_r}}\} \;\in\; \mathsf{Ger}(n) \tag{1.4}$$

by an **even** integer. This allows us to use all the functions of *Ger.py* for working with vectors in the shifted operad $\Lambda^{-2}\mathsf{Ger}$. Thus the monomial (1.3) is represented by the same list (1.2) as the monomial (1.4).

[2]Note that the standard tuple $(3,2,4)$ and the non-standard tuple $(3,4,2)$ represent the same $\Lambda\mathsf{Lie}$-word $\{a_3,\{a_2,a_4\}\} = \{a_3,\{a_4,a_2\}\}$.

[3]Note that, in this example, the Ger-word $\{a_1,a_4\}\{a_5,\{a_2,a_3\}\}$ is represented by a non-standard list. The output of $InsGG$ is always a linear combinations of standard lists.

The files **BTCirc.py** and **TwBT.py** define various functions for working with vectors of the operads BT, $\mathrm{Tw}\mathsf{BT}$, and $\mathsf{Br} \subset \mathrm{Tw}\mathsf{BT}$ [4, Sections 7, 8, 9]. For a brace tree T with n labeled vertices and k neutral vertices, we represent labeled (resp. neutral) vertices as integers $1, 2, \dots, n$ (resp. length one tuples $(1,), (2,), \dots, (k,)$). Then T is represented as the list of non-root edges $[e_1, e_2, \dots]$ which appear in the order coming from the planar structure of T. The edge connecting vertex v_1 to vertex v_2 is represented as the list $[v_1, v_2]$, where v_1 is closer to the root than v_2. As we go along the list corresponding to a brace tree T, the tuples corresponding to neutral vertices must appear in standard order:

$$(1,), (2,), \dots (k-1,), (k,).$$

For example, the brace tree shown in figure 1.1 is represented by the list

$$[\ [2,(1,)],\ [(1,),(2,)],\ [(2,),3],\ [(2,),4],\ [(1,),1]\ ].$$

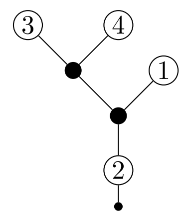


Fig. 1.1: An example of a brace tree

The function *hCirc* (in *BTCirc.py*) computes the elementary insertion of a brace tree into another brace tree and the function *hDiff* (in *BTCirc.py*) computes the image of the differential of a brace tree.

Example 1.1 After executing the file *BTCirc.py*, the command

In [2]: hCirc([[2,1]], 2, [[(1,), 1], [(1,), 2]])

produces

Out [2]: [[1, [[(1,), 1], [(1,), 2], [(1,), 3]]], [-1, [[(1,), 2], [2, 1], [(1,), 3]]], [-1, [[(1,), 2], [(1,), 1], [(1,), 3]]], [1, [[(1,), 2], [(1,), 3], [3, 1]]], [1, [[(1,), 2], [(1,), 3], [(1,), 1]]]]

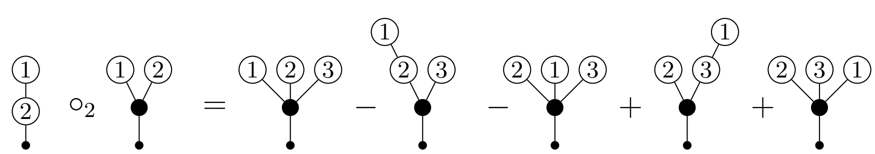
and the command

In [3]: hDiff([[(1,),1], [(1,),2], [(1,),3]])

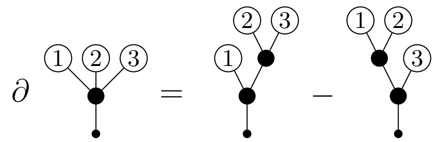
produces

Out [3]: [[1, [[(1,), 1], [(1,), (2,)], [(2,), 2], [(2,), 3]]], [-1, [[(1,), (2,)], [(2,), 1], [(2,), 2], [(1,), 3]]]]

The output in **Out [2]:** shows that



and the output in **Out [3]:** shows that



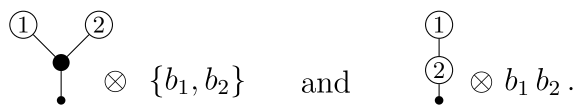
Conv.py is the main file of this package. In this file, we define the following functions for working with vectors of (1.1):

1. *dConv* (and its version *dConvlc*) which computes the differential of a vector in (1.1).
2. *pLie* (and its version *pLielc*) which computes the pre-Lie product of two vectors in (1.1).
3. *lieConv* (and its version *lieConvlc*) which computes the Lie bracket of two vectors in (1.1).

Tensor monomials of (1.1) are represented as length 2 tuples (T, W) where T is the list representing a brace tree and W is a list representing a $\Lambda^{-2}\mathsf{Ger}$-word. For example, the tuples

$$(\,[[(1,),1],[(1,),2]]\,,\ [(1,2)]\,) \quad \text{and} \quad (\,[[2,1]]\,,\ [(1,),(2,)]\,)$$

represent the tensor monomials



(1.5)

A tuple (T, W) representing a tensor monomial is called **standard** if

- the labeled vertices of T show up in the usual order and
- the list W (representing a $\Lambda^{-2}\mathsf{Ger}$-word) is standard.

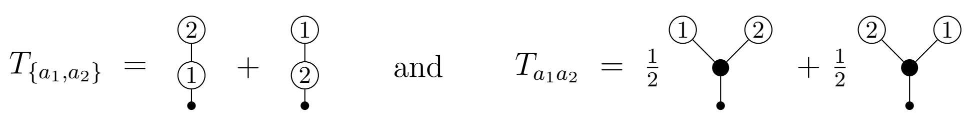
For example, the tuple $(\,[[2,1]]\,,\ [(1,),(2,)]\,)$ is non-standard because the labeled vertices of the brace tree corresponding to $[[2,1]]$ show up in the order $2,1$. On the other hand, the tuple $(\,[[(1,),1],[(1,),2]]\,,\ [(1,2)]\,)$ is standard. Indeed, the labeled vertices of the brace tree corresponding to $[[(1,),1],[(1,),2]]$ show up in the usual order $1,2$ and the list $[(1,2)]$ representing the $\Lambda^{-2}\mathsf{Ger}$-word $\{b_1, b_2\}$ is standard.

Since the S_n-module $\mathsf{Br}(n)$ is freely generated by brace trees whose labeled vertices show up in the usual order, tensor monomials corresponding to standard tuples form a basis of the space of coinvariants

$$\big(\mathsf{Br}(n) \otimes \Lambda^{-2}\mathsf{Ger}(n)\big)_{S_n}\,. \tag{1.6}$$

This is precisely the basis we use for our package.

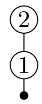
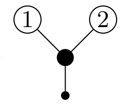
Example 1.2 Recall [3, Section 1] that the vectors



(1.7)

are cocycles in $\mathsf{Br}(2)$ whose cohomology classes generate the operad $H^{\bullet}(\mathsf{Br}) \cong \mathsf{Ger}$.

Thus the vector

$\alpha^{(1)} := 2$  $\otimes\, b_1 b_2 \;+$  $\otimes \{b_1, b_2\}$

is the first MC-sprout in (1.1). In our package, this vector is represented by the list

$$[\,[\,2, ([[1,2]], [(1,), (2,)])\,],\;[\,1, ([[(1,),1], [(1,),2]], [(1,2)])\,]\,]\,.$$

The file *Conv.py* also contains all the steps for finding the 2nd, 3rd and 4th MC-sprouts. Some of these steps are time consuming[4]. This is why this part of the program is commented. To find the 4th MC-sprout, we ran each step separately and “pickled” the results so that they can be used in further steps.

We should remark that the program *Conv.py* was actually looking for 240× MC-sprouts. Due to this small trick, most of the entries of augmented matrices of our linear systems are integers and the functions of SymPy run faster.

The commands after line 813 load the main result (i.e. 240× a 4-th MC-sprout):

- $al2$ is the list representing the linear combination of terms of (the lowest) arity 2.
- $al3$ is the list representing the linear combination of terms of arity 3.
- $al4$ is the list representing the linear combination of terms of arity 4.
- $al5$ is the list representing the linear combination of terms of arity 5.

In other words, the sum $al2+al3+al4+al5$ is the list representing 240× a 4-th MC sprout. 240× the first sprout is represented by the list:

$$al2 = [\,[120, ([[(1,),1], [(1,),2]], [(1,2)])], [240, ([[1,2]], [(1,), (2,)])]\,].$$

In order to test this result, you need to execute the file *Conv.py* and run the following commands

In [2]: test2 = dConvlc(al2)

In [3]: test3 = Simplify(mult(240, dConvlc(al3)) + pLielc(al2,al2))

In [4]: test4 = Simplify(mult(240, dConvlc(al4)) + lieConvlc(al2,al3))

In [5]: test5 = Simplify(mult(240, dConvlc(al5)) + lieConvlc(al2,al4) + pLielc(al3,al3))

After this, the command

In [6]: test2, test3, test4, test5

produces the tuple

Out [7]: ([ ], [ ], [ ], [ ])

[4]They require 2, 3 or more hours of computer time.

This confirms that, if α is the vector of (1.1) corresponding to the list $al2+al3+al4+al5$, then

$$240 \cdot \partial(\alpha) + \alpha \bullet \alpha$$

does not involve terms of arities ≤ 5. In other words,

$$\frac{1}{240}\,\alpha$$

is a 4-th MC-sprout in (1.1).

The commented lines below line 840 were used for additional testing of various functions in *Conv.py*. These additional tests were based on the identities

$$\partial^2 \;=\; 0, \qquad [\partial v, w] + (-1)^{deg(v)}[v, \partial w] = \partial([v, w])$$

and the Jacobi identity:

$$[[u, v], w] + (-1)^{deg(u)(deg(v)+deg(w))}[[v, w], u] + (-1)^{deg(w)(deg(u)+deg(v))}[[w, u], v] = 0.$$

All relationships between the Python files are shown in figure 1.2.

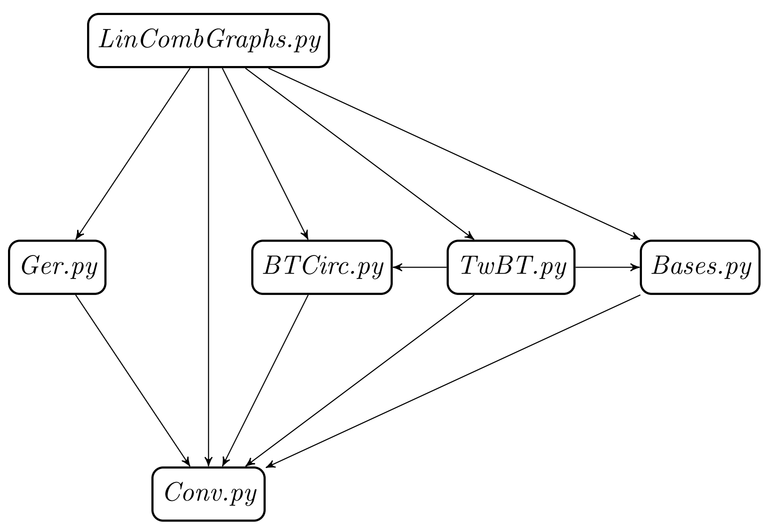


Fig. 1.2: The relationships between the Python files

Acknowledgements: The authors were partially supported by the NSF grant DMS-1501001. The authors are thankful to Sergey Plyasunov and Justin Y. Shi for showing them how to use the module *pickle.*

2 Main functions of *LinCombGraphs.py*

Here are the main functions for working with linear combinations:

- The function $Terms$ returns the tuple of "terms" of a linear combination represented as a nested list. For example, if

$$vec = [\ [2, ([[1,2]], [(1,), (2,)])\ ],\ \ [1, ([[(1,), 1], [(1,), 2]], [(1,2)])\ ]\ ]$$

then

$$Terms(vec) = (\ ([[1,2]], [(1,), (2,)]),\ ([[(1,), 1], [(1,), 2]], [(1,2)])\ ).$$

Note that there are no duplicates in the output of $Terms$.

- The function $Simplify$ simplifies a linear combination by combining similar terms and discarding summands of the form $0 \cdot term$. For example, if

$$vec = [\,[3, (1,)], [-1, (1,)], [5, (2,)], [-2, (2,)], [-3, (2,)], [7, (3,)]\,]$$

then

$$Simplify(vec) = [\,[2, (1,)], [7, (3,)]\,].$$

As you see, the blue summands cancel each other and the red summands are combined into the single summand $2 \cdot (1,)$.

- The function $mult(k, x)$ returns the result of multiplying the linear combination x by a scalar k. For example, $mult(3, [\,[2, (9,)], [-1, (9,)], [5, (7,)]])$ returns

$$[6, (9,)], [-3, (9,)], [15, (7,)]].$$

Note that $mult$ does NOT "simplify". As we said above, the scalar k is either an integer or a SymPy integer or a SymPy rational.

- Note that, for every pair of linear combinations v, w (represented via nested lists), $v+w$ (or better yet $Simplify(v+w)$) gives us the sum of these linear combinations.

- Many functions in this package are defined for basic vectors and then extended by linearity using the function $linExt$ and $bilinExt$. For example, if a function f operates as $(n \geq 2)$

$$f(\,(n,)\,)\ \ =\ \ [\,[1, (1, n-1)], [1, (2, n-2)], \ldots, [1, (n-1, 1)]\,],$$

then

$$linExt(f, [[3, (2,)], [-5, (3)]]) = [\,[3, (1,1)], [-5, (1,2)], [-5, (2,1)]\,].$$

The outputs of $linExt$ and $bilinExt$ **are simplified**.

- Let B be a tuple of basis elements and x be a simplified linear combination of elements of B. The output $Vect(x, B)$ of $Vect$ is the corresponding coordinate vector represented as the list. For example, if $B = ((1,), (2,), (3,), (4,))$ and $x = [[-5, (2,)], [7, (3,)]]$ then

$$Vect(x, B) = [0, -5, 7, 0].$$

- The function $toLC$ converts a coordinate vector v (with respect to a basis B) into the corresponding linear combination. We assume that B is a tuple and v is a list. For example,

$$toLC([-1, 0, 8], ((1,), (2,), (3,))) = [\,[-1, (1,)], [8, (3,)]\,].$$

Note that v and B must have **the same length.**

Some comments about permutations and graphs. In this package, a permutation

$$\begin{pmatrix} 1 & 2 & \dots & n-1 & n \\ i_1 & i_2 & \dots & i_{n-1} & i_n \end{pmatrix} \in S_n$$

is represented as the tuple $(i_1, i_2, \dots, i_{n-1}, i_n)$ and this should not be confused with the standard cycle notation. For example, the cycle $1 \mapsto 3 \mapsto 2 \mapsto 4 \mapsto 1$ is represented by the tuple $(3, 4, 2, 1)$.

Edges of a directed graph Γ with the set of vertices $\{1, 2, \dots, n\}$ are represented as lists of length 2. For example, an edge from vertex i to vertex j is represented by the list $[i, j]$. A directed graph Γ is represented by the list of its edges. For example, the directed graph shown in figure 2.1 is represented by the list

$$[[1, 2], [1, 2], [2, 1], [2, 3], [4, 2], [3, 3]].$$

Here we tacitly assume that we deal with graphs without vertices of valency 0.

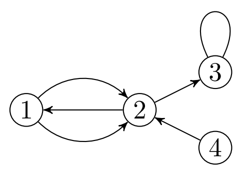


Fig. 2.1: An example of a directed graph

We also assume that all our trees are rooted and planar. All edges of trees are oriented "away from the root" and they are listed in the order coming from the planar structure of the tree.

Here are the main functions for working with permutations and graphs:

- $Perm(n)$ generates all permutation in S_n (as tuples). For example,

  $$Sn = tuple(s \quad for \quad s \quad in \quad Perm(n))$$

  gives us the tuple of all elements in S_n. For practical purposes, it is better to use $Perm(n)$ as the generator.

- The function inv computes the inverse of a permutation. For example $inv((3, 4, 2, 1))$ returns $(4, 3, 1, 2)$.

- The function $Vert$ returns the tuple of vertices (without repetitions) of a graph in the order they appear in the corresponding list. For example,

  $$Vert([[1, 2], [1, 2], [2, 1], [2, 3], [4, 2], [3, 3]])$$

  returns the tuple $(1, 2, 3, 4)$. $NumVert(G)$ gives the number of vertices of a graph G.

- $Val(G, i)$ gives the valency of vertex i in a graph G.

- $NumIn(T, i)$ returns the number of edges of a tree T which originate from vertex i. For example, $NumIn([[1, 2], [1, 3], [3, 4]], 1)$ returns 2, $NumIn([[1, 2], [1, 3], [3, 4]], 3)$ returns 1, and $NumIn([[1, 2], [1, 3], [3, 4]], 2)$ returns 0.

In *LinCombGraphs.py*, we also have 3 functions for working with SymPy matrices:

- The function *toCol* converts a numerical list (of length k) into the corresponding $k \times 1$ SymPy matrix.
- $Null(C)$ returns the basis of the null space of the SymPy matrix C. The output of $Null$ is a list. Each entry of this list is the list representing the corresponding vector. For example, if $C = Matrix([[1,1,1],[1,1,1],[1,1,1]])$ then $Null(C)$ is the nested list

  $$[[-1,1,0],[-1,0,1]].$$

  In other words, the null space of the matrix

  $$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

  is two-dimensional and it is spanned by the vectors

  $$\begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix} \qquad \text{and} \qquad \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}.$$

- The input of the function *Solve* is a SymPy matrix M. The output is a tuple

  $$(Consist, x_0, NullMtx).$$

  - If the system with the augmented matrix M is consistent then $Consist$ is True, x_0 is a solution of this system (represented as a SymPy column matrix), and $NullMtx$ is the SymPy matrix whose columns form a basis of the null space for the corresponding coefficient matrix.
  - If the system with the augmented matrix M is inconsistent, the function *Solve* returns (False, (), ()). Moreover, it raises the exception:

    Your system is inconsistent

  For example, if $M = Matrix([[3,5,-4,7],[-3,-2,4,-1],[6,1,-8,-4]])$ then $Solve(M)$ is the tuple

  $$(True,\ Matrix([[-1],[2],[0]]),\ Matrix([[4/3],[0],[1]])).$$

  In other words, the linear system with the augmented matrix

  $$\begin{pmatrix} 3 & 5 & -4 & 7 \\ -3 & -2 & 4 & -1 \\ 6 & 1 & -8 & -4 \end{pmatrix}$$

  is consistent; the vector

  $$\begin{pmatrix} -1 \\ 2 \\ 0 \end{pmatrix}$$

is a solution of this system; the null space of the corresponding coefficient matrix has dimension 1 and it is spanned by the vector

$$\begin{pmatrix} 4/3 \\ 0 \\ 1 \end{pmatrix}.$$

3 Main functions of *Ger.py*

Recall that the Lie bracket $\{\ ,\ \}$ of a Gerstenhaber algebra [4, Appendix A] is odd and the (commutative) multiplication is even. So, for every triple a, b, c of homogeneous elements of a Gerstenhaber algebra, we have

$$\mathrm{ad}_{\{a,b\}}(c) = -(-1)^{|a|}\mathrm{ad}_a\mathrm{ad}_b(c) - (-1)^{|b|+|a||b|}\mathrm{ad}_b\mathrm{ad}_a(c), \tag{3.1}$$

where $\mathrm{ad}_a := \{a,\ \}$ and $|a|, |b|$ are the degrees of a and b, respectively. In particular, if a is even, then

$$\mathrm{ad}_{\{a,b\}}(c) = -\mathrm{ad}_a\mathrm{ad}_b(c) - (-1)^{|b|}\mathrm{ad}_b\mathrm{ad}_a(c). \tag{3.2}$$

Identity (3.2) is used many times in lines 194–267 of *Ger.py*. This part of the code is the preparation for defining the functions $InsLLie$ and *toStandard*. The function $InsLLie$ has three inputs t, i, tt, where $t = (i_1, \dots, i_n)$ and $tt = (j_1, \dots, j_m)$ are tuples of positive integers without repetitions and i is an element of t. The output of $InsLLie$ is the list of lists of the form [*coefficient*, *tuple*] representing the vector of the free $\Lambda\mathsf{Lie}$-algebra

$$(-1)^{(m-1)(n-k-1)}\{a_{r_1}, \dots \{a_{r_{k-1}}\{L, \{a_{r_{k+1}}, \dots, a_{r_n}\}..\},$$

where

$$L = \{a_{j_1+i-1}, \{a_{j_2+i-1}, \dots \{a_{j_{m-1}+i-1}, a_{j_m+i-1}\}..\},$$

k is the unique index such that $i_k = i$, and

$$r_s = \begin{cases} i_s & \text{if} \quad i_s < i, \\ i_s + m - 1 & \text{if} \quad i_s > i. \end{cases}$$

For example, $InsLLie((4, 2, 5, 1), 2, (1, 2))$ returns

$$[[1, (5, 2, 3, 6, 1)], [1, (5, 3, 2, 6, 1)]]$$

which agrees with

$$-\{a_5, \{\{a_2, a_3\}, \{a_6, a_1\}\}\} = \{a_5, \{a_2, \{a_3, \{a_6, a_1\}\}\}\} + \{a_5, \{a_3, \{a_2, \{a_6, a_1\}\}\}\}.$$

If the integer i does not belong to the tuple t then $InsLLie(t, i, tt)$ returns an error message.

If tuples t and tt represent monomials $v \in \Lambda\mathsf{Lie}(n)$ and $\tilde{v} \in \Lambda\mathsf{Lie}(m)$, respectively, then $InsLLie(t, i, tt)$ returns the list representing the vector

$$v \circ_i \tilde{v} \in \Lambda\mathsf{Lie}(n + m - 1).$$

For example, $InsLLie((3, 2, 1), 2, (1, 2))$ returns

$$[[-1, (4, 2, 3, 1)], [-1, (4, 3, 2, 1)]]$$

which agrees with

$$\{a_3, \{a_2, a_1\}\} \circ_2 \{a_1, a_2\} \;=\; -\{a_4, \{a_2, \{a_3, a_1\}\}\} - \{a_4, \{a_3, \{a_2, a_1\}\}\}.$$

The function *toStandard* takes a tuple which represents a $\Lambda\mathsf{Lie}$ word and turns it into the linear combination of standard Ger word of the form $\{a_{i_1}, \{a_{i_2}, \dots\}..\}$. For example, $toStandard((4, 2, 3, 1))$ returns the list

$$[\,[1, [(2, 3, 1, 4)]], [1, [(2, 1, 3, 4)]], [-1, [(3, 1, 2, 4)]], [-1, [(1, 3, 2, 4)]]\,]$$

which agrees with

$$\{a_4, \{a_2, \{a_3, a_1\}\}\} \;=$$
$$\{a_2, \{a_3, \{a_1, a_4\}\}\} + \{a_2, \{a_1, \{a_3, a_4\}\}\} - \{a_3, \{a_1, \{a_2, a_4\}\}\} - \{a_1, \{a_3, \{a_2, a_4\}\}\}.$$

Note that the function *toStandard* **can** be applied to a standard tuple.

For a tuple $t = (i_1, i_2, \dots, i_n)$ and a list $L = [(j_{11}, \dots, j_{1k_1}), \dots, (j_{r1}, \dots, j_{rk_r})]$,

$$Ins1(t, L)$$

returns the list which represents the vector

$$(-1)^{|w|(n-2)}\{w, \{a_{i_2}, \{a_{i_3}, \dots \{a_{i_{n-1}}, a_{i_n}\}..\},$$

where

$$w = \{a_{j_{11}}, \dots, \{a_{j_{1k_1-1}}, a_{j_{1k_1}}\}..\} \;\dots\; \{a_{j_{r1}}, \dots, \{a_{j_{rk_r-1}}, a_{j_{rk_r}}\}..\}. \tag{3.3}$$

For example, $Ins1((2, 3, 1), [(5,), (3, 4)])$ returns

$$[\,[1, [(5, 3, 1), (3, 4)]],\; [1, [(3, 4, 3, 1), (5,)]],\; [1, [(4, 3, 3, 1), (5,)]]\,]$$

which agrees with

$$-\{a_5 \cdot \{a_3, a_4\}, \{a_3, a_1\}\} =$$
$$\{a_5, \{a_3, a_1\}\} \cdot \{a_3, a_4\} + \{a_3, \{a_4, \{a_3, a_1\}\}\} \cdot a_5 + \{a_4, \{a_3, \{a_3, a_1\}\}\} \cdot a_5\,.$$

Note that the output $Ins1(t, L)$ does not depend on the first entry $t[0]$ of the tuple t.

For a tuple $t = (i_1, i_2, \dots, i_n)$ (without repetitions), an integer i (in the tuple t), and a list

$$W = [(j_{11}, \dots, j_{1k_1}), \dots, (j_{r1}, \dots, j_{rk_r})],$$
$$InsLG(t, i, W)$$

returns the list which represents the vector

$$(-1)^{|w|(n-k-1)}\{a_{i_1}, \dots \{a_{i_{k-1}}, \{w, \{a_{i_{k+1}}, \dots \{a_{i_{n-1}}, a_{i_n}\}..\},$$

where k is the index for which $i_k = i$ and w is defined in (3.3). For example,

$$InsLG((2, 3, 1), 3, [(4,), (6, 5)])$$

returns

$$[\,[1, [(2, 4, 1), (6, 5)]], [-1, [(2, 6, 5), (4, 1)]], [-1, [(2, 6, 5, 1), (4,)]],$$
$$[-1, [(2, 4), (6, 5, 1)]], [-1, [(2, 5, 6, 1), (4,)]], [-1, [(2, 4), (5, 6, 1)]]\,]$$

which agrees with

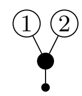
$$\{a_2,\{a_4{\cdot}\{a_6,a_5\},a_1\}\} = \{a_2,\{a_4,a_1\}\}{\cdot}\{a_6,a_5\} - \{a_2,\{a_6,a_5\}\}{\cdot}\{a_4,a_1\} - \{a_2,\{a_6,\{a_5,a_1\}\}\}{\cdot}a_4$$
$$-\{a_2,a_4\}\cdot\{a_6,\{a_5,a_1\}\} - \{a_2,\{a_5,\{a_6,a_1\}\}\}\cdot a_4 - \{a_2,a_4\}\cdot\{a_5,\{a_6,a_1\}\}.$$

If i does not belong to the tuple t, then $InsLG$ returns an error message. Note that "terms of" the output $InsLG(t,i,W)$ are not necessarily standard.

We use the function $InsLG$ to define the function $InsGG$ described above in Section 1.

4 Main functions of *BTCirc.py* and *TwBT.py*

We use two different ways to represent brace trees in this package. The first one is explained in Section 1 and we call it the *h-presentation* (h for "human"). For example, the brace tree


(4.1)

corresponding to the cup product has the h-presentation

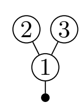
$$[[(1,),1],[(1,),2]].$$

The vertex labeled by the tuple $(1,)$ is the (only) neutral vertex of this brace tree.

To describe the second presentation (we call it *c-presentation*), we let T be a brace tree with n labeled vertices and k neutral vertices. To this T, we assign the new brace tree T' with $n+k$ labeled vertices and no neutral vertices at all. The brace tree T' is obtained from T following these steps:

- first, we shift all the labels of T up by k,
- second, we turn the neutral vertices into labeled ones by assigning the labels $1,2,\dots,k$ according to the total order coming from the planar structure.

For example, if T is the brace tree shown in (4.1) then

$T' =$ 

So the c-presentation of a brace tree T (with k neutral vertices) is the list of the form $[k,L]$ where L is the list corresponding to the brace tree T'. For example, the c-presentation of the brace tree in (4.1) corresponding to the cup product is the list

$$[\,1\,,\;[[1,2],[1,3]]\,].$$

Although the h-presentation makes the visualization easier, the implementation of the elementary insertions and the differential is more straightforward in c-presentation.

The module *TwBT.py* is used to convert the c-presentation of a brace tree to its h-presentation and vice versa. Thus,

- The input of $H2Comp$ is the h-presentation of a brace tree T and the output is the c-presentation of T.

- Conversely, the input of $Comp2H$ is the c-presentation of a brace tree T and the output is the h-presentation of T.
- $Comp2Hlc$ is the extension of $Comp2H$ to linear combinations.
- For an h-presented brace tree T with n labeled vertices, $getPermBr(T)$ is the permutation $\sigma \in S_n$ such that $T = \sigma(T_{can})$, where T_{can} is the unique brace tree corresponding to T in which the labeled vertices appear in the standard order coming from the planar structure. For example if $T = [[(1,),3],[(1,),1],[(1,),2],]$ then $getPermBr(T)$ is the tuple $(3,1,2)$.
- The function $ActBr(\, , \,)$ implements the left action of a permutation on an h-presented tree. For example, $ActBr((3,1,2),[[(1,),1],[(1,),2],[(1,),3]])$ returns
$$[[(1,),3],[(1,),1],[(1,),2]].$$
- Let T be a brace tree without neutral vertices and with the standard order of labels. Let r be a positive integer $<$ the number of vertices of T. Then $giveBr(r,T)$ generates all admissible[5] braces trees with r neutral vertices which can be obtained from T by replacing labeled vertices by neutral vertices. For example, if $T = [[1,2],[2,3]]$ then $giveBr(1,T)$ does not generate anything. On the other hand, if $T = [[1,2],[2,3],[2,4],[1,5]]$ then the command

  In [2]: for TT in giveBr(1,T):

  ...: print(TT)

  ...:

  returns

  [[(1,), 1], [1, 2], [1, 3], [(1,), 4]]

  [[1, (1,)], [(1,), 2], [(1,), 3], [1, 4]]

  The command

  In [3]: for TT in giveBr(2,T):

  ...: print(TT)

  ...:

  returns

  [[(1,), (2,)], [(2,), 1], [(2,), 2], [(1,), 3]]

In the module *BTCirc.py*, we define the functions $twCirc$ and $twDiff$ which implement the elementary insertion and the differential (in terms of the c-presentation) in $\mathrm{Tw}\mathsf{BT}$, respectively.

In lines 65–338, we define various auxiliary functions for working with brace trees. The most important auxiliary function in these lines are $prune(\, , \,)$ and $graft(\, , \, , \,)$:

[5] Recall that a brace tree is admissible if every neutral vertex has at least 2 children.

- For vertex i of a brace tree T (without neutral vertices), $prune(T,i)$ returns the list of branches originating from i in the usual order coming from the planar structure. If no edges originate from i then $prune(T,i)$ returns the empty list [ ]. For example, if $T = [[7,3],[3,6],[3,1],[1,4],[1,5],[7,2]]$, then $prune(T,3)$ returns (see the top part of figure 4.1)
$$[\ [[3,6]]\ ,\ [[3,1],[1,4],[1,5]]\ ]$$
and $prune(T,7)$ returns (see the bottom part of figure 4.1)
$$[\ [[7,3],[3,6],[3,1],[1,4],[1,5]]\ ,\ [[7,2]]\ ].$$

- The function $graft(\ ,\ ,\ )$ has 3 arguments: a brace tree T (without neutral vertices), a list L of branches, and the tuple sec of positions of T for attaching branches. For instance, the brace tree T shown in figure 4.2 has 13 such positions[6] and they are indicated in the figure by red numbers $1,2,\dots,13$. The length of sec must coincide with the length of L, sec may have repetitions, for every brach B of L the lowest vertex $B[0][0]$ of B must be univalent. The label of the lowest vertex of B is replaced by the label of the vertex to which the branch B is attached. For example, if $T = [[7,3],[3,6],[3,1],[1,4],[1,5],[7,2]]$ (the brace tree in figure 4.2) and
$$B1 = [[9,8]];\quad B2 = [[10,12],[12,11],[12,13]];\quad B3 = [[15,14],[14,16]];\quad B4 = [[17,18]]$$
then $graft(T,[B1,B2,B3,B4],(3,4,11,11))$ returns this list
$$[[7,3],[3,6],[6,8],[3,12],[12,11],[12,13],[3,1],[1,4],[1,5],[7,14],[14,16],[7,18],[7,2]]$$
and this process is illustrated in figure 4.3.

  Note that the function $graft$ deals with labeled planar trees whose set of labels is not necessarily $\{1,2,\dots,n\}$, where n is the number of non-root vertices.

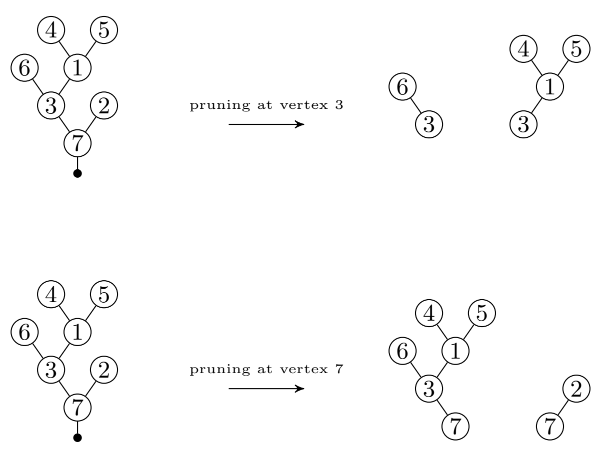


Fig. 4.1: Illustrations of the function *prune*

The function $twCirc$ implements the operadic insertion in the c-presentation. For example, if $T = [\,1\,,\,[[1,2],[1,3]]\,]$ (i.e. the c-presentation of the brace tree in (4.1)) and $TT =$

[6] It is not hard to see that a brace tree with e non-root edges has $2e+1$ positions for attaching branches.

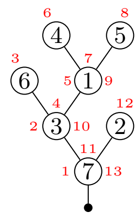


Fig. 4.2: This brace tree has 13 positions for attaching the branches

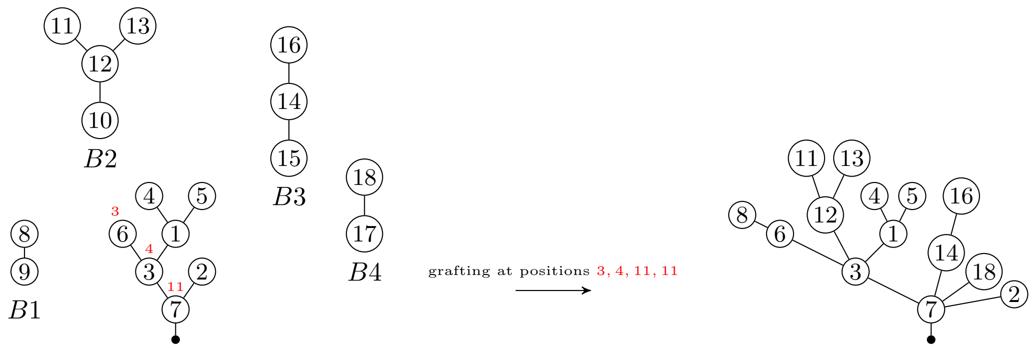


Fig. 4.3: An illustration of grafting

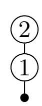


Fig. 4.4: This brace tree has the c-presentation $[\,0\,,\,[[1,2]]\,]$

$[\,0\,,\,[[1,2]]\,]$ (i.e. the c-presentation of the brace tree in figure 4.4) then $twCirc(T,1,TT)$ returns

$$[\,[-1,[1,[[1,2],[2,3],[1,4]]]]\,],$$

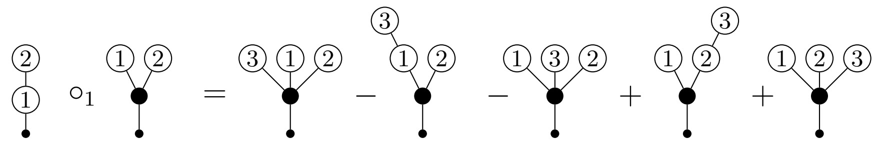
$twCirc(TT,1,T)$ returns

$$\begin{gathered}[\,[1,[1,[[1,4],[1,2],[1,3]]]],\\ [-1,[1,[[1,2],[2,4],[1,3]]]],\\ [-1,[1,[[1,2],[1,4],[1,3]]]],\\ [1,[1,[[1,2],[1,3],[3,4]]]],\\ [1,[1,[[1,2],[1,3],[1,4]]]]\,],\end{gathered} \tag{4.2}$$

and $twCirc(TT,2,T)$ returns

$$[\,[1,[1,[[2,1],[1,3],[1,4]]]]\,].$$

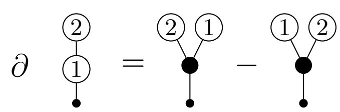
The output in (4.2) shows that



The function $twDiff$ implements the differential in the c-presentation. For example, if $TT=[\,0\,,\,[[1,2]]\,]$ (i.e. the c-presentation of the brace tree in figure 4.4) then $twDiff(TT)$ returns

$$[\,[1,[1,[[1,3],[1,2]]]],[-1,[1,[[1,2],[1,3]]]]\,]$$

which agrees with



For $T=[\,1\,,\,[[1,2],[1,3]]\,]$ (i.e. the c-presentation of the brace tree in (4.1)), $twDiff(T)$ returns the empty list $[\,]$. It agrees with the fact that the brace tree in (4.1) is a cocycle.

The function $hCirc$ (resp. $hDiff$) implements the operadic insertion for brace trees (resp. the differential) in the h-presentation. The functions $hCirc$ and $hDiff$ are obtained from $twCirc$ and $twDiff$ in the obvious way using the functions $H2Comp$ and $Comp2H$ from *TwBT.py*.

Finally, the function $hDifflc$ is the extension of $hDiff$ to linear combinations.

5 Main functions of *Conv.py*

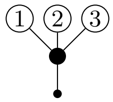
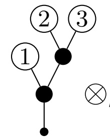
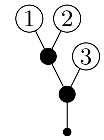
The main functions of *Conv.py* are *dConv*, *pLie*, *lieConv* and their extensions *dConvlc*, *pLielc*, *lieConvlc* to linear combinations. Since these function were already mentioned in Section 1, we will only give several examples.

If $v=([[(1,),1],[(1,),2],[(1,),3]],[(1,2,3)])$ then the $dConv(v)$ returns

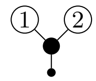
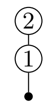
$$[\,[1,([[(1,),1],[(1,),(2,)],[(2,),2],[(2,),3]],[(1,2,3)])],$$

$$[-1,([[(1,),(2,)],[(2,),1],[(2,),2],[(1,),3]],[(1,2,3)])]\,].$$

This output shows that

∂  $\otimes_{S_3}\ \{b_1\{b_2,b_3\}\} \ = $  $\otimes_{S_3}\ \{b_1\{b_2,b_3\}\} \ - $  $\otimes_{S_3}\ \{b_1\{b_2,b_3\}\}.$

dConvlc is the extension of *dConv* to linear combination of basis vectors in (1.1) For example, the first sprout

$\frac{1}{2}$  $\otimes_{S_2}\ \{b_1,b_2\} \ + $  $\otimes_{S_2}\ b_1b_2$

is represented by the list

$$[\,[S(1)/2,([[(1,),1],[(1,),2]],[(1,2)])],[1,([[1,2]],[(1,),(2,)])]\,].$$

So

$$dConvlc\big(\,[\,[S(1)/2,([[(1,),1],[(1,),2]],[(1,2)])],[1,([[1,2]],[(1,),(2,)])]\,]\big)$$

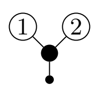
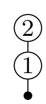
returns the empty list [ ].

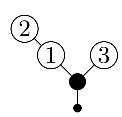
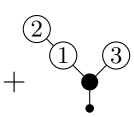
pLie(v,w) computes the pre-Lie bracket of basis vectors v and w in (1.1). For example, if $v=([[(1,),1],[(1,),2]],[(1,2)])$ and $w=([[1,2]],[(1,),(2,)])$, then *pLie*(v,w) returns

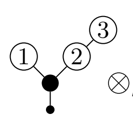
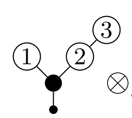
$$[\,[1,([[(1,),1],[1,2],[(1,),3]],[(1,3),(2,)])],[1,([[(1,),1],[1,2],[(1,),3]],[(1,),(2,3)])],$$

$$[-1,([[(1,),1],[(1,),2],[2,3]],[(1,2),(3,)])],[-1,([[(1,),1],[(1,),2],[2,3]],[(1,3),(2,)])]\,].$$

This agrees with

 $\otimes_{S_2}\ \{b_1,b_2\}\ \bullet$  $\otimes_{S_2}\ b_1\,b_2\ =$

 $\otimes_{S_3}\ \{b_1,b_3\}\,b_2\ \ +$  $\otimes_{S_3}\ b_1\,\{b_2,b_3\}$

$-$  $\otimes_{S_3}\ \{b_1,b_2\}\,b_3\ \ -$  $\otimes_{S_3}\ \{b_1,b_3\}\,b_2.$

pLielc(v_1,v_2) computes the pre-Lie bracket of two linear combinations of basis vectors. For example, if

$$v=[\,[1,([[(1,),1],[(1,),2]],[(1,2)])]\ ,\ \ [2,([[1,2]],[(1,),(2,)])]\,]$$

then *pLielc*(v,v) returns

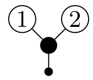
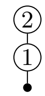
$$[\,[-1,([[(1,),(2,)],[(2,),1],[(2,),2],[(1,),3]],[(1,2,3)])],$$

$$[-1,([[(1,),(2,)],[(2,),1],[(2,),2],[(1,),3]],[(2,1,3)])],$$

$$[1,([[(1,),1],[(1,),(2,)],[(2,),2],[(2,),3]],[(1,2,3)])],$$

$$[2,([[(1,),1],[1,2],[(1,),3]],[(1,),(2,3)])],[-2,([[(1,),1],[(1,),2],[2,3]],[(1,3),(2,)])],$$
$$[2,([[(1,),1],[(1,),2],[(1,),3]],[(1,),(2,3)])],[-2,([[(1,),1],[(1,),2],[(1,),3]],[(1,3),(2,)])],$$
$$[2,([[(1,),1],[(1,),2],[(1,),3]],[(1,2),(3,)])],[2,([[1,(1,)],[(1,),2],[(1,),3]],[(1,),(2,3)])]\,]$$

which agrees with the computation of the pre-Lie bracket of

 $\otimes_{S_2} \{b_1, b_2\} \ + \ 2$  $\otimes_{S_2} b_1\, b_2$

with itself.

We should remark that the outputs of $pLie$ and $lieConv$ are not simplified, in general. For example, if $v = ([[(1,),1],[(1,),2]],[(1,2)])$ and $w = ([[1,2]],[(1,),(2,)])$, then $lieConv(v,w)$ returns the linear combination with 10 summands:

$$[\,[1,([[(1,),1],[1,2],[(1,),3]],[(1,3),(2,)])],[1,([[(1,),1],[1,2],[(1,),3]],[(1,),(2,3)])],$$
$$[-1,([[(1,),1],[(1,),2],[2,3]],[(1,2),(3,)])],[-1,([[(1,),1],[(1,),2],[2,3]],[(1,3),(2,)])],$$
$$[1,([[(1,),1],[(1,),2],[(1,),3]],[(1,),(2,3)])],[-1,([[(1,),1],[1,2],[(1,),3]],[(1,3),(2,)])],$$
$$[-1,([[(1,),1],[(1,),2],[(1,),3]],[(1,3),(2,)])],[1,([[(1,),1],[(1,),2],[2,3]],[(1,2),(3,)])],$$
$$[1,([[(1,),1],[(1,),2],[(1,),3]],[(1,2),(3,)])],[1,([[1,(1,)],[(1,),2],[(1,),3]],[(1,),(2,3)])]\,].$$

For the same basis vectors v and w, the command $Simplify(lieConv(v,w))$ returns a linear combination with 6 summands:

$$[\,[1,([[(1,),1],[1,2],[(1,),3]],[(1,),(2,3)])],[-1,([[(1,),1],[(1,),2],[2,3]],[(1,3),(2,)])],$$
$$[1,([[(1,),1],[(1,),2],[(1,),3]],[(1,),(2,3)])],[-1,([[(1,),1],[(1,),2],[(1,),3]],[(1,3),(2,)])],$$
$$[1,([[(1,),1],[(1,),2],[(1,),3]],[(1,2),(3,)])],[1,([[1,(1,)],[(1,),2],[(1,),3]],[(1,),(2,3)])]\,].$$

On the other hand, outputs of commands $pLielc$ and $lieConvlc$ are necessarily simplified because the function $Simplify$ is used in the definition of the function $bilinExt$.

5.1 Leaving the homogenous part for the dessert

Let us now describe a simple trick which we used often in the process of finding the 4-th MC sprout. This trick allows us to split a large linear system into two somewhat smaller linear systems. Then we can express the solution set of the original system in terms of the solution sets of these smaller systems.

Every linear system can be split into two linear systems:

$$A_1\vec{x} = \vec{r}, \tag{5.1}$$

$$A_2\vec{x} = \vec{0}, \tag{5.2}$$

where $\vec{r}$ is a vector whose all components are non-zero.

Let $\vec{x}_0$ be a solution of (5.1) and C be the matrix whose columns form a basis of the null space of A_1. Furthermore, let $\vec{y}_0$ be a solution of the system

$$A_2C\vec{y} = -A_2\vec{x}_0 \tag{5.3}$$

and C_1 be a matrix whose columns form a basis of the null space of $A_2 C$.

Then the vector

$$\vec{x}_0 + C\vec{y}_0$$

is a solution of the original linear system

$$A_1\vec{x} = \vec{r}, \qquad A_2\vec{x} = \vec{0}$$

and columns of the matrix CC_1 form a basis of the subspace of vectors $\vec{x}$ satisfying

$$A_1\vec{x} = \vec{0} \quad \text{and} \quad A_2\vec{x} = \vec{0}.$$

In this documentation, we call (5.1) (resp. (5.3)) the first (resp. the second) layer of the original linear system.

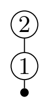
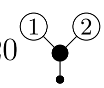
5.2 Solving the linear systems for terms in arity 5

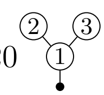
Let

$$\alpha_2^\bullet + \alpha_2^\circ + \alpha_3^\bullet + \alpha_3^\circ + \alpha_4^\bullet + \alpha_4^\circ \tag{5.4}$$

be 240× a 3rd MC-sprout found in *Conv.py* between lines 267 and 388, where $\alpha_m^\bullet$ (resp. α_m°) is the sum of terms of arity m with exactly one neutral vertex (resp. zero neutral vertices).

For example (see lines 278, 280, and 313 in *Conv.py*),

$\alpha_2^\circ \;=\; 240$  $\otimes_{S_2} b_1 b_2\,,$ $\qquad$ $\alpha_2^\bullet \;=\; 120$  $\otimes_{S_2} \{b_1, b_2\},$

$\alpha_3^\circ \;=\; 120$  $\otimes_{S_2} b_1\{b_2, b_3\}.$

Remark 5.1 Note that the 3rd MC-sprout (5.4) cannot be extended to a 4th one. In other words, the sum

$$-[\alpha_2^\bullet, \alpha_4^\bullet] - \alpha_3^\bullet \bullet \alpha_3^\bullet - [\alpha_2^\bullet, \alpha_4^\circ] - [\alpha_2^\circ, \alpha_4^\bullet] - [\alpha_3^\bullet, \alpha_3^\circ]$$

does not belong to the image of ∂

$$\partial\Big(\big(\mathsf{Br}(5) \otimes \Lambda^{-2}\mathsf{Ger}(5)\big)_{S_5} \Big).$$

We modify (5.4) later in *Conv.py* by adding an appropriate vector from the subspace

$$\big(\mathsf{Br}(4) \otimes \Lambda^{-2}\mathsf{Ger}(4)\big)_{S_4} \;\cap\; \ker(\partial). \tag{5.5}$$

Let us go over the process of finding vectors[7]

$$\alpha_5^\bullet \;\in\; \big(\mathsf{Br}(5) \otimes \Lambda^{-2}\mathsf{Ger}(5)\big)_{S_5}, \qquad \gamma_4^\bullet \;\in\; \big(\mathsf{Br}(4) \otimes \Lambda^{-2}\mathsf{Ger}(4)\big)_{S_4} \;\cap\; \ker(\partial)$$

[7]Every term in $\alpha_5^\bullet$ and $\gamma_4^\bullet$ has exactly one neutral vertex.

for which

$$240 \cdot \partial \alpha_5^\bullet + [\alpha_2^\bullet, \gamma_4^\bullet] = -[\alpha_2^\bullet, \alpha_4^\bullet] - \alpha_3^\bullet \bullet \alpha_3^\bullet\,. \tag{5.6}$$

The augmented matrix $LAug$ (as a nested list) of the linear system corresponding to (5.6) is found using lines 437–473. It is "pickled" in LAugfile. Using the two commands

$$M = Matrix(LAug); \quad M = M.T$$

one can convert the nested list $LAug$ into the corresponding SymPy matrix. Then the command $M.shape$ returns the tuple $(2016, 1376)$. In other words, the augmented matrix of the linear system corresponding to (5.6) has the size 2016×1376.

In lines 476–503, we form the first layer of the linear system corresponding to (5.6), find a particular solution for the first layer and find the matrix whose columns form a basis of the null space of the coefficient matrix of the first layer. (This took approximately 2 hours of computer time). This particular solution is "pickled" in $Xfile$ and this matrix is "pickled" in $MNullfile$.

In lines 505–536, we form the second layer, find a particular solution ("pickled" in $XXfile$) and find the matrix ("pickled" in $MMNullfile$) whose columns form a basis of the null space of the coefficient matrix from the second layer.

In lines 539–551, we use the solution sets coming from these two layers to get a particular solution ("pickled" in $YYfile$) for the linear system corresponding to (5.6) and a SymPy matrix ("pickled" in $Nullbfile$) whose columns form a basis of the null space of the coefficient matrix of the linear system corresponding to (5.6). In lines 553–571, we test results from lines 539–551.

In lines 577–605, we convert the particular solution obtained in lines 539–551 to the actual vectors $\alpha_5^\bullet$ (this is $al5b$ in *Conv.py*) and $\gamma_4^\bullet$ (this is $al4More$ in *Conv.py*) which satisfy (5.6). These vectors are "pickled" in $al5bfile$ and $al4bMorefile$, respectively. They were tested in lines 607-609.

In lines 612–800, we proceed in the similar manner and find vectors[8]

$$\alpha_5^\circ \;\in\; \big(\mathsf{Br}(5) \otimes \Lambda^{-2}\mathsf{Ger}(5)\big)_{S_5}, \qquad \gamma_4^\circ \;\in\; \big(\mathsf{Br}(4) \otimes \Lambda^{-2}\mathsf{Ger}(4)\big)_{S_4} \;\cap\; \ker(\partial)$$

which satisfy the equation

$$240 \cdot \partial \alpha_5^\circ + [\alpha_2^\bullet, \gamma_4^\circ] = -[\alpha_2^\bullet, \alpha_4^\circ] - [\alpha_2^\circ, \alpha_4^\bullet + \gamma_4^\bullet] - [\alpha_3^\circ, \alpha_3^\bullet]. \tag{5.7}$$

In *Conv.py*, vectors α_5° and γ_4° are represented by the lists $al5c$ ("pickled" in $al5cfile$) and $al4cMore$ ("pickled" in $al4cMorefile$), respectively.

The resulting vectors α_5° and γ_4° are tested in lines 803–806.

Thus the sum

$$\alpha_2^\bullet + \alpha_2^\circ + \alpha_3^\bullet + \alpha_3^\circ + (\alpha_4^\bullet + \gamma_4^\bullet) + (\alpha_4^\circ + \gamma_4^\circ) + \alpha_5^\bullet + \alpha_5^\circ$$

is a 4-th MC-sprout in (1.1). Therefore, due to [2, Corollary 2.16], the MC-sprout

$$\alpha_2^\bullet + \alpha_2^\circ + \alpha_3^\bullet + \alpha_3^\circ + (\alpha_4^\bullet + \gamma_4^\bullet) + (\alpha_4^\circ + \gamma_4^\circ)$$

is a truncation of a genuine MC element of (1.1) (defined over rationals).

[8] Every term in α_5° and γ_4° has zero neutral vertices.

Remark 5.2 A simple test shows that, if $[\alpha_2^\bullet, \gamma] = 0$ for

$$\gamma \in \big(\mathsf{Br}(4) \otimes \Lambda^{-2}\mathsf{Ger}(4)\big)_{S_4} \cap \ker(\partial)$$

and all terms of γ have exactly one neutral vertex, then $\gamma = 0$. It is this observation, which allows us to use equation (5.7) instead of the more general equation

$$240 \cdot \partial\alpha_5^\circ + [\alpha_2^\bullet, \gamma_4^\circ] + [\alpha_2^\circ, \gamma] = -[\alpha_2^\bullet, \alpha_4^\circ] - [\alpha_2^\circ, \alpha_4^\bullet + \gamma_4^\bullet] - [\alpha_3^\circ, \alpha_3^\bullet]$$

with the unknowns α_5°, γ_4°, and

$$\gamma \in \big(\mathsf{Br}(4) \otimes \Lambda^{-2}\mathsf{Ger}(4)\big)_{S_4} \cap \ker(\partial) \cap \ker([\alpha_2^\bullet, \;]),$$

where we assume that all terms of γ have exactly one neutral vertex.

6 A few remarks about *Bases.py*

The file Bases.py plays an auxiliary role. This file was used to form bases for several vector spaces:

- The function $saveBr9()$ was used to form and store the list $Br9$ of length 9. For $1 \le n \le 9$, $Br9[n-1]$ is the list of all admissible[9] brace trees of arity n with exactly 1 neutral vertex. The labeled vertices of all such trees appear in the standard order (coming from the planar structure). The list $Br9$ can be "unpickled" using the command $loadBr9()$.

  For example, the sequence of lines in the console:

  In [2]: Br9 = loadBr9()

  In [3]: len(Br9[2])

  Out[3]: 4

  In [4]: Br9[2]

  Out [4]:
  [[[(1,), 1], [(1,), 2], [(1,), 3]],
  [[(1,), 1], [1, 2], [(1,), 3]],
  [[(1,), 1], [(1,), 2], [2, 3]],
  [[1, (1,)], [(1,), 2], [(1,), 3]]]

  shows that there are exactly 4 admissible brace trees of arity 3 with exactly one neutral vertex and with the standard order of labeled vertices. These brace trees are drawn in figure 6.1.

[9] Recall that a brace tree is admissible if every neutral vertex has at least 2 children.

- The function *saveConvCirc*() was used to form and then store the list *ConvCirc* of length 6. For every $1 \le n \le 6$, $ConvCirc[n-1]$ is the basis of degree 1 elements[10] in $\mathsf{BT}(n) \otimes_{S_n} \Lambda^{-2}\mathsf{Ger}(n)$. For example, the command $ConvCirc = loadConvCirc()$ loads the list *ConvCirc* from the storage file. Then the command $ConvCirc[2]$ returns this list with 6 entries

$$[\,([[1,2],[1,3]],[(1,2),(3,)]),([[1,2],[2,3]],[(1,2),(3,)]),$$
$$([[1,2],[1,3]],[(1,3),(2,)]),([[1,2],[2,3]],[(1,3),(2,)]),$$
$$([[1,2],[1,3]],[(1,),(2,3)]),([[1,2],[2,3]],[(1,),(2,3)])\,].$$

  The basis vectors from this list are shown in figure 6.2.

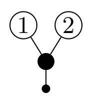
- The function *saveConvBul*() was used to form and store the list *ConvBul* of length 6. For every $1 \le n \le 6$, $ConvBul[n-1]$ is the basis of degree 1 elements in

$$\mathsf{Br}(n) \otimes_{S_n} \Lambda^{-1}\mathsf{Lie}(n).$$

  For example, the command $ConvBul = loadConvBul()$ loads the list *ConvBul* from the storage file. Then the command $ConvBul[1]$ returns this list with 1 element:

$$[\,([[(1,),1],[(1,),2]],[(1,2)])\,].$$

  This means that the subspace of degree 1 elements of $\mathsf{Br}(2) \otimes_{S_2} \Lambda^{-1}\mathsf{Lie}(2)$ is spanned by the single vector



$$\otimes_{S_2} \{b_1, b_2\}.$$

- *genGer*(n) returns the list of all standard Ger-words in $\mathsf{Ger}(n)$. For example, the command *genGer*(3) returns this list with 6 elements:

$$[\,[(1,2,3)],\ [(2,1,3)],\ [(1,2),(3,)],\ [(1,3),(2,)],\ [(1,),(2,3)],\ [(1,),(2,),(3,)]\,].$$

Fig. 6.1: The brace trees in the list $Br9[2]$

[10]For the definition of the operad BT, we refer the reader to [4, Section 7].

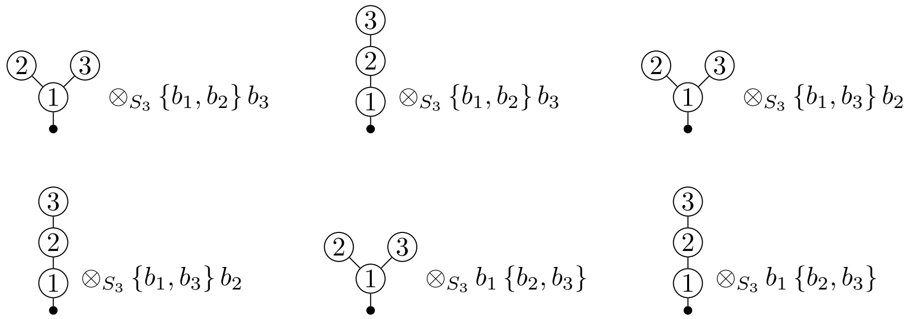


Fig. 6.2: The basis vectors in the list $ConvCirc[2]$

References

[1] V.A. Dolgushev and C.L. Rogers, Notes on algebraic operads, graph complexes, and Willwacher's construction, *Mathematical aspects of quantization,* 25–145, Contemp. Math., **583**, Amer. Math. Soc., Providence, RI, 2012.

[2] V.A. Dolgushev and G.E. Schneider, When can a formality quasi-isomorphism over rationals be constructed recursively? arXiv:1610.04879.

[3] V. A. Dolgushev and T. H. Willwacher, A Direct Computation of the Cohomology of the Braces Operad, Forum Math. **29**, 2 (2017) 465–488; arXiv:1411.1685.

[4] V. A. Dolgushev and T. H. Willwacher, Operadic twisting – with an application to Deligne's conjecture, J. Pure Appl. Algebra **219**, 5 (2015) 1349–1428.

[5] The library SymPy, http://www.sympy.org/en/index.html