5-2011

On the homology of automorphism groups of free groups.

Jonathan Nathan Gray
gray@math.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Jonathan Nathan Gray entitled "On the homology of automorphism groups of free groups.." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Mathematics.

James Conant, Major Professor

We have read this dissertation and recommend its acceptance:

Robert Daverman, Morwen Thistlethwaite, Michael Berry

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a dissertation written by Jonathan Nathan Gray entitled "On the homology of automorphism groups of free groups." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Mathematics.

James Conant, Major Professor

We have read this dissertation
and recommend its acceptance:

Robert Daverman

Morwen Thistlethwaite

Michael Berry

Accepted for the Council:

Carolyn R. Hodges
Vice Provost and
Dean of the Graduate School

(Original signatures are on file with official student records.)

On the homology of automorphism groups of free groups

A Dissertation
Presented for the
Doctor of Philosophy
Degree
The University of Tennessee, Knoxville

Jonathan Nathan Gray
May 2011

Acknowledgments

First and foremost, I must acknowledge my advisor, Dr. James Conant. Throughout my studies, Dr. Conant has been encouraging and supportive not only in my academic pursuits, but has also been a thoughtful and kind friend. Judging by my captivation and interest in my research topic, I can only conclude that Dr. Conant may have a supernatural ability to determine ones passions in mathematics.

Next, I would like to thank various members of the math department that have helped me along the way. Dr. Thistlethwaite is one of the most charismatic and impassioned instructors I have had the privilege to learn under; I can best describe his love for mathematics as infectious and inspiring. Dr. Dobbs taught me much of the algebra that I know. His lectures on category theory and homological algebra have proved their utility in this dissertation. Much credit is also due to Pam Armentrout. She surely wins the award of most helpful person in the department.

I would also like to thank my committee members: Dr. Michael Berry, Dr. Robert Daverman, and Dr. Morwen Thistlethwaite for devoting their time to reading my dissertation and attending my defense.

Abstract

Following the work of Conant and Vogtmann on determining the homology of the group of outer automorphisms of a free group, a new nontrivial class in the rational homology of Outer space is established for the free group of rank eight. The methods started in [8] are heavily exploited and used to create a new graph complex called the space of good chord diagrams. This complex carries with it significant computational advantages in determining possible nontrivial homology classes.

Next, we create a basepointed version of the Lie operad and explore some of it properties. In particular, we prove a Kontsevich-type theorem that relates the Lie homology of a particular space to the cohomology of the group of automorphisms of the free group.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

There is an intimate interplay between group theory and topology. Through the desire to describe this connection, some of the most beautiful mathematics of the last century was born: category theory in Cartan and Eilenberg's "General Theory of Natural Equivalences," the axiomization of homology by Eilenberg and Steenrod, and homological algebra culminating in Grothendieck's "Sur quelques points d-algébre homologique" among many others. While homology had its beginnings in the work of Poincaré, its abstraction came to fruition during a meeting between Noether and Vietoris. At the time, invariants for a space were given numerically: Noether suggested they should be given as *groups.*

One could argue that the most basic group is the free group on n generators; indeed, every finitely generated group is a quotient of a free group. The crux of free groups is that they are mysterious beasts and they, at times, completely defy our intuition (e.g., F_n has subgroups of every countable rank when $n > 1$). Our goal in this paper is to analyze the homology of two groups related to the free group F_n: its group of automorphisms $Aut(F_n)$ and its group of outer automorphisms $Out(F_n) = Aut(F_n)/Inn(F_n)$. In particular, we establish two results. First, in Chapter 3, we establish a new nontrivial class in the cohomology of $Out(F_n)$ by using the methods of Conant and Vogtmann [8]:

Theorem. $H^{12}(Out(F_8)) \neq 0$

Then, in Chapter 4, we give a Kontsevich-type theorem relating the homology of a certain infinite Lie module with that of the cohomology of $Aut(F_n)$:

Theorem. $H_k(\ell_\infty; p_1\ell_\infty)^{\mathrm{co}\,\ell_\infty} \cong H_k(\mathfrak{sp}(\infty)) \oplus \bigoplus_{n\geq 2} H^{2n-1-k}(Aut(F_n); \mathbb{Q})$

There are many avenues by which one can study the free group. One can work from a purely algebraic standpoint, one can work geometrically, or a combination thereof. One method by which we can approach the free group geometrically, is part of a bigger framework started by Thurston and Gromov. In general, one can create a space where the group of interest acts in a nice and predictable manner. In Thurston and Gromov's case, they considered the mapping class group with its action on Teichmüller space. We will focus on the groups $G = Out(F_n)$ or $Aut(F_n)$ and the contractible space associated to each on which G acts with finite stabilizers. It follows that the rational homology of G is the rational homology of the space quotiented by the action of G: $H^\bullet(G; \mathbb{Q}) \cong H^\bullet(X/G; \mathbb{Q})$.

The spaces on which $Out(F_n)$ or $Aut(F_n)$ act nicely are called Outer space and Auter space, respectively. In [9], Culler and Vogtmann created Outer space, translating Thurston and Gromov's analysis of marked Riemannian surfaces in Teichmüller space to the marked graphs of Outer space. Informally, Outer space is a contractible (almost) simplicial complex on which $Out(F_n)$ acts with finite stabilizers. There is a deformation retract of Outer space onto its so-called spine K_n. This spine is also contractible, the point-stabilizers in K_n are finite, and, in addition, the spine can be viewed as a complex of cubes of graphs. The latter simplification helps with the computation of the cohomology groups $H^\bullet$ because of its structured and combinatorial definition. Much of the first results in the study of the cohomology of Outer space were achieved in this manner, see [16, 17], [11].

In [21, 22], Kontsevich developed a completely new approach to determining the cohomology of $Out(F_n)$ by way of graph homology. In particular, Kontsevich showed

Theorem. *(Kontsevich)* $PH_k(\ell_\infty) \cong H_k(\mathfrak{sp}(\infty)) \oplus \bigoplus_{n\geq 2} H^{2n-2-k}(Out(F_n))$

by relating the Lie homology of the infinite-dimensional Lie algebra ℓ_∞ to that of the cohomology of $Out(F_n)$. The middle-man in this equivalence is a graph complex called the Lie graph complex; its graph homology also calculates the Lie homology of ℓ_∞.

The Lie algebra ℓ_∞ in the theorem can be viewed as the vector space arising from a generating set of *symplectospiders*. We can picture a symplectospider as an object with two or more legs, an abdomen which consists of a binary tree, and at the end of each leg is a "shoe" which is an element of a symplectic vector space. There is a mating operation which gives rise to a Lie bracket and in turn makes this space of symplectospiders into a Lie algebra.

Loosely speaking, from a collection of spiders we can form a graph by gluing their feet together in a reasonable manner. The space spanned by graphs which consist of a particular number of spiders-body components defines a chain group and it is this chain group which served as Kontsevich's graph homology middle-man.

The Lie homology of the Lie algebra ℓ_∞ carries a Hopf algebra structure and so the prefix P seen in the statement refers to primitives in the homology. These primitives in the Hopf algebra $H_\bullet(\ell_\infty)$ correspond to the "spider-graphs" which are connected (as in consist of a single component). If we dig deeper into the graph complex and consider the polygonal subcomplex (connected graphs where all the spiders-components have two legs), then it can be shown that the graph homology of this subcomplex coincides with the homology of the Lie algebra $\mathfrak{sp}(\infty)$. The resulting quotient of the graph complex by the polygonal subcomplex calculates the rational cohomology of $Out(F_n)$ via a second middle-man, the *forested graph complex*. See [11], [25], or [7, 8] for a thorough treatment of this topic.

Drawing from properties of the mapping class groups and the work of Kontsevich, Morita [31] defined a trace map that he conjectures gives rise to nontrivial cohomology classes:

Conjecture. *(Morita)* $H^{4k-4}(Out(F_{2k}); \mathbb{Q}) \neq 0$.

Answers in the affirmative (for some values of k) have been reached for this conjecture by Hatcher and Vogtmann [17], Ohashi [32], and Conant and Vogtmann [8].

Kontsevich's Lie algebra ℓ_∞ was originally viewed as a space of derivations on a free Lie algebra which kill a special element ω. It was this definition of ℓ_∞ that Morita used to create his trace map. It was later shown in [8] that Morita's trace has a purely combinatorial interpretation and gives rise to a cocycle in the forested graph complex. In the case of the forested graph complex, Morita's trace is called the *graphical trace map* τ.

Since the graphical trace τ is a cocycle, we may consider the subspace $\partial(\ker\tau)$ of the forested graph complex and the quotient of [a graded piece in] the complex by this subspace. This has the instant advantage that our problem of showing the nonvanishing of the cohomology groups has changed instead to attempting to overdetermine the quotient space via relations in $\partial(\ker\tau)$ and hence show the kernel coincides with the generating set. This is the approach by which Conant and Vogtmann in [8] established the nontriviality of the first two Morita cocycles and consequently answered Morita's conjecture in two cases.

We comment that a student of Morita, R. Ohashi has determined the nontriviality of the cohomology spaces in a different manner. In [32], Ohashi performs a computer calculation which shows that in the cases of $k = 1$, 2, and 3 the cohomology is singly generated – whether the generator for each case corresponds to a Morita cocycle was not determined in the paper. Together with the work of Conant and Vogtmann in [8], it can be said that the single generator for the first two cases does correspond to the Morita cocycle. Whether the Morita cocycle is the *sole* generator for cohomology is an intriguing question, as well.

Although it has not been mentioned, Auter space shares many of the same properties of Outer space (e.g., existence of a spine, an associated cubical complex, and most importantly, it can be treated homologically in the same manner as Outer space), but also carries the additional structure of being basepointed. The stability range of $Aut(F_n)$ is better understood than that of $Out(F_n)$. In particular, Hatcher and Vogtmann [17] showed $H_{k-1}(Aut(F_n)) \cong H_{k-1}(Aut(F_{n+1}))$ for $n \geq 5k/4$. Furthermore, it was shown in [18] that $H_k(Aut(F_n)) \cong H_k(Out(F_n))$ for $n \geq 2k+2$. In the thesis of Gerlits [11], the first known instance ($k = 7$, $n = 5$) where the homology of $Out(F_n)$ and $Aut(F_n)$ differ was determined via a computation and thus gives a lower bound for the stability range. It is also worth noting that in the range $k \leq 7$, (and any value of n) the only known case where $H_k(Aut(F_n))$ is nonzero is $n = k = 4$ and, in light of the work of Gerlits, the nonvanishing homology class in $H_4(Aut(F_4))$ carries over to $H_4(Out(F_4))$.

If we go back to our discussion of Outer space and now allow for the addition of a single distinguished basepoint; the result, suitably extended, is Auter space as given in [17]. In [7], Conant and Vogtmann generalized Kontsevich's result to the case of a *cyclic operad*. In an effort to carry the results from the Out case over to the Aut case, troubles arise. While one can give a multiple-basepointed structure to the Lie operad to make it into a cyclic operad, it is not the case that the resulting space of symplectospiders is graded in the way we desire: the mating of two basepointed symplectospiders will result in multiple basepoints while the Aut situation allows for one. The consequence of this is that we no longer get a Lie algebra of symplectospiders, but now get a Lie module of symplectospiders. Similar compromises must be made through the resulting construction. For instance, we have a Hopf module structure on the homology of the basepointed symplectospiders and graph complex of basepointed graphs. In Chapter 4 we will show that, despite the weakened algebraic structures that arise from a basepoint, there remains a Kontsevich-type relation

Theorem. $H_k(\ell_\infty; p_1\ell_\infty)^{\mathrm{co}\,\ell_\infty} \cong H_k(\mathfrak{sp}(\infty)) \oplus \bigoplus_{n\geq 2} H^{2n-1-k}(Aut(F_n); \mathbb{Q})$

where the superscript on the lefthand side consists of the ℓ_∞-coinvariants in the comodule $H_k(\ell_\infty; p_1\ell_\infty)$.

1.2 Organization of the paper

Let us outline what is to follow in the coming pages. In Chapter 2, we provide much of the details regarding our above discussion. We begin by completely defining Outer space and Auter space and elaborate on the hinted-at spine with its cubical complex structure. After, we recall some notions from category theory in preparation to define an operad. Once an operad is defined, we make rigorous our description of a "symplectospider" and "spider-graph." We then give a treatment of Lie algebra homology with coefficients, a crucial generalization that will be needed in Chapter 4. Finally, we define the forested graph complex and completely detail Morita's trace map and its translation into the language of forest graphs.

In Chapter 3 we utilize our foundation set up in Chapter 2 regarding a programme by which to determine the nonvanishing of the Morita cocycle. First, we exhibit a generating set known to Conant and Vogtmann in [8] called the space of chord diagrams. Using a relation resulting from Morita's trace map, we reduce the dimension of the space of chord diagrams so as to be more computationally tractable. At the end of the chapter, we describe the algorithm and programs used to confirm Morita's conjecture for the case $k = 4$.

Lastly, in Chapter 4 we expand on the Kontsevich-type isomorphism for $Aut(F_n)$. A basepointed version of the Lie operad will be defined. Using this operad, we will give a suitable notion of a basepointed spider and a basepointed Lie graph and show their connection to the cohomology of $Aut(F_n)$.

Chapter 2

Background

In this section we provide the necessary background to understand the questions in this dissertation. Due to the scope of the material, complete explanations are not always provided and where brevity is chosen the reader is referred to a resource for a more detailed exposition.

2.1 $Aut(F_n)$, $Out(F_n)$, Auter space, and Outer space

For a positive integer n, let F_n denote the free group on n generators. The group of automorphisms of F_n, its [normal] subgroup of inner automorphisms, and the resulting quotient of the latter two will be denoted by $Aut(F_n), Inn(F_n)$, and $Out(F_n)$, respectively.

By a *graph*, we mean a one-dimensional CW-complex G where the 0-cells $V(G)$ are called *vertices* and the 1-cells $E(G)$ are called *edges*. A contractible graph is called a *tree* and a union of trees is a *forest*. We say an edge of a graph is *separating* if its removal causes the graph to be disconnected. To an edge e we associate the two *half-edges* $H(e)$ that comprise it and to a vertex v we associate the collection of half-edges $H(v)$ incident to it. For a vertex v, its *valence* $|v|$ is the cardinality of $H(v)$ and so, in particular, a loop which is an edge with its endpoints identified has only one vertex and contributes two to the valence of the vertex.

To each free group $F_n = \langle x_1, \cdots, x_n \rangle$ we associate a *rose* with n petals $R_n = \vee_n S^1$ via the fundamental group π_1 by stating that each generator x_i of F_n corresponds to an edge of R_n. Thus $\sigma \in Aut(F_n)$ is represented by a homotopy equivalence of the rose that sends the edge loop of x_i to the edge-path loop σx_i. A *marked metric graph* (g, G) is composed of the ingredients:

1. A finite connected graph G with all vertices at least trivalent
2. Each edge of G is assigned a length so that the sum over all edge lengths of G is 1.
3. A *marking* g which is a homotopy equivalence $R_n \xrightarrow{g} G$

By (2) we can view the graph G as a metric space with the path metric. We say the marked graphs (g, G) and (g', G') are equivalent if there is an isometry $G \xrightarrow{f} G'$ such that $g' = f \circ g$ (up to homotopy).

Equivalence classes of finite marked metric graphs comprise the points of our space of interest, Outer space X_n. Formally,

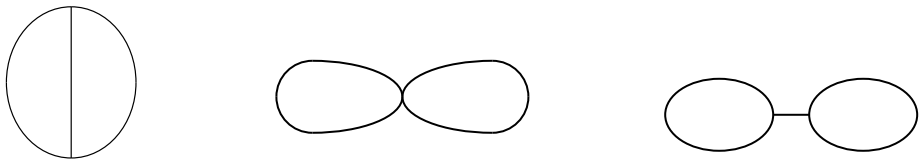
Definition 2.1.1. The space X_n called *Outer space* is the collection of marked metric graphs (g, G) which are finite and connected with $\pi_1 G \cong F_n$. The group $Out(F_n)$ acts (on the right) on Outer space by changing the marking, i.e., for $\sigma \in Out(F_n)$ we take a representative $R_n \xrightarrow{\rho} R_n$ for σ and define $(g, G)\sigma = (g \circ \rho, G)$.

Note that in the marked graph (g, G) we can vary the length of any edge of G as long as we keep the sum of all of the edges equal to one. Moreover, if the graph G has a subforest, we can collapse the subforest to a obtain new graph which is still homotopy equivalent to R_n. Thus we can decompose Outer space into a disjoint union of open simplices where a graph G with e edges corresponds to an open simplex of dimension $e - 1$ in X_n. It is clear that if G has no subforest, then there are no edges to collapse and so these graphs form the bottom dimension.

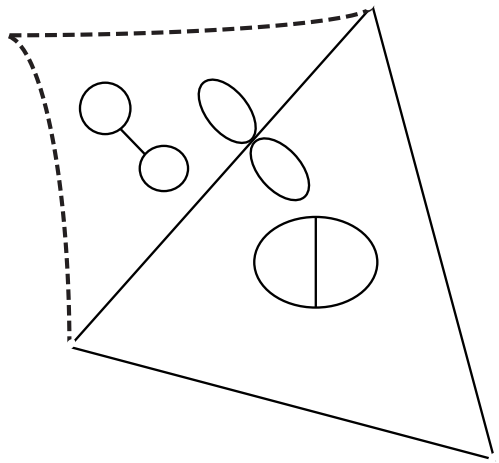
If we collapse all separating edges of marked graphs uniformly, the result equivariantly deformation retracts onto a subspace of X_n called *reduced Outer space*, denoted Y_n. We define the *spine* K_n of Outer space to be the geometric realization of the posets of the open simplices of Y_n. One can further retract Outer space onto its spine and this has the distinct advantages that (1) K_n is a simplicial complex (X_n fails this as it is missing 0-cells, etc.), (2) the spine is contractible, and (3) the quotient of X_n by $Out(F_n)$ is compact with finite point stabilizers. More to the point (see VII.2, exercise 2 of [5]),

Theorem 2.1.2. $H_\bullet(Out(F_n); \mathbb{Q}) \cong H_\bullet(X_n/Out(F_n); \mathbb{Q})$

Let us consider the simplest case, F_2. In Figure 2.1, we have depicted the three rank-two



(a) G_1 (b) G_2 (c) G_3



(d) A snippet of Outer space

Figure 2.1: Ingredients of Outer space in dimension two

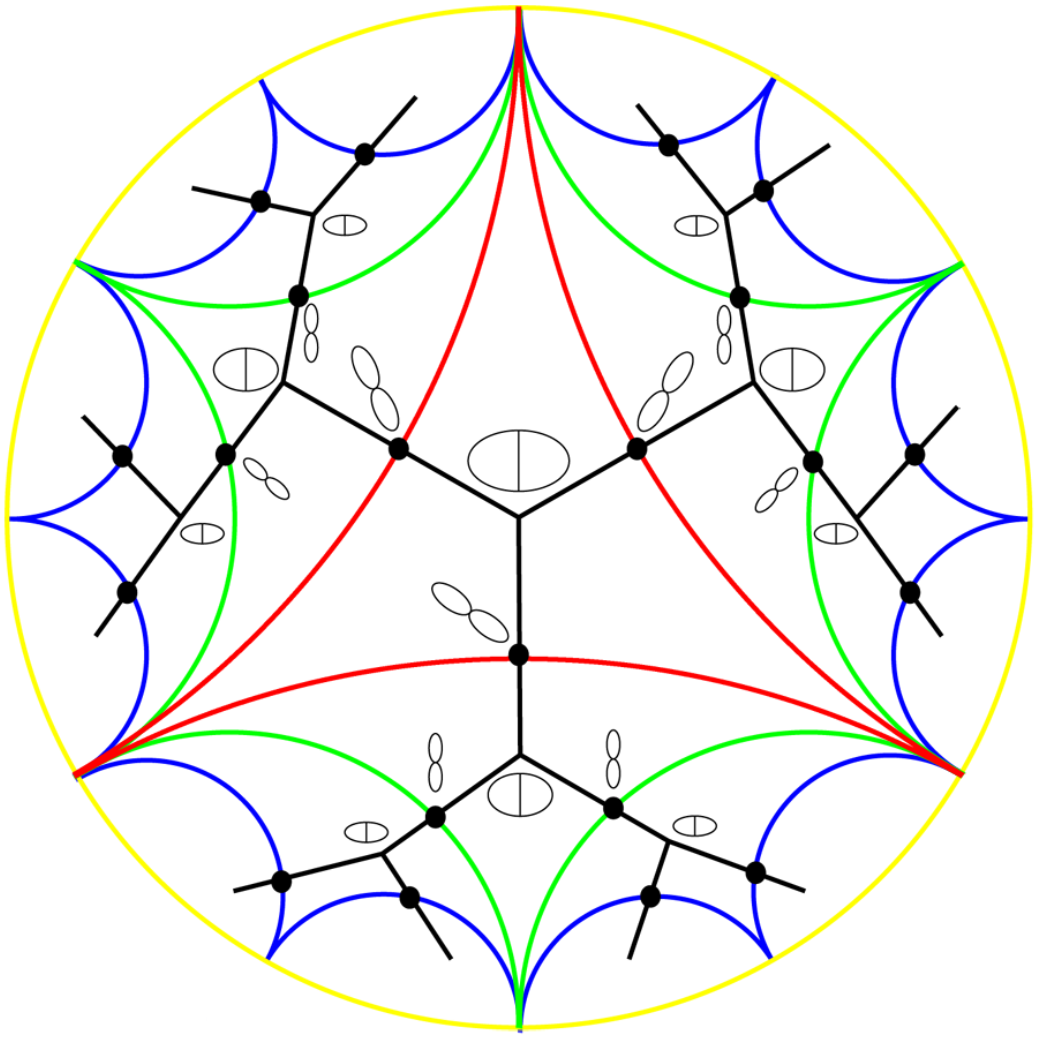


Figure 2.2: Reduced Outer space with its spine for $n = 2$

graphs (up to graph isomorphism). Note that the first graph G_1 has three edges and so corresponds to a 2-simplex. If we collapse any of the edges of G_1, the result is the 2-rose, G_2. As G_2 has two edges, it corresponds to a 1-simplex which "decorates" a face of the 2-simplex of G_1. As there are no subforests of G_2 to collapse, there are no 0-simplices. It can be shown that this (almost) simplicial structure is the ideal triangulation of the hyperbolic plane. Let us now see how G_3 fits into this picture. Since G_3 has three edges of which only one can be collapsed, it is represented by a 2-simplex with two edges missing. We visualize these additional faces as fins protruding from the hyperbolic plane. In this case, Y_2 (Figure 2.2) is the ideal triangulation without the fins and the spine is the 2-3 tree with vertices given by graphs of the form of G_1 (the trivalent vertices) and G_2 (the bivalent vertices).

Let us modify the construction of Outer space. If we now require our graphs to be basepointed by some distinguished vertex $*$ and correspondingly basepoint the n-rose, then a marking of such a graph is a marking in the familiar sense with the extra condition that the marking preserves basepoints. We remark that the basepoint does not necessarily have to occur at one of the original vertices of the graph and hence may sit on an edge and have

valence 2. Furthermore, the equivalence relation on marked graphs now is required to be a relation where the isometry is basepoint-preserving. Similarly, there is a space A_n on which $Aut(F_n)$ acts with finite point stabilizers, see [14], [16], or [17].

Let us focus on the space A_n. Following the construction of Outer space (while making the necessary adjustments so as to include the basepoint), we deformation retract onto the subspace of graphs without a separating edge. This space $\mathbb{A}_n$ is known as Auter space and is invariant under the action of $Aut(F_n)$, see [14]. Thus a point of Auter space $\mathbb{A}_n$ is an equivalence class of basepointed marked graphs $(g, G, *)$ without any separating edges.

As described, $\mathbb{A}_n$ is not a simplicial complex (being a union of simplices, it is missing some faces as in the *Out* case). We describe the *spine* $S\mathbb{A}_n$ of Auter space. A vertex of $S\mathbb{A}_n$ is an open simplex of $\mathbb{A}_n$ and a pair $(G, \Phi_1 \subset \cdots \subset \Phi_{k+1})$ determines a k-simplex of $S\mathbb{A}_n$ where G is a basepointed marked graph and the subforests of G are partially ordered by inclusion. Similar to the Out case, $Aut(F_n)$ acts with finite point stabilizers on the simply connected $S\mathbb{A}_n$ and so we can compute the rational homology of the group $Aut(F_n)$ by considering instead the space $Q_n = S\mathbb{A}_n/Aut(F_n)$.

It will be useful to describe $S\mathbb{A}_n$ as a cubical complex. Let $(g, G, \Phi, *)$ denote a marked graph together with a k-edged subforest Φ of G and basepoint $*$. We may order the k edges of Φ in $k!$ ways and so this gives rise to $k!$ simplices in the spine $S\mathbb{A}_n$. We call the union of these simplices a *cube* and denote the cube by brackets rather than parentheses: $[g, G, \Phi, *]$.

There are two operations δ_C, δ_R which we can place on an edge $e \in \Phi$ which we will write $(\delta_C)|_e$ and $(\delta_R)|_e$. The operation δ_C contracts an edge e of the forest and the operation δ_R removes an edge e from the forest, i.e., given an edge e, $(\delta_C)|_e[g, G, \Phi, *] = [g/e, G/e, \Phi/e, *]$ and $(\delta_R)|_e[g, G, \Phi, *] = [g, G, \Phi - e, *]$. In the cubical complex, the codimension-one faces of $[g, G, \Phi, *]$ correspond to either $(\delta_C)|_e[g, G, \Phi, *]$ or $(\delta_R)|_e[g, G, \Phi, *]$, see Figure 2.3 or 4.4.

Due to the existence of a distinguished basepoint, we may filter Auter space by degree. The degree of a basepointed graph $(G, *)$ is given by $deg(G, *) = \sum_v (|v| - 2)$ where $|v|$ denotes the valence of the vertex v and the sum is over all the vertices aside from the basepoint. Our graphs do not include univalent vertices (thus for any vertex v, $|v| \geq 2$) and so the summand given can be interpreted as all of the "surplus" valency at the vertex (or as Hatcher- and Vogtmann [16] call it, the "multiplicity"). It is shown in [16] by an argument involving edge-collapses and the Euler characteristic that the degree of a basepointed graph of rank n can be equivalently calculated as $2n - |*|$.

Let us describe the filtration from degree that takes place on the spine and its interpretation in the cubical complex. We let $S\mathbb{A}_{n,k}$ be spanned by graphs of $S\mathbb{A}_n$ with degree at most k and we let $Q_{n,k}$ be the quotient $S\mathbb{A}_{n,k}$ by $Aut(F_n)$. It is a result of Hatcher and Vogtmann [16] that $S\mathbb{A}_{n,k}$ is $(k-1)$-connected and we have $H_i(Q_{n,k}; \mathbb{Q}) \cong H_i(Aut(F_n); \mathbb{Q})$ for $k > i$. By the properties of degree, we note from [17] that the maximal cubes of $S\mathbb{A}_{n,k}$ correspond to basepointed graphs $[G, \Phi, *]$ such that

1. Φ is a maximal tree with k edges,
2. $deg(G, *) = k$, and
3. All the vertices of G (aside from $*$) are trivalent.

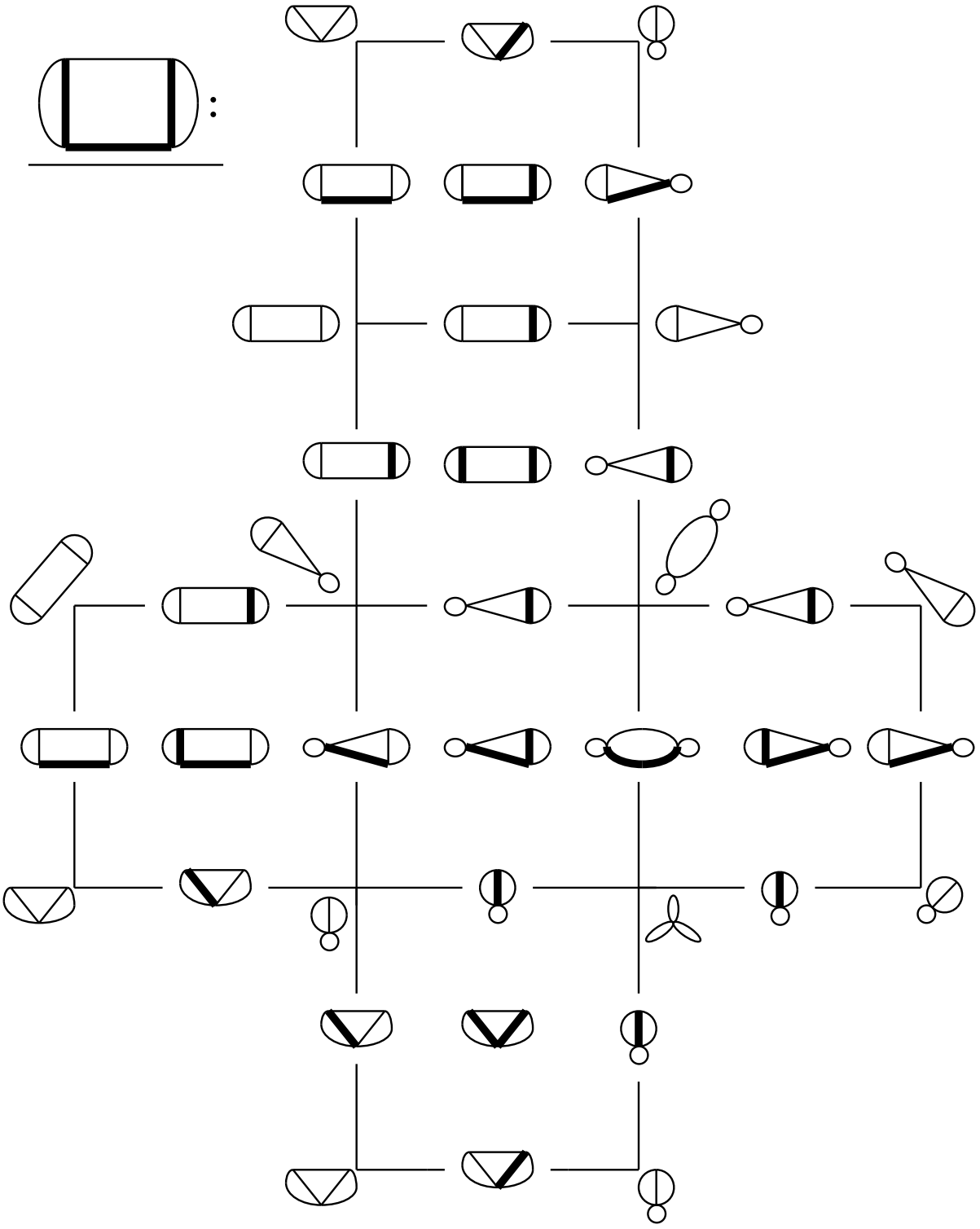


Figure 2.3: A cube in the 3-spine corresponding to the graph in the upper left

2.2 Executive Summary

The reader may be wondering how our discussion of Outer space and Auter space may relate to the theory of Lie algebras and graph homology. As an aid, we provide a map of the landscape.

We begin by considering the "creatures" A_1 and A_2 pictured in Figure 2.4(a) and the operation $\circ$ on A_1 and A_2 illustrated in Figure 2.4(b). We think of these creatures as "spiders" with labeled legs. The $\circ$ operation can be roughly seen as an identification of a pair of legs and subsequent merging of the circular structures. By taking wedges of these creatures and adding some additional structure, one can form a Lie algebra of creatures and treat it homologically in the sense of Chevalley and Eilenberg.

Now, imagine starting out with a graph and superimposing a creature upon each vertex (where the valence of the vertex coincides with the number of legs on the creature) to create a "creature-graph." If we define a differential on such a graph as the sum over all edge-collapses and subsequently "mate" the creatures inhabiting the vertex endpoints of the edge collapse, it turns out that the homology of the corresponding chain complex (roughly) coincides with the homology of the Lie algebra homology of the creatures.

We wish to formalize the process and make precise the claims made above in the context that the spider carries some kind of structure. Amazingly, when we impose a "Lie" structure on the spiders, either of the above homologies captures the cohomology of the group $Out(F_n)$. More can be said if we impose "Commutative" or "Associative" structure, but these cases will not be discussed herein. See [11], [25], or [7] for these other flavors.

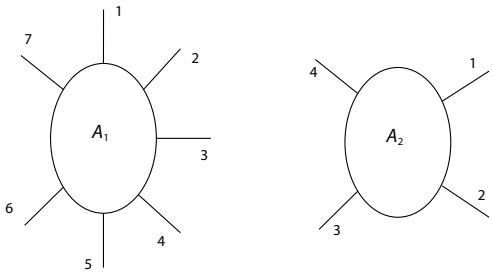
To achieve our goal, we review some basic concepts from algebra. The reader is invited to keep this example in mind as a template for the process. We follow [7] for much of the following material involving operads, graph homology, and the homology of the Lie algebra of a cyclic operad. The absence of a reference should not be interpreted as a claim to originality on the author's part.

2.3 Some notions from category theory

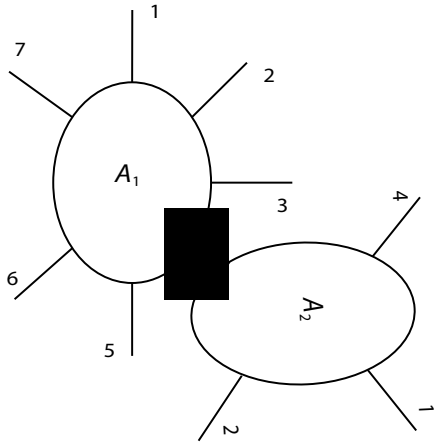
In preparation to define an operad, we review some basic definitions from category theory. Additional recommended texts are [23] [30], and [3].

By a *category* $\mathbf{C}$, we shall mean a class of objects together with a set of *morphisms* (or *arrows*) $\mathbf{C}(A, B)$ for every pair of objects A, B. These morphisms are required to form a monoid under composition. If an arrow f of $\mathbf{C}$ has a left and right inverse g, then we call f an *isomorphism*. A morphism of categories is a *functor*, i.e., a functor F is a map of categories that preserves the order of composition and maps the unit map of an object of a category to its corresponding image under F.

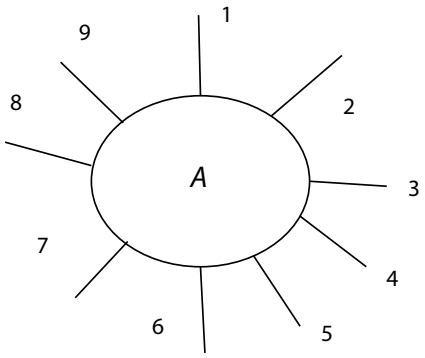
Many of the objects we consider in mathematics are categorical in nature. The collection of sets **Set** with arrows given by set maps forms a category. In particular, the category **FinSet** of finite sets with arrows given by bijections [permutations] is a category. Some algebraic categories include the category of groups, rings, (left) R-modules, and vector spaces over a field k denoted, respectively, $\mathbf{Gp}, \mathbf{Ring}, {}_R\mathbf{Mod}$, and **k-Vect**. On the topological side, we have the category of topological spaces, pointed topological spaces, smooth manifolds, and Lie groups denoted, respectively, $\mathbf{Top}, \mathbf{Top}_*, \mathbf{SMan}$ and **LieGp**. The morphisms in each of the given categories should be clear from the context.



(a) The creatures A_1 and A_2



(b) Mating along the legs 4 of A_1 and 3 of A_2



(c) The result of mating

Figure 2.4: Creature mating

One of first functors we are introduced to as a student is the first homotopy group π_1: $\mathbf{Top}_* \longrightarrow \mathbf{Gp}$ and later we learn of a sequence of functors called the (co-)homology functors $H_\bullet^{(\bullet)}$. Algebraic examples include the Abelianization of a group and the turning of an associative algebra into a Lie algebra via the commutator bracket. An important functor which plays a critical role in [21, 22] and [7] is the invariants functor $V \to V^{\mathfrak{g}}$ associated to a Lie algebra $\mathfrak{g}$ with an action on V.

Definition 2.3.1. Given two functors $\mathbf{C} \xrightarrow{F,G} \mathbf{D}$, a *natural transformation* η from F to G is an assignment of every object C in $\mathbf{C}$, a morphism $FC \xrightarrow{\eta_C} GC$ in $\mathbf{D}$ such that for every morphism $C \xrightarrow{f} C'$ in $\mathbf{C}$, we have a commutative diagram as below.

$$\begin{array}{ccc} FC & \xrightarrow{\eta_C} & GC \\ {\scriptstyle Ff}\downarrow & & \downarrow{\scriptstyle Gf} \\ FC' & \xrightarrow{\eta_{C'}} & GC' \end{array}$$

The concept of a natural transformation makes precise our notion of "a morphism of functors." A familiar case of naturality arises in the Mayer-Vietoris sequence for homology

which relates the homology of a space to certain subspaces of the space. If we assume $X = int(A) \cup int(B)$, then there is an exact sequence called the Mayer-Vietoris sequence

$$\cdots \to H_n(A \cap B) \to H_n(A) \oplus H_n(B) \to H_n(X) \to H_{n-1}(A \cap B) \to \cdots .$$

To say that the Mayer-Vietoris sequence is natural is to say that if we have another space X' that decomposes into the interior of its subspace A' and B' and if f is a continuous map $X \to X'$ that carries A into A' and B into B', then we have a commutative diagram with exact rows

$$\begin{array}{ccccccccccc}
\cdots & \longrightarrow & H_n(A \cap B) & \longrightarrow & H_n(A) \oplus H_n(B) & \longrightarrow & H_n(X) & \longrightarrow & H_{n-1}(A \cap B) & \longrightarrow & \cdots \\
 & & \downarrow{\scriptstyle f_*} & & \downarrow{\scriptstyle f_*} & & \downarrow{\scriptstyle f_*} & & \downarrow{\scriptstyle f_*} & & \\
\cdots & \longrightarrow & H_n(A' \cap B') & \longrightarrow & H_n(A') \oplus H_n(B') & \longrightarrow & H_n(X') & \longrightarrow & H_{n-1}(A' \cap B') & \longrightarrow & \cdots
\end{array}$$

2.4 Monoidal categories and operads

Recall that a monoidal structure on a set is an associative binary relation with a distinguished identity element 1. We wish to mimic this construction in the context of a category:

Definition 2.4.1. A *monoidal category* $(\mathbf{C}, \otimes)$ is a category $\mathbf{C}$ with a functor $\mathbf{C} \times \mathbf{C} \xrightarrow{\otimes} \mathbf{C}$, a natural isomorphism (called the *associator*)

$$- \otimes (- \otimes -) \xrightarrow{\alpha_{-,-,-}} (- \otimes -) \otimes -,$$

and an identity object I with natural isomorphisms $I \otimes C \xrightarrow{l_C} C$ and $C \otimes I \xrightarrow{r_C} C$ such that the diagrams below commute.

$$\begin{array}{ccc}
 & (A \otimes B) \otimes (C \otimes D) & \\
{\scriptstyle \alpha}\nearrow & & \searrow{\scriptstyle \alpha} \\
A \otimes (B \otimes (C \otimes D)) & & ((A \otimes B) \otimes C) \otimes D \\
\downarrow{\scriptstyle 1_A \otimes \alpha} & & \uparrow{\scriptstyle \alpha \otimes 1_D} \\
A \otimes ((B \otimes C) \otimes D) & \xrightarrow[\alpha]{} & (A \otimes (B \otimes C)) \otimes D
\end{array}
\qquad
\begin{array}{ccc}
A \otimes (I \otimes B) & \xrightarrow{\alpha} & (A \otimes I) \otimes B \\
\downarrow{\scriptstyle 1_A \otimes l_B} & & \downarrow{\scriptstyle r_A \otimes 1_B} \\
A \otimes B & \xleftarrow{=} & A \otimes B
\end{array}$$

It is a standard exercise in a beginning graduate algebra class to verify the isomorphism $R \otimes_R M \cong M \otimes_R R \cong M$ for a module over a commutative ring R. Unbeknownst to us at the time, we were verifying that the ring R acts as a unit in ${}_R\mathbf{Mod}$ under the tensor product! Furthermore, the familiar associativity of $\otimes_R$ is the pentagonal diagram above. In particular, if we specialize our ring R, we may assert that the category of k-vector spaces and the category of Abelian groups are monoidal. A less obvious example arises if we consider the category of pointed topological spaces $\mathbf{Top}_*$ under the smash product. In this case, the unit is realized as S^0.

Definition 2.4.2. A *species* is a functor **FinSet** $\to$ **k-Vect**. The category of species will be denoted **Sp** and the morphisms are the natural transformations between the functors

and given a species $\mathbf{q}$, the image of the set I will be denoted $\mathbf{q}[I]$ and is called the space of $\mathbf{q}$-*structures* on I.

Note that the objects of **Sp** are functors and so it makes sense to say the arrows of **Sp** are natural transformations. There is a monoidal structure which can be placed on the category of species; it is given by the $\circ$ operation illustrated in Figure 2.4(b) and is realized as a type of formal substitution.

Let us algebraically describe the monoidal structure given by $\circ$ as in Appendix B of [1]. Let $\mathbf{p}[I]$ denote the set of all partitions of the finite set I and given a partition π of $\mathbf{p}[I]$, we write $B \in \pi$ to mean a *block* in the partition π. Then, given two species $\mathbf{q}$ and $\mathbf{q}'$, we define the substitution product $\circ$ by

$$(\mathbf{q} \circ \mathbf{q}')[I] = \bigoplus_{\pi \in \mathbf{p}[I]} (\mathbf{q}[\pi] \otimes \bigotimes_{B \in \pi} \mathbf{q}'[B]) \tag{2.1}$$

The reader likely noticed the obtuse notation $\mathbf{p}[I]$ to denote the collection of all partitions on I. The collection $\mathbf{p}[I]$ here refers to the species of partitions: given a finite set I, the functor $\mathbf{p}$ associates to it all partitions of I.

This $\circ$ operation can be viewed as a kind of substitution by the following description. Fix a finite set I and a partition π of I. Each summand in 2.1 consists of two components; the second component $\bigotimes_{B \in \pi} \mathbf{q}'[B]$ can be thought of as the parameters determined by the species $\mathbf{q}'$ and the first component can be viewed as a post-processor or placeholder for the output of $\mathbf{q}'$

As a concrete example, let us consider the extreme case where our partition π^0 of I is into singletons. Then each block B in the partition π^0 consists of a single element which is fed to $\mathbf{q}'$. The "output" $\mathbf{q}'[\,\{x\}\,]$ is then associated to the partition coordinate of π^0 in the image of $\mathbf{q}[\pi^0]$. For another example in the case of the Lie species, see Example 2.4.7 below.

The above construction extends to an arbitrary category and the choice of **k-Vect** is for our intent. A left Σ_n-module structure can be given to the $\mathbf{q}$-structure $\mathbf{q}[\{1, 2, 3, \cdots, n\}] = \mathbf{q}[n]$ as follows: every permutation $\sigma \in \Sigma_n$ induces an automorphism $\mathbf{q}[n](\sigma)$ of $\mathbf{q}[n]$ which permutes the set $[n]$.

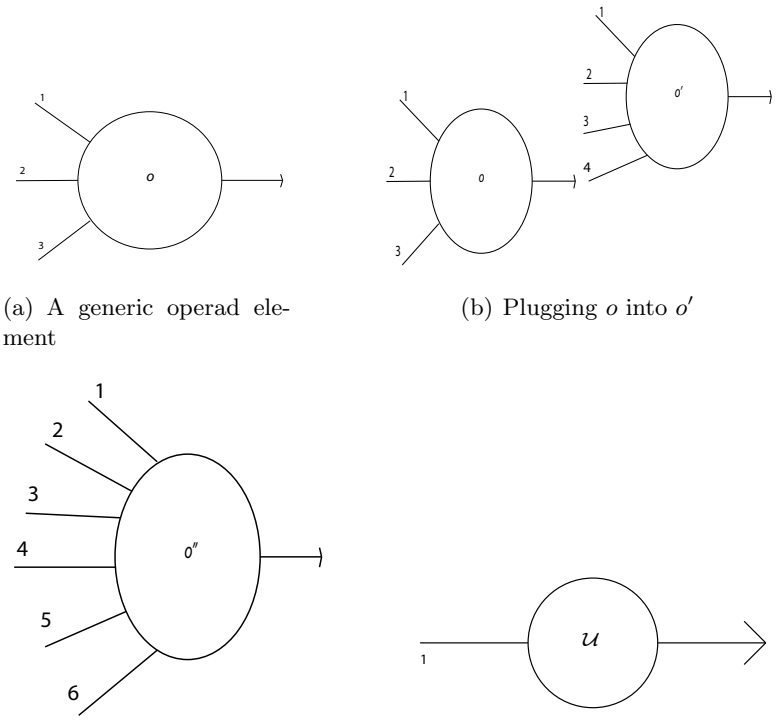
There are several approaches to developing operads, two of which are given below. The original motivation for operads was to study iterated loop spaces in J.P. May's "The Geometry of Iterated Loop Spaces," but it has been noted [26] that the study of operads dates back to 1898 in Whitehead's "A Treatise on Universal Algebra." In any event:

Definition 2.4.3. An *operad* is a monoid $\mathcal{O}$ in the monoidal category $(\mathbf{Sp}, \circ, \mathbf{u})$ where the unit is given by $\mathbf{u}[X] = k$ if $|X| = 1$ and $\emptyset$ otherwise and the operation $\circ$ is the formal substitution described above. [We note that the unit of the operad may also be denoted $1_{\mathcal{O}}$.]

Another approach, in the context of real vector spaces, is given in [7]; the reader is warned that we shall use them interchangeably.

Definition 2.4.4. A collection $\mathcal{O}$ of real vector spaces $\mathcal{O}[m]_{m \geq 1}$, is an *operad* if there is an associative composition

$$\begin{aligned} \mathcal{O}[m] \otimes \mathcal{O}[i_1] \otimes \cdots \otimes \mathcal{O}[i_m] &\xrightarrow{\gamma} \mathcal{O}[i_1 + \cdots i_m] \\ (o, o_1, o_2, \cdots, o_m) &\longmapsto (((o \circ_1 o_1) \circ_{l_1+1} o_2) \circ_{l_1+l_2+1} o_3) \cdots) \end{aligned}$$



(a) A generic operad element

(b) Plugging o into o'

(c) The result

(d) The unit operad

Figure 2.5: Components of an operad

where l_i is the arity of o_i, together with a right Σ_m-action on $\mathcal{O}[m]$, and a unit $1_{\mathcal{O}} \in \mathcal{O}[1]$.

Additionally, there is an axiom of equivariance. We refer the reader to [26] for a description and proof of the equivalence of these two formulations of an operad.

Example 2.4.5 (The Unit Operad, $\mathcal{U}$)**.** We define the unit species $\mathbf{u}$ by $\mathbf{u}[X] = \{X\}$ if $|X| = 2$ and $\emptyset$ otherwise. The operad $\mathcal{U}$ is the defined as $\mathcal{U}[X] = \{X\}$ if X is a singleton and $\emptyset$ otherwise. Pictorially,

Example 2.4.6 (The Associative Operad, $\mathcal{A}$)**.** Let T be a planar rooted tree with n edges emanating from one internal vertex. The image $\mathcal{A}[n]$ is spanned by such trees and the cyclic ordering given by the planar embedding is equivalent to a cyclic ordering of the edges of the tree. See Figure 2.5(a). The $\circ$ operation of two such elements is performed by identifying the root of the first tree with an edge of the second tree, collapsing the resulting edge and inserting the edges from the first tree into the second while preserving the original ordering. This is illustrated in Figure 2.5(c) if one assumes the operads pictured are embedded in the plane.

Example 2.4.7 (The Lie Operad, $\mathcal{L}$)**.** A bracket sequence on the set $X = \{x_1, \cdots, x_k\}$ is a parenthesization of the elements of X such that each element is used once and only once. An example of a bracket sequence on $\{a, b, c, d, e\}$ is $[[[c, a], e], [b, d]]$. As a note of interest, the total number of such bracket sequences on a set of size $n + 1$ is given by the

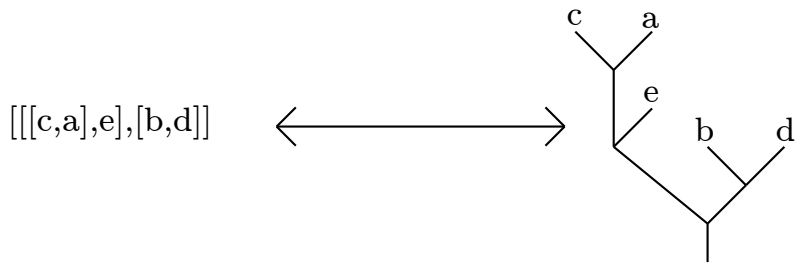
n-th Catalan number

$$C_n = \frac{1}{n+1}\binom{2n}{n}.$$

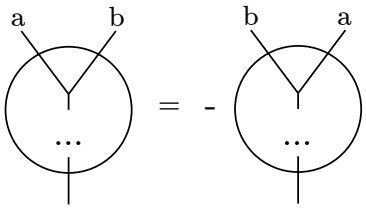
We define the Lie species $\mathbf{Lie}[n]$ to be the span of all bracket sequences on n elements subject to the familiar antisymmetry and Jacobi relations of a Lie algebra. The substitution rule for the elements s_1 and s_2 is given by inserting in place the bracket sequence s_1 into the desired element in the bracket sequence of s_2, e.g. $[b,c] \circ_b [[a,b],c] = [[a,[b,c]],c]$.

To bring this into a more tractable form, we note that we can identify a bracket sequence as a rooted planar binary tree modulo the antisymmetry (Figure 2.6(b)) and IHX (Figure 2.6(c)) relations. The substitution operation for the Lie operad is given in Figure 2.6(d).

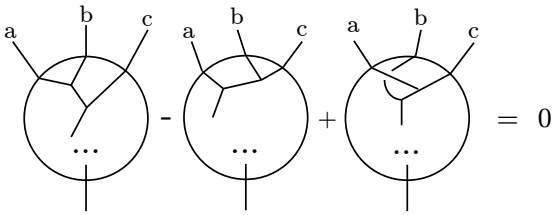
We now explain how the monoidal structure on the category of species gives rise to an operadic composition $\mathbf{Lie} \circ \mathbf{Lie} \xrightarrow{\gamma} \mathbf{Lie}$. Given a finite set I, a partition π on I, and bracket sequences on π and a block B of π, we can determine a bracket sequence on I by substituting the bracket sequence for each block B in place into the partition bracket sequence associated



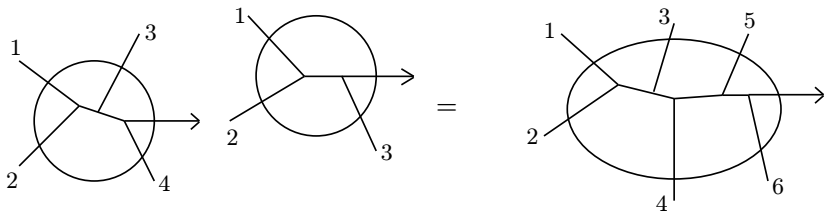
(a) The correspondence between a bracket sequence and a planar binary tree



(b) The antisymmetry relation



(c) The IHX relation



(d) Substitution in the Lie operad

Figure 2.6: Properties of the Lie operad

to π. We visualize this by forming a rooted tree where the leaves correspond to the blocks in the partition and for every block, we determine a bracket sequence. This block-bracket-sequence in turn defines a rooted tree of which we graft its root onto the corresponding leaf for the block on the partition tree.

Definition 2.4.8. A *cyclic operad* $\mathcal{O}$ is an operad such that the Σ_n action on $\mathcal{O}$ extends to a Σ_{n+1} action such that the axioms for $\mathcal{O}$ still hold.

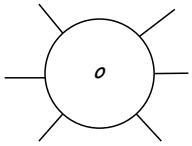


Figure 2.7: Visualizing an element of a cyclic operad

One way to think of a cyclic operad is to erase the distinction of the output and allow any input to serve as an output. We accomplish this by labeling the output by 0 and think of Σ_{n+1} as being the group of bijections on the set $\{0, 1, \cdots, n\}$. Rather than illustrating an operad as a "directed" object, we visualize in a more symmetric manner as in Figure 2.7.

The Lie operad is an example of a cyclic operad. Later, we shall define a pointed version of the Lie operad and show that it is also a cyclic operad. Using this, we will be able to analyze the homology of Auter space.

2.5 The graph homology of a cyclic operad

Recall from the introductory example that we wish to formalize the concept of attaching a "creature" to the vertices of a graph and subsequently define a homology on the resulting graph complex generated by these graphs. Of import in this process and the sequel is the notion of an oriented graph:

Definition 2.5.1. An *orientation* on the graph G is a choice of a unit vector in

$$\det \mathbb{R}V(G) \otimes \bigotimes_{e \in E(G)} \det \mathbb{R}H(e)$$

where $\det \mathbb{R}\mathcal{B}$ corresponds to the top-dimensional wedge of the finite dimensional real vector space with basis $\mathcal{B}$, $E(G)$ = set of edges of G, $V(G)$ = vertices of G, $H(e)$ = the two half edges that compose the edge e, and $H(v)$ = the collection of half edges incident to the vertex v.

Note, if $\dim \mathbb{R}\mathcal{B} = n$, then $\det \mathbb{R}\mathcal{B} = \wedge^n \mathbb{R}\mathcal{B}$ is one-dimensional and hence it makes sense to speak of a choice of a "positive" or "negative" orientation. For a connected graph, there are equivalent formulations of orientation; we refer the reader to [7] for a proof of the equivalences and record the result here for reference.

Lemma 2.5.2. *The following formulations of orientation for a connected graph X are equivalent, up to canonical isomorphism:*

1. $\det H_1(X;\mathbb{R}) \otimes \det \mathbb{R}E(X)$

2. $\det \mathbb{R}V(X) \otimes \bigotimes_{e\in E(X)} \det \mathbb{R}H(e)$

3. $\bigotimes_{v\in V(X)} \det \mathbb{R}H(v) \otimes \det \bigoplus_{|H(v)|even} \mathbb{R}v$

To make sense of identifying a vertex of a graph with a "creature" we set the groundwork for defining an $\mathcal{O}$-spider

Definition 2.5.3. Fix an n-star S with $n \geq 2$ edges emanating from a central vertex. Then a *labeling* of S is a bijection between $\{0, 1, \cdots, n-1\}$ and $E(S)$, the edges of S.

As discussed in the previous section, the symmetric group Σ_n acts on the finite set $[n]$ attached to **p**-structure $\mathbf{p}[\mathbf{n}]$ and hence when **p** gives rise to a cyclic operad $\mathcal{O}$, there is a coherent extension of the action to a Σ_{n+1} action on $\mathcal{O}$. Similarly, we have an action of Σ_n on the labelings of an n-star. Recall that the space of coinvariants corresponding to the action of a group G on the space X, denoted X_G, is the space $X/\langle x - gx\rangle$.

Definition 2.5.4. Let $\mathcal{O}$ be a cyclic operad and let $\mathcal{L}$ be the set of labelings of the n-star for $n \geq 2$. The *space of $\mathcal{O}$-spiders with n legs* is the space

$$\mathcal{OS}[n] = \left(\bigoplus_{\mathcal{L}} \mathcal{O}[n-1]\right)_{\Sigma_n}$$

where the Σ_n action is given by $\sigma(o_L) = (\sigma o)_{\sigma L}$.

From the space of $\mathcal{O}$-spiders with n legs, we form the full space of $\mathcal{O}$-spiders $\mathcal{OS} = \bigoplus_{n\geq 2} \mathcal{OS}[n]$. Since $\mathcal{O}$ is a cyclic operad, there is a composition law defined on $\mathcal{O}$ and this gives rise to a mating law on $\mathcal{OS}$. We illustrate the mating law via an example with the Lie operad:

Example 2.5.5. Consider the two Lie spiders S (left spider) and S' (right spider) in Figure 2.6(d). We wish to mate S and S' along the legs l of S and l' of S. To do so, choose a representative S_Σ and ${S_\Sigma}'$ of S and S' where the labeling of the leg l of S is the output of underlying operad element o_S of S and the labeling of the leg l' is the first input of the underlying operad element o_T of T. Via the substitution law in $\mathcal{L}$, plug the leg l of o_S into the leg l' of o_T, collapse the adjoined legs to form a new edge in the tree, and subsequently relabel the legs so that the leg ordering is coherent with the original leg ordering.

Definition 2.5.6. Let G be an oriented graph with all vertices at least bivalent and let $\mathcal{O}$ be a cyclic operad. The n-valent vertex v of G is said to be *decorated* by the spider $S \in \mathcal{OS}[n]$ if there is a fixed bijection between $H(v)$ and the legs of S. The graph G is said to be an *$\mathcal{O}$-graph* if all the vertices of G are $\mathcal{O}$-decorated. If the graph G is $\mathcal{O}$-decorated, then we shall denote this by $\mathcal{O}G$. When there is not danger of confusion, we shall suppress the $\mathcal{O}$ prefix.

Later, we shall give another formulation of an $\mathcal{O}$-graph that allows the vertices of G to be decorated by two operad spiders of differing type. This formulation will be used to extend the methods for determining the homology of $Out(F_n)$ to that of $Aut(F_n)$.

Definition 2.5.7. The *k-th* chain group $\mathcal{OG}_k$ of $\mathcal{O}$-graphs is the real vector space spanned by $\mathcal{O}$-graphs with k vertices subject to the relations

1. (Orientation) $(\mathcal{O}G, or) = -(\mathcal{O}G, -or)$
2. (Vertex Linearity) If a vertex v of $\mathcal{O}G$ is decorated by the spider $S = a_1S_1 + a_2S_2$ where $a_1, a_2 \in \mathbb{R}$ and $S_1, S_2 \in \mathcal{OS}[k]$, then $\mathcal{O}G = a_1\mathcal{O}G_{S_1} + a_2\mathcal{O}G_{S_2}$ where $\mathcal{O}G_{S_i}$ denotes the decoration of the vertex v of $\mathcal{O}G$ by the spider S_i.

Let G be an oriented graph and let $\mathcal{O}G$ be its decoration by $\mathcal{O}$-spiders. Define $\mathcal{O}G_e$ to be the $\mathcal{O}$-graph formed by mating the underlying spiders decorating the vertex endpoints v and w of e along their legs that form e. In the event that e is a loop, define $\mathcal{O}G_e = 0$. We orient G_e by choosing an orientation representative of G such that v is first and w is second in the vertex ordering and e is oriented from v to w; the orientation of G_e is then inherited from its parent G and the new vertex formed by mating the spiders decorating v and w is first in the vertex ordering.

With these conventions, this action has degree -1 and we subsequently define

$$\begin{aligned} \mathcal{OG}_k &\xrightarrow{\partial_E} \mathcal{OG}_{k-1} \\ \mathcal{O}G &\longmapsto \sum_{e \in E(G)} \mathcal{O}G_e. \end{aligned} \tag{2.2}$$

As we formed the total space of $\mathcal{O}$-spiders via a grading on the number of spider legs, we form the space of $\mathcal{O}$-graphs

$$\mathcal{OG} = \bigoplus_{k \geq 1} \mathcal{OG}_k.$$

Proposition 2.5.8. $\partial_E^2 = 0$. *In particular, the pair* $(\mathcal{OG}, \partial_E)$ *is a chain complex.*

Proof. Let e_1, e_2 be edges of G with vertex endpoints v_1, v_2 and w_1, w_2, respectively. Without loss of generality (the result would differ by a global sign), choose the orientation of G so that in the vertex ordering we have v_1 before v_2, w_1 before w_2, and v_2 before w_2. Then $(G_{e_1})_{e_2} = (G_{e_2})_{e_1}$ except that the orientation differs since when we collapse e_2 first, we pick up a sign from the fact that v_2 precedes w_2 in the vertex ordering. □

As shown, the pair $(\mathcal{OG}, \partial_E)$ is a chain complex and so we define the $\mathcal{O}$-graph homology of the cyclic operad $\mathcal{O}$ to be the homology of $\mathcal{OG}$ with respect to ∂_E. We also define the reduced chain groups $\overline{\mathcal{OG}}_k$ to be $\mathcal{OG}_k$ quotiented by the subspace of graphs with some vertex decorated by $1_{\mathcal{O}}$. The following proposition is noted in [7] and we provide a proof now.

Proposition 2.5.9. *The spaces* $\mathcal{OG}$ *and* $\overline{\mathcal{OG}}$ *are Hopf algebras.*

Proof. Recall that a Hopf algebra is a bialgebra with an antipode map. We define an algebra structure by disjoint union with the unit being the empty graph, denoted 1. A coalgebra structure is defined so that the connected graphs are the primitives and the comultiplication is extended linearly over disjoint union with the counit dual to the empty graph. We show that these structures are compatible. Note $\Delta(1) = 1 \otimes 1$ and if G, G' are connected, then

(using the sumless Sweedler notation)

$$\begin{aligned}
(\Delta\mu)(G \otimes G') &= \Delta(G) \sqcup \Delta(G') \\
&= (G_{(1)} \otimes 1 + 1 \otimes G_{(2)}) \sqcup (G'_{(1)} \otimes 1 + 1 \otimes G'_{(2)}) \\
&= G_{(1)} \sqcup G'_{(1)} \otimes 1 + G_{(1)} \otimes G'_{(2)} + G'_{(1)} \otimes G_{(2)} + 1 \otimes G_{(2)} \sqcup G'_{(2)} \\
&= (\mu \otimes \mu)(G_{(1)} \otimes G'_{(1)} \otimes 1 \otimes 1 + G_{(1)} \otimes 1 \otimes 1 \otimes G'_{(2)} \\
&\quad + G'_{(1)} \otimes 1 \otimes 1 \otimes G_{(2)} + 1 \otimes 1 \otimes G_{(2)} \otimes G'_{(2)} \\
&= (\mu \otimes \mu)(I \otimes T \otimes I)(G_{(1)} \otimes 1 \otimes G'_{(1)} \otimes 1 + G_{(1)} \otimes 1 \otimes 1 \otimes G'_{(2)} \\
&\quad + G'_{(1)} \otimes 1 \otimes G_{(2)} \otimes 1 + 1 \otimes 1 \otimes G_{(2)} \otimes G'_{(2)} \\
&= (\mu \otimes \mu)(I \otimes T \otimes I)(\Delta \otimes \Delta)(G \otimes G').
\end{aligned}$$

That is, $\Delta\mu = (\mu\otimes\mu)(I\otimes T\otimes I)(\Delta\otimes\Delta)$ by linear extension and so Δ and μ are compatible. The remaining conditions $\epsilon(1) = 1$ and $\epsilon\mu(G \otimes G') = \mu(\epsilon(G) \otimes \epsilon(G'))$ follow from the definitions of the counit and multiplication map. Finally, the antipode of a graph reverses the orientation. □

Since Δ is defined so that the connected graphs are primitive, the subspace of primitives $P\mathcal{OG}$ is generated by connected graphs and similarly for $\overline{\mathcal{OG}}$. Furthermore, the spaces $P\mathcal{OG}$ and $P\overline{\mathcal{OG}}$ are chain complexes with respect to ∂_E.

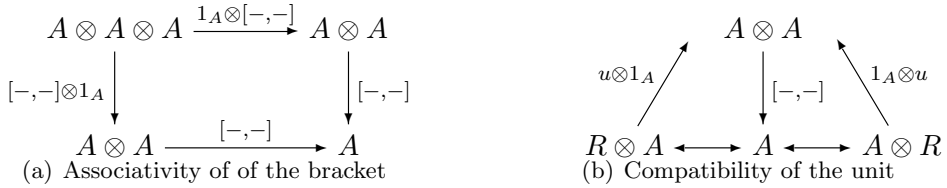
2.6 Lie algebra homology

Let R be a commutative unital ring. An *algebra* over R is an R-module A with a bilinear binary operation $A \otimes A \xrightarrow{[-,-]} A$. If there is a unit map $R \xrightarrow{u} A$ and the diagrams in Figure 2.8 are commutative (that is, A is a monoid), then A is called an *associative algebra*. Note that all algebras will not be assumed to be associative.

Definition 2.6.1. Let k be a field of characteristic $\neq 2$. Then a k-algebra $\mathfrak{g}$ is a *Lie algebra* if the bilinear operation of $\mathfrak{g}$ satisfies

1. (Antisymmetry) $[x, y] = -[y, x]$ for all x, y in $\mathfrak{g}$,
2. (Jacobi identity) $[x, [y, z]] + [y, [z, x]] + [z, [x, y]] = 0$ for all x, y, z in $\mathfrak{g}$.

Given a Lie algebra $\mathfrak{g}$, one can construct its universal enveloping algebra $U(\mathfrak{g})$ by quotienting the tensor algebra of $\mathfrak{g}$ by the ideal generated by $a \otimes b - b \otimes a - [a, b]$. The quotient is a unital associative algebra and using standard methods we can form objects such as $U(\mathfrak{g})$-modules. This is the route we take when defining the homology of a Lie algebra with



(a) Associativity of of the bracket (b) Compatibility of the unit

Figure 2.8: Commutative diagrams for an associative algebra A

coefficients in a $U(\mathfrak{g})$-module. The complete derivation of Lie algebra homology comes from an analysis of a cocomplex which is suitably dualized to form the Koszul complex which calculates our desired homology. Ultimately, it is an exercise in homological algebra and proofs are omitted; the reader is referred to the excellent book [20] for a detailed exposition and derivation.

Definition 2.6.2. Let $\mathfrak{g}$ be a Lie algebra and let V be a $U(\mathfrak{g})$-module. The *homology of* $\mathfrak{g}$ *with coefficients in* V, denoted $H_*(\mathfrak{g}; V)$, is the homology of the chain complex $X_n = \wedge^n \mathfrak{g} \otimes V$ where the boundary operator is given by

$$\begin{aligned}\partial(X_1 \wedge \cdots \wedge X_n \otimes v) = &\sum_{i=1}^{n} (-1)^i X_1 \wedge \cdots \wedge \widehat{X_i} \wedge \cdots \wedge X_n \otimes X_i v \\ &+ \sum_{r<s} (-1)^{r+s} [X_r, X_s] \wedge X_1 \wedge \cdots \wedge \widehat{X_r} \wedge \cdots \wedge \widehat{X_s} \wedge \cdots \wedge X_n \otimes v.\end{aligned}$$

We note that in the case of trivial coefficients the definition given reduces to one given in [7].

Example 2.6.3. We calculate the homology groups with trivial $\mathbb{R}$ coefficients of $\mathfrak{g} = \mathfrak{gl}(2, \mathbb{R})$. Since $\mathbb{R}$ acts trivially on $\mathfrak{g}$, the first summation in the boundary definition vanishes and we are left with

$$\partial(X_1 \wedge \cdots \wedge X_n) = \sum_{i<j}^{n} (-1)^{i+j} [X_i, X_j] \wedge X_1 \wedge \cdots \wedge \widehat{X_i} \wedge \cdots \wedge \widehat{X_j} \wedge \cdots \wedge X_n.$$

Note that $\mathfrak{g}$ is generated by $X_1 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, X_2 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, X_3 = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$, and $X_4 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Computing the brackets $[X_i, X_j] = X_i X_j - X_j X_i$, we get

$$\begin{aligned}[X_1, X_2] &= 2X_2 \\ [X_1, X_3] &= -2X_3 \\ [X_2, X_3] &= X_1,\end{aligned}$$

and all other brackets are zero. If we work our way down the complex

$$0 \longrightarrow \wedge^4 \mathfrak{g} \longrightarrow \wedge^3 \mathfrak{g} \longrightarrow \wedge^2 \mathfrak{g} \longrightarrow \wedge \mathfrak{g} \longrightarrow \mathbb{R} \longrightarrow 0$$

we see that the only nontrivial images of elements from the chain groups are

$$\begin{aligned}\partial(X_1 X_2 X_4) &= 2X_2 X_4 \\ \partial(X_1 X_3 X_4) &= -2X_3 X_4 \\ \partial(X_2 X_3 X_4) &= X_1 X_4 \\ \partial(X_1 X_2) &= 2X_2 \\ \partial(X_1 X_3) &= 2X_3 \\ \partial(X_2 X_3) &= X_1\end{aligned}$$

and so when we compute the homology, we get

$$\begin{aligned}
H_1(\mathfrak{g};\mathbb{R}) &= \mathbb{R}\{X_1,X_2,X_3,X_4\}/\mathbb{R}\{X_1,X_2,X_3\}\\
&\cong \mathbb{R}\\
H_2(\mathfrak{g};\mathbb{R}) &= \mathbb{R}\{X_1X_4,X_2X_4,X_3X_4\}/\mathbb{R}\{X_2X_4,X_3X_4,X_1X_4\}\\
&\cong 0\\
H_3(\mathfrak{g};\mathbb{R}) &= \mathbb{R}\{X_1X_2X_3\}/\mathbb{R}\{0\}\\
&\cong \mathbb{R}\\
H_4(\mathfrak{g};\mathbb{R}) &= \mathbb{R}\{X_1X_2X_3X_4\}/\mathbb{R}\{0\}\\
&\cong \mathbb{R}.
\end{aligned}$$

In the previous section, we discussed how to associate a graph homology to a cyclic operad. We now associate a Lie algebra homology to a cyclic operad. The idea is to attach an element from a symplectic vector space to the legs of an $\mathcal{O}$-spider to get what is called a symplectospider; the space generated by these special spiders then forms a Lie algebra and we can discuss its (Lie algebra) homology.

Definition 2.6.4. A (real) *symplectic vector space* is a vector space $\mathbb{R}^{2n}$ with basis $\mathcal{B}_n = \{p_1,\cdots,p_n,q_1,\cdots,q_n\}$ and a bilinear form $\omega(\cdot,\cdot)$ that satisfies

1. $\omega(p_i,q_j) = -\omega(q_j,p_i) = \delta_{ij}$
2. $\omega(p_i,p_j) = \omega(q_i,q_j) = 0$.

Definition 2.6.5. Let (V_n,ω) be a real symplectic vector space. Then the *space of symplectospiders* is defined to be

$$\mathcal{LO}_n = \bigoplus_{m\geq 2} \left(\mathcal{OS}[m]\otimes V_n^{\otimes m}\right)_{\Sigma_m}$$

where the action of Σ_m is simultaneous. We write the element $\mathbb{S}$ of $\mathcal{LO}_n$ as $[S\otimes v_1\otimes\cdots\otimes v_m]$ where S is an $\mathcal{O}$-spider.

As indicated, the space $\mathcal{LO}_n$ forms a Lie algebra with the bracket defined as follows. Given symplectospiders $\mathbb{S} = [S\otimes v_1\otimes\cdots\otimes v_m]$ and $\mathbb{S}' = [S'\otimes v_1'\otimes\cdots\otimes v_m']$, we perform the symplectic mating $(\mathbb{S},\lambda)!(\mathbb{S}',\lambda')$ along the legs λ of S and λ' of S' and then attach the coefficient $\omega(v_\lambda,v_{\lambda'})$ where v_* is the element of V_n associated to the leg $*$ of S. The bracket of these spiders is then defined to be the sum of all possible matings of the spiders:

$$[\mathbb{S},\mathbb{S}'] = \sum_{\lambda\in\mathbb{S},\lambda'\in\mathbb{S}'} (\mathbb{S},\lambda)!(\mathbb{S}',\lambda').$$

We note that this bracket is of the Lie type as it satisfies the Jacobi identity and is anti-symmetric by the properties of the symplectic form.

Recall that the symplectic Lie algebra $\mathfrak{sp}(2n)$ is the collection of $2n\times 2n$ matrices X such that $XJ = -JX^T$ where $J = \begin{pmatrix} 0 & I \\ -I & 0\end{pmatrix}$. Let $\mathcal{LO}_n^0$ denote the subspace of $\mathcal{LO}_n$ spanned by symplectospiders decorated by $1_{\mathcal{O}}$. Then it can be shown that $\mathcal{LO}_n^0\cong\mathfrak{sp}(2n)$. We note $\mathcal{LO}_n^0$ acts on $\mathcal{LO}_n$ via the bracket and, in light of the previous isomorphism,

$\mathfrak{sp}(2n)$ acts on $\mathcal{LO}_n$. Moreover, the natural inclusion of $V_n \to V_{n+1}$ extends to an inclusion $\mathcal{LO}_n \to \mathcal{LO}_{n+1}$ compatible with the inclusion $\mathfrak{sp}(2n) \to \mathfrak{sp}(2n+2)$. Thus we can define the infinite dimensional symplectic Lie algebra as a direct limit $\mathcal{LO}_\infty = \varinjlim \mathcal{LO}_n$. Note that the natural inclusion $\mathcal{LO}_n \to \mathcal{LO}_{n+1}$ mentioned above induces a chain map $\wedge\mathcal{LO}_n \to \wedge\mathcal{LO}_{n+1}$ and so we have $H_k(\mathcal{LO}_\infty;\mathbb{R}) = \varinjlim H_k(\mathcal{LO}_n;\mathbb{R})$.

We are now able to state a theorem of Kontsevich that connects the graph and Lie algebra homologies associated to certain cyclic operads:

Theorem 2.6.6. *(Kontsevich) Let $\mathcal{O}$ be the Associate, Commutative, or Lie operad. Then there is a Hopf algebra isomorphism between $H_*(\mathcal{LO}_\infty;\mathbb{R})$ and $H_*(\mathcal{OG})$. In particular, we have $PH_*(\mathcal{LO}_\infty;\mathbb{R}) \cong H_*(P\mathcal{OG})$.*

The proof of this result requires [among other things] one to analyze the $\mathfrak{sp}(2n)$-invariants of $\wedge\mathcal{LO}_n$ and particular subspaces of $\mathcal{OG}$. A thorough, detailed exposition that holds for *all* cyclic operads can be found in [7].

2.7 The homology of $Out(F_n)$ and Morita's trace map

The question is now how does this seemingly disconnected background connect to the cohomology of $Out(F_n)$? This is answered by a theorem of Kontsevich [21, 22]:

Theorem 2.7.1. *(Kontsevich)*

$$PH_k(\ell_\infty) \cong H_k(\mathfrak{sp}(2\infty)) \oplus \bigoplus_{n\geq 2} H^{2n-2-k}(Out(F_n))$$

One can deduce this theorem from Theorem 2.6.6 by decomposing the space $P\mathcal{OG}$ into the space of graphs with only bivalent vertices (polygons) and the space of graphs with at least one ≥ 3-valent vertex. The space of polygons has the same Lie homology as $\mathfrak{sp}(\infty)$ whereas the complement space can be shown to compute the cohomology of $Out(F_n)$ by an intermediate connection which is explained in the following subsection (2.7.1) on the forested graph complex.

The main idea behind the proof is to relate the primitives in the homology of the infinite Lie algebra $\ell_\infty = \mathcal{L}\mathcal{L}_\infty$ to the graphs of $P\mathcal{LG}$. So, indeed, there is a relation between the homology of the Lie operad and $Out(F_n)$. A natural question to then ask is

Question 2.7.2. Is there an operad $\mathcal{O}$ such that its homology captures the cohomology of $Aut(F_n)$?

It will be shown in Chapter 4 that there is such an operad, the pointed Lie operad.

2.7.1 The forested graph complex

A combinatorial interpretation of the spine K_n of Outer space gives rise to the *forested graph complex* [7]:

Definition 2.7.3. Let G be a finite graph with trivalent vertices and let Φ be a subgraph of G that is acyclic and contains all of the vertices of G. A *forested graph* is a pair (G,Φ) with an orientation given by ordering the edges of Φ modulo even permutations. We call Φ

a *subforest* of G and when Φ is the largest subforest of G, we say Φ is the *maximal subforest* of G.

We make two observations based on this definition. First, it is possible that Φ may be a disconnected graph and so the terminology of subforest is appropriate as the subforest may be a disconnected collection of trees. Second, the reader will recall that an orientation of a graph is given by an ordering of the vertices and a choice of a direction on each of the edges of the graph, i.e. a choice of a unit vector in

$$\det \mathbb{R}V(G) \otimes \bigotimes_{e \in E(G)} \det \mathbb{R}H(e).$$

Following [7], we show that an orientation of the forested graph G is equivalent to an ordering of the edges of the forest.

Lemma 2.7.4. *Let (G, Φ) be a forested graph. Then an orientation of G is equivalent to an ordering of the forest (up to even permutation).*

Proof. Consider the graph $\widehat{G}$ formed from G by collapsing each subforest of Φ to a point to create a vertex of $\widehat{G}$. For each vertex v of $\widehat{G}$, we consider the ϵ-neighborhood of the pullback of v under the collapsing action and denote this by T_v. Note that each T_v is necessarily a binary tree and the interior of each T_v consists of one of the connected components of Φ. We write $\Phi = \cup_v \Phi_v$ where Φ_v is a connected component of Φ.

An orientation of $\widehat{G}$ is a choice of a unit vector in

$$\det \mathbb{R}V(\widehat{G}) \otimes \bigotimes_{e \in E(\widehat{G})} \det \mathbb{R}H(e)$$

and for the binary tree $T_\bullet$,

$$\bigotimes_{v \in V(\widehat{G})} \det \mathbb{R}E(T_v).$$

Thus an orientation of $(\widehat{G}, \{T_i\})$ is a choice of a unit vector in

$$\det \mathbb{R}V(G) \otimes \bigotimes_{e \in E(G)} \det \mathbb{R}H(e) \otimes \bigotimes_{v \in V(\widehat{G})} \det \mathbb{R}E(T_v).$$

Observe that if Φ_v has k edges, there are $k+3$ leaves on T_v and hence $H(v)$ has $k+3$ elements and so there are $2k+3$ edges of T_v. It follows that $E(\Phi_v)$ and $H(v)$ have opposite parity.

If we apply the partition lemma (Lemma 2 of [7]) to $E(T_v)$ with respect to the subsets $H(v)$ and $E(\Phi_v)$ and recall the fundamental isomorphism $V \otimes V^* \cong \mathbb{R}$ (for $\dim V = 1$), we get the canonical isomorphism

$$\det \mathbb{R}E(\Phi_v) \cong \det \mathbb{R}H(v) \otimes \det \mathbb{R}E(T_v).$$

Simplifying the expression for orientation with Lemma 2.5.2 and applying the canonical

isomorphism recently given we note

$$\begin{aligned}\det \mathbb{R}V(\widehat{G}) \otimes \bigotimes_{e\in E(\widehat{G})} \det \mathbb{R}H(e) \otimes \bigotimes_{v\in V(\widehat{G})} \det \mathbb{R}E(T_v) \\ \cong \bigotimes_{v\in V(G)} \det \mathbb{R}H(v) \otimes \det \bigoplus_{|H(v)|even} \mathbb{R}v \otimes \det \mathbb{R}E(T_v) \\ \cong \det \mathbb{R}E(\Phi_v) \otimes \bigotimes_{v\in V(G)} \det \bigoplus_{|H(v)|even} \mathbb{R}v \\ \cong \det \mathbb{R}E(\Phi_v) \otimes \bigotimes_{v\in V(G)} \det \bigoplus_{|E(\Phi_v)|odd} \mathbb{R}v \\ \cong \det \bigoplus_{e\in\Phi} \mathbb{R}e.\end{aligned}$$

□

We define the vector space $f\mathcal{G}_n$ to be the space spanned by forested graphs with $|E(\Phi)| = n$, modulo the IHX relation. The boundary of (G, Φ) is given by

$$\partial(G, \Phi) = \sum_e (G, \Phi \cup \{e\})$$

where the sum is over all edges of $G - \Phi$ such that $\Phi \cup \{e\}$ is a forest. We set $f\mathcal{G} = \oplus f\mathcal{G}_n$. We specialize $f\mathcal{G}_n$ further by defining $\widetilde{f\mathcal{G}}_n$ to be $f\mathcal{G}_n$ modulo the forested graphs that possess a separating edge, that is, an edge which its removal results in a disconnected graph.

2.7.2 The graphical trace map

Recall that a generator of $\wedge\ell_n$ is a wedge of symplectospiders. A *pairing* π on a wedge is a pairing of the legs of the spiders into disjoint two-element subsets. Note that the pairing π induces a pairing of the two elements of the symplectic vector space V_n decorating the legs given by π. If we now state that the individual leg pairings are identified at their "feet," the resulting object is a trivalent graph, denoted G_π. Since each [basis] symplectospider is built from a Lie operad element, there is a planar trivalent tree associated to each spider and we consider the subgraph of G_π consisting of the union of the interior of the spiders. This union is designated as our subforest, Φ_π which carries the orientation from the planar embedding of the tree decorating the operad element. As there is a natural order (up to even permutation) of the elements in the wedge of spiders, there is a natural induced order on the vertices of G_π and hence on the subforest Φ_π. The result of this pairing is the forested graph (G_π, Φ_π).

Given a pairing π on a wedge of symplectospiders, there is an associated weight $\omega(\pi)$ formed by taking the product over all paired elements' corresponding weights under the symplectic form ω. Consider the map $\wedge\ell_n \xrightarrow{\psi_n} f\mathcal{G}$ defined by

$$X_1 \wedge \cdots \wedge X_k \mapsto \sum_\pi \omega(\pi)(G_\pi, \Phi_\pi).$$

It was shown in [7] that ψ_n is a chain map and, in the limit, an isomorphism on homology:

$$H_*(\wedge\ell_\infty) \xrightarrow{H_*\psi_\infty} H_*(f\mathcal{G})$$

.

We define a functional on $f\mathcal{G}$ called the *graphical trace map*: $f\mathcal{G}_{4k-4} \xrightarrow{\tau_{4k-4}} \mathbb{Q}$. If (G, Φ) is a forested graph with $4k-4$ edges with G of rank $2k$, then $\tau_{4k-4}(G, \Phi) = 0$ unless the following hold:

1. Φ is the disjoint union of two linear trees Φ_1 and Φ_2,
2. Both Φ_1 and Φ_2 have $2k-2$ edges each,
3. There are non-forested edges connected the ends of Φ_1 and Φ_2, and
4. The vertices of Φ_1 and Φ_2 and remaining non-forested edges of G form a bipartite graph.

In this event, we reorder the edges of Φ_1 and Φ_2 so the edges of each subforest are in order and precede the ordering of the edges of Φ_2. Collapse each of the forests and the non-forested edge joining the endpoints of each of the forests. This collapse can be viewed as a planar embedding of a 2-vertex graph with $2k-1$ edges. We now adjust the quotient graph by reordering the edges coming into each vertex (via an action by some permutation σ) so as to coincide with the initial planar embedding of our forested graph. In this case, define $\tau_{4k-4}(G, \Phi) = sign(\sigma)$.

We illustrate the process by considering the forested graph (G, Φ) in Figure 2.9(a). To see that G is indeed a candidate for nonzero trace, note Φ is the disjoint union of two linear trees of length $4 = 2*3-2$ edges, $\pi_1(G)$ has rank $6 = 2*3$, each of the subforests has a nonforested edge joining its endpoints, and the remaining edges form a bipartite graph on the vertices of G. As Φ is not coherently ordered, we reorder it via a left Σ_8-action with the element $(2\,8\,7\,3\,4)$. Note $sign(2\,8\,7\,3\,4) = (-1)^4 = 1$ and so the sign of G remains the same (Figure 2.9(b)). We collapse each subforest and its "cap" to get a planar embedding of a graph with $5 = 2*3-1$ edges between two vertices and an orientation at each vertex given by the forest ordering (Figure 2.9(c)). Finally, we "comb" the edges of our graph to give the standard planar embedding of this graph; this is achieved by permuting (relative to the bottom vertex), the first and second, the second and third, the fourth and fifth, and the third and fourth edges (Figure 2.9(d)). Since this required an even number of moves, the sign of the graph remains the same and the resulting trace of G is $\tau_8(G, \Phi) = 1$.

Question 2.7.5. For which values of k is $H^{4k-4}(Out(F_{2k}))$ nonzero?

Positive results have been reached in the cases $k = 1$, $k = 2$ [Vogtmann], $k = 3$ [CV, Ohashi]; it is conjectured that it is nonzero for all k. A new case ($k = 4$) will be shown to hold in Chapter 3.

Theorem 2.7.6. *(Conant-Vogtmann) Let $f\mathcal{G}^{(r)}$ denote the subcomplex of $f\mathcal{G}$ spanned by connected forested graphs of rank r. Then $H_k(f\mathcal{G}^{(r)}) \cong H^{2r-2-k}(Out(F_r); \mathbb{Q})$.*

It should be further noted that the above theorem specializes to the subcomplex $\widetilde{f\mathcal{G}}^{(r)}$ of forested graphs without separating edges so that $H_k(\widetilde{f\mathcal{G}}^{(r)}) \cong H^{2r-2-k}(Out(F_r)); \mathbb{Q}$.

If we wish to show the cohomology group $H^{4k-4}(Out(F_{2k}))$ is nonzero, we must first exhibit a non-vanishing generator for $H_k(\widetilde{f\mathcal{G}}^{(2r)})$. As it turns out, the trace map fits the bill.

Lemma 2.7.7. *τ_{4k-4} is a cocycle: $\tau_{4k-4}(\partial(G,\Phi)) = 0$.*

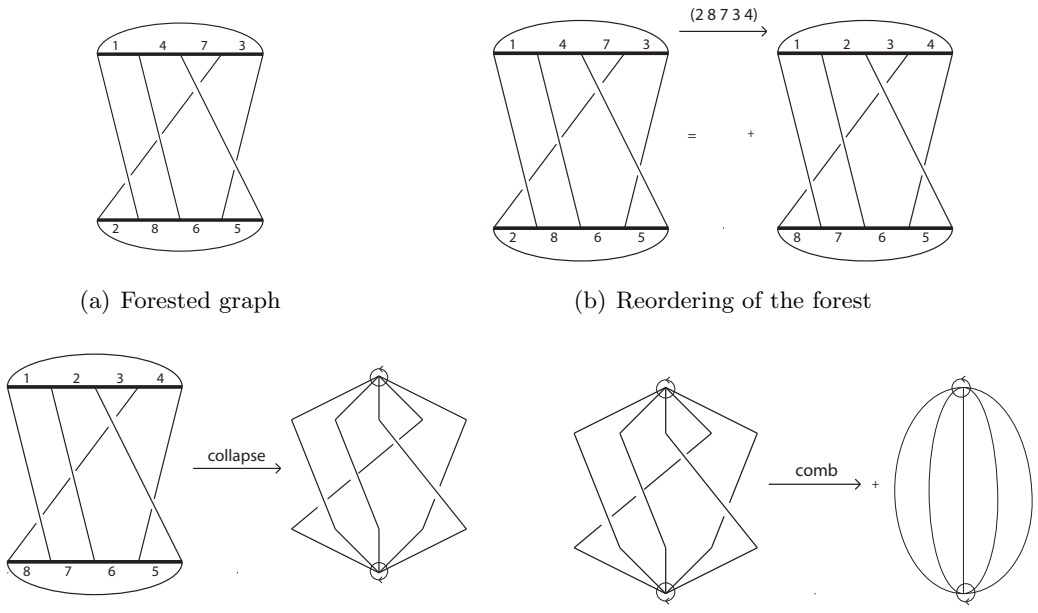
Proof. Let (G,Φ) be a forested graph of rank $2k$. It suffices to consider the following two cases as all others will have zero trace:

1. $\Phi = \Phi_1 \cup \Phi_2$ where Φ_1 has $2k-2$ edges and Φ_2 has $2k-3$ edges
2. $\Phi = \Phi_1 \cup \Phi_2 \cup \Phi_3$ where Φ_1 has $2k-2$ edges and Φ_2 together with Φ_3 has $2k-3$ edges.

The boundary of a type 1 graph has terms that either force Φ to have a trivalent vertex, join the endpoints of Φ_1 and Φ_2, or increases the linear length of Φ_1 or Φ_2 by one. The first two cases vanish under the trace and if the length of Φ_1 is increased by one, then each of the subforests are not of length $2k-2$ and hence the term is traceless. We are thus left with the case that the boundary splits Φ into two components of length $2k-2$ with nonforested edges joining the vertices of the subforests. Note that the trace of the resulting boundary will have two canceling terms in this case since the added edge can either "precede" or "follow" the subforest Φ_2 and the trace of each respective term will differ in sign.

As in the type 1 case, the remaining pair of terms of concern after the boundary is applied are the two that the edge addition joins the endpoints of Φ_2 and Φ_3. The sum of these two terms will vanish upon application of the trace because the number of edges in Φ_2 and Φ_3 necessarily have opposite parity. □

Let us show directly that $H^0(Out(F_2)) \cong \mathbb{Q}$. The space $\widetilde{f\mathcal{G}}_0$ is generated by the graph G_1 shown in Figure 2.1(a) and if we designate any edge of G_1 as forested, then the resulting [forested] graph G generates $\widetilde{f\mathcal{G}}_1$. If we apply an IHX relation to the forested edge of G, the result is a sum of three graphs, two of which are isomorphic to G and the third which



(a) Forested graph

(b) Reordering of the forest

(c) Collapsing cycles

(d) Combing the graph

Figure 2.9: The trace of a forested graph

possesses a separating edge (and is equivalent to the graph in Figure 2.1(c)), hence is zero (recall $\widetilde{f\mathcal{G}_i}$ is defined to be $f\mathcal{G}_i$ modulo graphs with separating edges). Thus $2G = 0$ and we have $\widetilde{f\mathcal{G}}_1 = 0$. It now follows that $H_0(\widetilde{f\mathcal{G}}_0) \cong \mathbb{Q}$ and so $H^0(Out(F_2)) \cong \mathbb{Q}$.

In the case $k = 2$, there are 5 trivalent graphs of rank 4 without separating edges (see Figure 2.4 of [11] for the explicit generators). Any maximal tree in one of these graphs will have $5 = 2 * 4 - 3$ edges (of the 9 total edges) and so there are approximately 630 possible forested graphs. It is evident that the case of $k = 2$ is already reaching the limits of hand-calculation. The determination of a cycle on which the trace does not vanish is not entirely obvious and examination of the $k = 3$ case is even more ghastly. Thus we consider the quotient $\mathcal{CG}_{2k} = f\mathcal{G}_{4k-3}/\partial(\ker \tau_{4k-4})$. The problem of finding a nonvanishing cycle under the trace has now been transformed into finding a sufficient number of relations on the forested graph complex so as to kill the space $\mathcal{CG}_{2k}$. To see that we can procure such a cycle if $\mathcal{CG}_{2k}$ vanishes, note that if γ is a graph with nonzero trace, then $\mathcal{CG}_{2k} = 0$ gives $f\mathcal{G}_{4k-3} = \partial(\ker \tau_{4k-4})$ and so $\partial(\gamma) \in \partial(\ker \tau_{4k-4})$. Thus the boundary of γ can be realized as the boundary of a traceless element γ'. If we let $G = \gamma - \gamma'$, then $\tau_{4k-4}(G) = \tau_{4k-4}(\gamma - \gamma') = \tau_{4k-4}(\gamma) \neq 0$ hence the cycle G does not vanish under the trace.

Chapter 3

The non-vanishing of the homology of $Out(F_8)$

Recall that elements of the top-dimensional chain group for $Out(F_r)$ can be realized as connected trivalent graphs without separating edges with an ordered maximal forest. Conant, Gerlits, Hatcher, Ohashi, and Vogtmann [8], [11], [17],[32] have calculated the cohomology $H^i(Out(F_r))$ of this complex with success for $i \leq 6$ and all r. To make the calculations more tractable, we discuss a reduction of our generating set of forested graphs to the set of chord diagrams which we subsequently reduce to the set of good chord diagrams. From here on, we will assume all forested graphs are without separating edges and hence specialize to the reduced case; there is no loss of generality in this case as each complex equivalently calculates the cohomology of $Out(F_r)$.

3.1 Chord diagrams

In [8], Conant and Vogtmann utilized a generating set for the reduced forested graph complex that simplified calculations greatly:

Definition 3.1.1. A *chord diagram* is a forested graph such that the forest is linear and there is a non-forested edge (called a *transversal*) of the graph connecting the endpoints of the linear forest. The remaining edges of the graph are called *chords*.

It is important to note that a chord diagram still carries the orientation associated to the ordering of the linear forest.

Recall that the IHX relation in the Lie operad (Figure 2.6(c)) is the *signed* sum of the three blowups of a 4-valent vertex is zero. This relation carries over to the forested graph

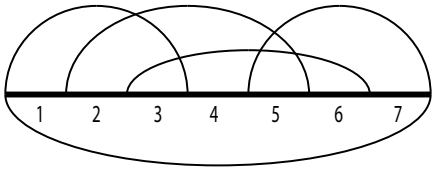


Figure 3.1: A chord diagram

(a) IHX for chord diagrams

(b) Shorthand for HX

Figure 3.2: IHX/HX for chord diagrams

complex and hence to chord diagrams as an *unsigned* sum, see Figure 3.2(a). To see how an $I - H + X$ relator in the Lie operad turns into an $I + H + X$ relator in the space of forested graphs, one must analyze the proof given for Theorem 2.7.6 in [7].

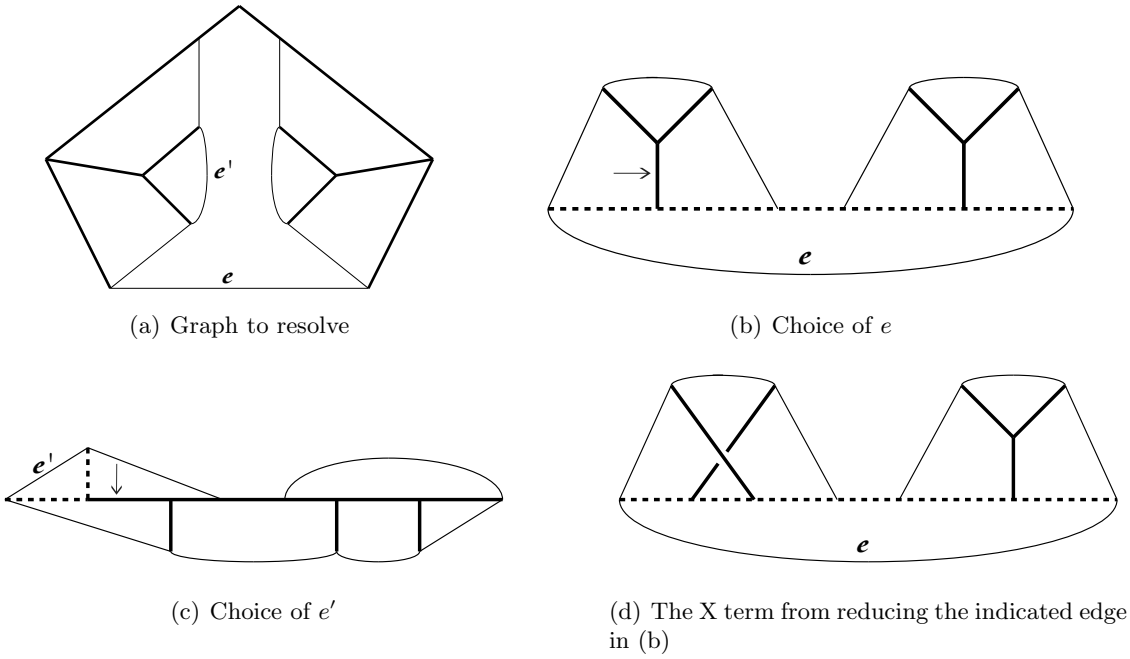
Define a filtration on the spine K_r of Outer space by the number of vertices in a marked graph together with the subcomplex spanned by the previous stage of the filtration (the initial filtratio is given by all trivalent connected marked graphs of rank r). Note that the action of $Out(F_r)$ preserves the filtration and therefore induces a filtration on the quotient of K_r by the $Out(F_r)$ action. It can then be shown that the spectral sequence associated to the quotient filtration collapses to a cochain complex which calculates the homology of the forested graph complex. Recall that a coboundary operator associated to the cubical complex is the operator δ that expands an edge in a marked graph. By taking the coboundary of a particular cube face in K_r and then passing to the quotient with $Out(F_r)$, the result is a sum of three terms which corresponds to an $I + H + X$ relator in $im(\delta)$. Noting that the kernel of δ coincides with [the dual of] the space of marked graphs modulo the antisymmetry relation together with the canonical isomorphisms of the latter spaces with their duals, the isomorphism follows.

Using the IHX relation (in $f\mathcal{G}$), we can show that the subspace $\mathfrak{C}$ of chord diagrams generates the space of forested graphs.

Lemma 3.1.2. *The space of forested graphs is generated by chord diagrams.*

Proof. Suppose (G, Φ) is a forested graph. If the forest Φ is linear (i.e., all vertices of Φ are bivalent), then we are done since an edge (there can be two) joining the endpoints of Φ will form the transversal and the remaining edges will necessarily intersect segments of Φ.

Suppose that Φ is nonlinear and let $e = (v, v')$ be an edge of $G - \Phi$. Since Φ is a maximal tree, any two vertices are part of the forest and hence there exists a shortest path p in Φ joining v and v' (the dashed edges in Figures 3.3(b-d)). Note that Φ is not linear and so there must be a trivalent edge of Φ along p, otherwise Φ would be linear (the arrowed edges in Figures 3.3(b,c)). If we apply an IHX relation to the first edge that protrudes from p (that is not a part of p), the result is a sum of two diagrams where the length of p has increased by one (as in Figure 3.3(d)) and p still joins v and v'. Again, travel along p in search of trivalent vertices in Φ. If no such vertex exists, we are done. Otherwise, proceed as before by lengthening p one edge at a time until p encompasses Φ, i.e., Φ is linear. □



(a) Graph to resolve

(b) Choice of e

(c) Choice of e'

(d) The X term from reducing the indicated edge in (b)

Figure 3.3: Resolving a forested graph into a linear combination of chord diagrams

3.2 Relations of chord diagrams

In §2.7, we discussed a method to trivialize the space $f\mathcal{G}_{4k-3}/\partial(\ker \tau_{4k-4})$ by generating relations that result from taking the boundary of elements in the kernel of the trace τ_{4k-4}. We explore this concept further by describing a series of relations on the space of chord diagrams that result from the trace map and the IHX relation.

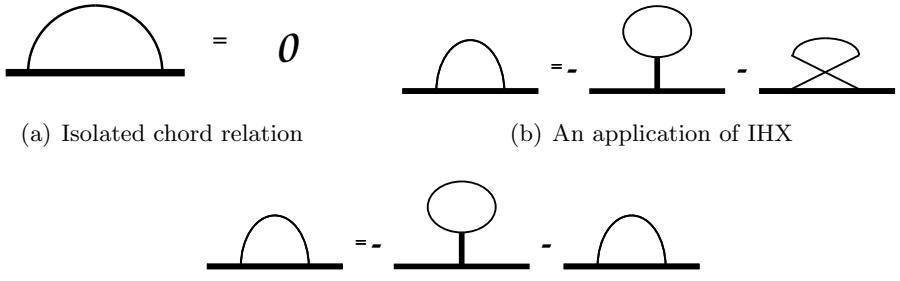
3.2.1 IHX-type relations

We start with relations that only require the IHX relation. The first relation allows one to choose any particular chord in a chord diagram and subsequently produce a sum of diagrams where this chosen chord is the transversal. The proof is very similar to the preceding lemma that shows the space of forested graphs is generated by chord diagrams.

Lemma 3.2.1. *(Transversal Permutation) Let C be a chord in the diagram D. Then D can be written as a sum of chord diagrams D_i where C is now the transversal of each D_i.*

Proof. Let D be a chord diagram with a specified chord C as given in the statement. Assuming C is not a transversal, reposition the diagram so that pieces of the forest protrude from the endpoints of C. Apply an IHX relation to the forest edge incident to an endpoint of C to create two a sum of two diagrams where the sprouting edge has been reduced by one and the C now spans one more edge. Repeat this process until there are no remaining sprouting edges in the sum. □

It turns out that diagrams that possess chords of an extreme type vanish, i.e., diagrams with a chord that encompass all of the forest or a single edge of the forest. The latter type are called *isolated chords*, see Figure 3.4(a).



(a) Isolated chord relation

(b) An application of IHX

(c) Untwisting the chord from (b)

Figure 3.4: The isolated chord relation

Lemma 3.2.2. *(Isolated Chord Relation) If a chord joins the endpoints of a single edge in the forest, then the diagram is zero.*

Proof. Apply an IHX relation to the single forested edge spanned by the isolated chord to get a sum (of the negation) of two terms; one of which is the original diagram and the other is a diagram with a separating edge, see Figures 3.4(b,c). Recall that diagrams with a separating edge vanish and so that diagram is zero. We are then left with two identical diagrams [which coincide with our original diagram] whose sum is zero. Since the characteristic of $\mathbb{Q}$ is not two, we determine that the chord diagram is zero. □

3.2.2 Trace-type relations

We now turn our attention to diagrams that require more than an application of the IHX relation and we consider the boundary of traceless diagrams.

Recall from §2.7.2 that one can create a relation by considering the boundary of traceless diagrams (i.e., elements in $\ker \tau_{4k-4}$). For the benefit of the reader, we record the definition of the cocycle τ_{4k-4} in the manner in which it will be utilized

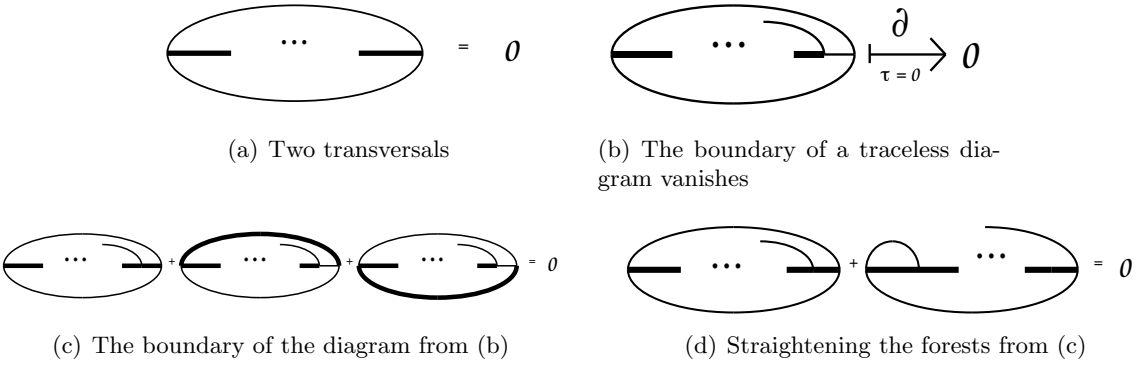
Definition 3.2.3. Suppose (G, Φ) is a forested graph with $4k-4$ edges with G of rank $2k$, then $\tau_{4k-4}(G, \Phi) = 0$ provided at least one of the following conditions does not hold:

1. Φ is the disjoint union of two linear trees Φ_1 and Φ_2,
2. Both Φ_1 and Φ_2 have $2k-2$ edges each,
3. There are non-forested edges connected the ends of Φ_1 and Φ_2, and
4. The vertices of Φ_1 and Φ_2 and remaining non-forested edges of G form a bipartite graph.

Lemma 3.2.4. *(Double Transversal Relation) If a chord diagram has two transversals, then the diagram is zero.*

Proof. Let D be a diagram with a chord that spans the entire forest, but remove the forest designation from the last edge in the forest, i.e., the diagram is of codimension 1.

Note that this diagram is traceless because the forest is not the disjoint union of two subforests as it is linear, see Figure 3.5(b). The boundary of the graph in Figure 3.5(b) is a sum of three terms shown in Figure 3.5(c).



(a) Two transversals

(b) The boundary of a traceless diagram vanishes

(c) The boundary of the diagram from (b)

(d) Straightening the forests from (c)

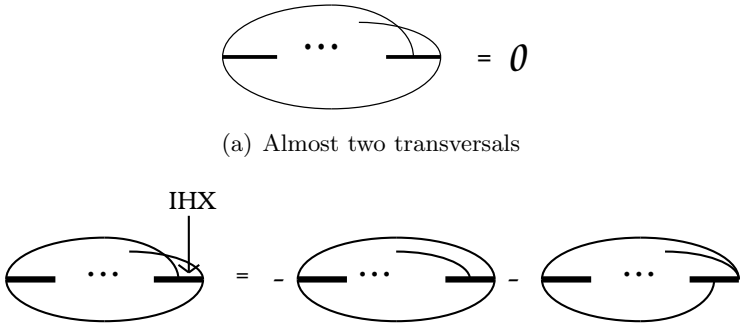
Figure 3.5: The double transversal relation

Inspection of the last two terms in the sum reveals the presence of a chord that "complements" the newly forested edge. This edge is of length one and so it creates an isolated chord. As a result, these two diagrams are zero in the sum and the result follows. □

With this relation, we can now say *the* transversal with impunity. The two extreme chord types have been shown to cause the vanishing of their corresponding diagram. One may hope that a diagram which possesses a chord that either spans two edges of the forest or all but one edge of the forest will vanish, as well. While it is unknown if a two-spanning chord in a diagram causes it to vanish, it can be shown that a diagram that has "almost two" traversals does vanish. We comment that, strictly speaking, this is not a trace-type relation but rather is of the IHX-type.

Lemma 3.2.5. *(Almost Double Transversal Relation) If a chord spans all but one edge of the forest, then the diagram is zero.*

Proof. Let D be a diagram with a chord C that spans all but one edge of the forest. Apply an IHX relation to the one edge of the forest that is not spanned by the chord C. Note that the X term returns the original diagram and the I term is the same as the X term once the transversal is realized. The result is then a sum of three terms where the I and X terms coincide and the H term vanishes by the double transversal relation. □



(a) Almost two transversals

(b) An application of IHX

Figure 3.6: The almost double transversal relation

We now discuss a relation that involves multiple chords and yields multiple terms in a relation. A pair of chords is said to be *parallel* if they both start and terminate at adjacent vertices and do not cross one another.

Lemma 3.2.6. *(Parallel Chord Relation) Suppose a diagram has a pair of parallel chords. Then the sum of the original diagram and the diagram where one pair of the endpoints are interchanged is zero.*

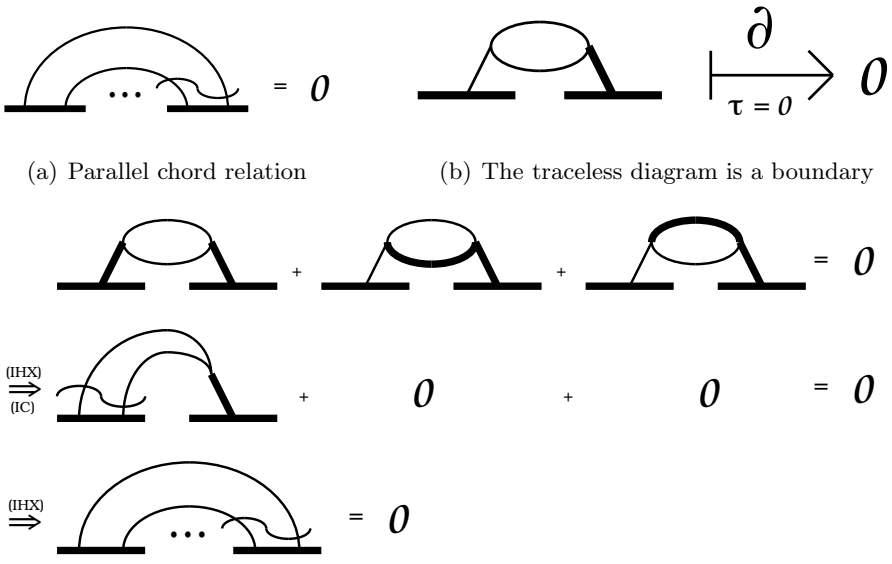
Proof. Note that an attached bigon (of codimension one) is traceless and the resulting sum of three terms from the boundary of the diagram vanishes. Two of the terms in the sum have an isolated chord and hence vanish, see Figure 3.7(c). If we apply IHX relations to the two sprouted forested edges, then we get a sum of four terms equal to zero. As two pairs coincide, we get the sum of a diagram with parallel chords and one with permuted endpoints is zero. □

3.2.3 The space of good chord diagrams

Note that a chord diagram of rank $n+1$ necessarily has $2n$ vertices and n chords (recall we disregard the transversal in the counting of chords). Thus there are exactly

$$\begin{aligned} p_n &= (2n)!/2^n n! \\ &= (2n-1)!! \\ &= 1 \cdot 3 \cdot 5 \cdots (2n-1) \end{aligned}$$

possible pairings of the vertices and hence p_n diagrams with n chords. Examination of the formula for p_n makes it immediately clear that the size of these spaces grows at an alarming rate as exhibited in the second column of Table 3.1. It would be quite nice if we could



(a) Parallel chord relation

(b) The traceless diagram is a boundary

(c) Manipulation of (b)

Figure 3.7: The parallel chord relation

further reduce our generating set for $f\mathcal{G}_n$ as the next unknown case of $n = 7$ has more than 10^5 diagrams. A possible guess for a generating subspace would be diagrams where the chord endpoints lie on opposite sides of the diagram's midpoint, to wit:

Definition 3.2.7. A chord diagram is *good* if each chord crosses the midpoint of the forest. The subspace of good chord diagrams will be denoted $\mathfrak{G}_n$. If a diagram has a chord which does not cross the midpoint, the diagram is called *bad*.

Note that if it is in fact the case that the space $\mathfrak{G}_n$ generates $\mathfrak{C}_n$, then the reduction in size of the spaces is staggering since the size each of the spaces $\mathfrak{G}_n$ is $n!$. To see this note that if we attach only the "starting vertex" of any chord to the first n vertices of our forest Φ, then there are precisely $n!$ permutations (i.e., ways of attaching the terminus of each edge) of the chord "ending vertex."

Table 3.1: The size of the spaces $\mathfrak{C}_n$ and $\mathfrak{G}_n$

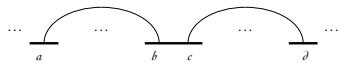
n	$\lvert\mathfrak{C}_n\rvert$	$\lvert\mathfrak{G}_n\rvert$
1	1	1
2	3	2
3	15	6
4	105	24
5	945	120
6	10395	720
7	135135	5040
8	2027025	40320
9	$\approx 3 \times 10^7$	362880
10	$\approx 6 \times 10^8$	3628800

A pair of chords $C_1 = (a, b)$ and $C_2 = (c, d)$ are *adjacent* if $c = b + 1$ and $a < b < c < d$. As an example, the chords that intersect the forest edge labelled "4" in Figure 3.1 are adjacent (or Figure 3.8(a)). The following relation allows us to take a pair of adjacent chords and expand them into a sum of diagrams where all of the chords either remain the same in length or increase in length.

Proposition 3.2.8. *(6T Relation) The sum of the six ways of permuting the endpoints of a pair of adjacent chords is zero.*

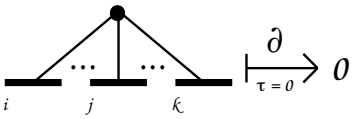
Proof. If one joins three non-forested edges to an exterior vertex which is regarded as the part of the forest, the result is a traceless diagram (Figure 3.8(c)). The boundary of this diagram is a sum of three terms which correspond to the three non-forested edges we started out with (Figure 3.8(d)). If we apply an IHX relation to the sprouting forested edge in each boundary component we get a sum of $3! = 6$ terms that vanish and which correspond to the permutations of the chord endpoints (Figure 3.8(e)). Recall that the addition of an edge to the forest via the boundary map causes the forest labeling of the new edge to occur last in the order. A shuffling of the forest labels gives the signs as indicated in Figure 3.8(f) □

Corollary 3.2.9. *(Maximally Adjacent Chords) Suppose $C_1 = (1, k)$ and $C_2 = (k + 1, 2n)$ are chords in the n-chorded diagram D. Then $D = -D'$ where D' is the chord diagram with C_1 and C_2 replaced by $(1, k + 1)$ and $(k, 2n)$, respectively.*

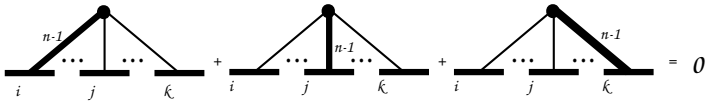


(a) Adjacent chords

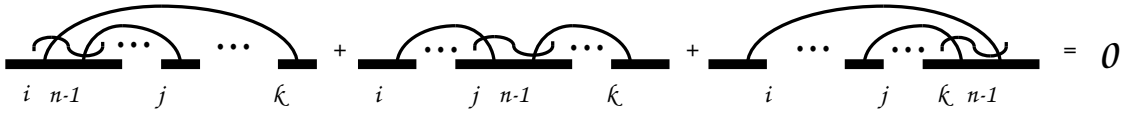
(b) The six permutations of the chord endpoints



(c) The traceless diagram of consideration



(d) The boundary of the diagram from (c)



(e) Three applications of the IHX relation

(f) Reordering the forest

Figure 3.8: The 6T relation

Proof. Apply the 6T relation to the adjacent chords C_1 and C_2 to get a sum of five negatively signed chord diagrams. Four of these diagrams contain either a second transversal or almost two transversals and are hence zero. The remaining two diagrams are precisely D and D' as described in the statement. □

It turns out that our aspiration that the space of good diagrams generates the space of chord diagrams is true.

Lemma 3.2.10. *The space of chord diagrams is generated by good chord diagrams. Thus the space of forested graphs is generated by good chord diagrams.*

Proof. Let D be a chord diagram. If all of the chords of D are good, we are done. Suppose then that D has a bad chord C. It suffices to show that any bad diagram can first be reduced to a diagram where the bad chord C is adjacent to another bad chord with the adjacency occurring at the midpoint of the forest.

To see that this is sufficient, consider the case where C falls next to the forest midpoint and is adjacent to another bad chord. Then an application of the 6T relation to this pair yields five diagrams in which this pair now passes the midpoint of the forest.

Suppose now that the bad chord C is adjacent to a good chord. A 6T relation applied to the pair will push C at least one edge closer to the midpoint. In the event that a chord incident to the midpoint started out good but turned bad after the 6T application, note

that it remains adjacent to the midpoint and so upon further application of the 6T relation to our diagram it will eventually be adjacent to the newly bad chord and hence falls under the first scenario. If a chord incident to the midpoint is bad, then it is now good and we have increased the number of good chord by two. In either case, the process terminates to yield a linear combination of diagrams which all have chord passing the forest midpoint. □

Let us verify Question 2.7.5 for $k = 1, 2, 3$ using Lemma 3.2.10. As a matter of convenience, we may represent a chord diagram (with n chords) as an ordered list of vertex endpoint labels. The diagram corresponding to the list +[1 4 2 6 3 7 5 8] is given in Figure 3.1.

In §2.7.2 we determined $H^0(Out(F_2))$ was nontrivial by a direct computation. For the sake of completeness, let us verify this fact with the new tools at hand. The space of good diagrams in dimension one is 1-dimensional and is generated by the theta graph. Depending on how one interprets the graph, it either has an isolated chord or two transversals. In any event, the graph generates $\partial(\ker \tau_{4k-4})$ and so we have

Theorem 3.2.11. $H^0(Out(F_2)) \neq 0$.

The cases $k = 2, 3$ require a bit more work:

Theorem 3.2.12. *(Vogtmann)* $H^4(Out(F_4)) \neq 0$

Proof. In rank four, there are $3! = 6$ possible good chord diagrams:

$$\begin{array}{cc} [1\ 4\ 2\ 5\ 3\ 6] & [1\ 4\ 2\ 6\ 3\ 5] \\ D_1 & D_2 \\ \\ [1\ 5\ 2\ 4\ 3\ 6] & [1\ 5\ 2\ 6\ 3\ 4] \\ D_3 & D_4 \\ \\ [1\ 6\ 2\ 4\ 3\ 5] & [1\ 6\ 2\ 5\ 3\ 4] \\ D_5 & D_6 \end{array}$$

By observation, we note D_2, D_3, and D_4 have almost two transversals while D_5 and D_6 possess double transversals hence are all zero. Therefore the only remaining diagram is D_1. The chords (1,4) and (2,5) form the "crossed" part of a parallel chord relation and so

$$\begin{aligned} D_1 &= -[1\ 5\ 2\ 4\ 3\ 6] \\ &= -D_3 \\ &= 0. \end{aligned}$$

□

Theorem 3.2.13. *(Conant, Vogtmann, Ohashi)* $H^8(Out(F_6)) \neq 0$

Proof. There are a total of 120 good diagrams in $\mathfrak{G}_5$. Once we account of double transversals and almost two transversals which instantiate at the first forest vertex (note that the latter are mutually exclusive), 72 diagrams remain and if we now allow for our almost two transversals chord to terminate at the end of the forest, we have 54 diagrams left. After an application of the isolated chord relation, the space is reduced to 46 diagrams $(24 - 3 \cdot 6 + 2)$.

Lastly, we apply the parallel chords relation to reduce the space down to six diagrams, four of which are equal due to an orientation reversing automorphism. The four remaining diagrams are [1 8 2 6 3 10 4 7 5 9], [1 7 2 9 3 6 4 8 5 10], [1 8 2 6 3 9 4 7 5 10], and [1 7 2 9 3 6 4 10 5 8]. This remaining four-dimensional space is not as easy to trivialize by hand and requires a computer search for relations. The author has independently shown the nontriviality of this cohomology group via an independent calculation. □

3.3 $H^{12}(Out(F_8)) \neq 0$ (the case $k = 4$)

3.3.1 Explanation of computer code

To determine $H_*(f\mathcal{G}; \mathbb{Q})$, Python code was written to generate the spaces of chord diagrams $\mathfrak{C}_n$. After, relations are created via a generator which attaches graphs onto chord diagrams so as to create a traceless [forested] graph of codimension one. We have coined the term *monster* to describe the special attaching graphs which create these relations.

The monstered graphs are then passed to a program that resolves forested graphs into chord diagrams via the IHX relation. Finally, these chord diagrams are fed to a program that resolves a diagram into a linear combination of good chord diagrams.

Below is a list of the utilized code and a brief description of its purpose. The code (along with a more thorough explanation of the syntax, etc.) can be found in the appendix.

1. `diagram_gr.py`

   Generates all chord diagrams with n chords from the space of diagrams with $n-1$ chords. Takes as the base space the file `fg_2` with contents `1 2`.

2. `good_diagram_genr.py`

   All good diagrams from the file `fg_$` are filtered out by testing if each chord "straddles" the midpoint. The result is saved as `g$`.

3. `reformat_and_filter.py`

   For the space of good chord diagrams, it suffices to give a diagram by its endpoints alone. This has the computational advantage that the memory allocation is essentially halved. Because of this, the diagrams are reformatted so as to be given by their endpoints.

4. `mc_*#.py` where $* \in \{r, n\}$ and `#` is a positive integer

   In the calculations, two types of monster (see Figure 3.9) were utilized: Mo$\underline{r}$ita and $\underline{n}$ormal. In each case, `#` indicates the number of legs present on the monster, e.g. 3-legged normal monsters give rise to the 6T relation. A topological sort was performed for each monster type: we insert the legs of the monster into the linear forest subject to some order relations on the attachment vertices; this was done to eliminate duplication. For instance, in the case of attaching a 4-legged normal monster, permuting the two adjacent attaching legs give rise to a duplicate relation. Therefore if the vertices we are attaching to are labelled a, b, c, and d we would impose the conditions $a < b$ and $a < c < d$.

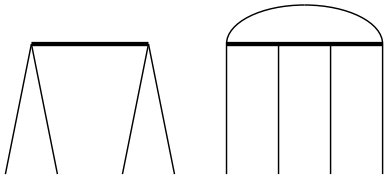


Figure 3.9: A normal monster and a Morita monster (each with four legs)

Once all the possible monsters of a particular type have been instantiated (i.e. we have fixed $*$ and `#`), for each case the unused vertices of the linear forest (called `lvnu`) are collected and passed to a function which creates a bijection between `lvnu` and the set $\{1, 2, \cdots, \#\texttt{lvnu}\}$ via the `dict` type. All possible chord diagrams are created with the set $\{1, 2, \cdots, \#\texttt{lvnu}\}$ and then translated back into the initial labeling via the dictionary.

After the possible monstered diagrams are enumerated, each diagram is operated upon by the boundary operator ∂_E to get a zero-sum of diagrams in the top dimension. The result is saved as the pickled python file `mo*#_$` where `*` and `#` are as above and `$` is a counter used to enumerate *all* possible monstered diagrams of type (`*, #`).

An additional feature of this program is that all relations involving isolated, long, and parallel chords among chord diagrams with `N` chords are created and stored to the files `ic_N`, `lc_N`, and `pc_N`. This is accomplished by reading in a dictionary of good chords and doing the requisite test for each relation.

Example 3.3.1. We illustrate the algorithm from the previous program by attaching a two-legged Morita monster to a diagram which will have five chords after the boundary operator is applied and hence the result gives rise to a relation in $\mathfrak{C}_5$. We start with eight vertices in the linear forest and choose all two element subsets of the eight vertices such that the first chosen vertex precedes the second (this is the topological sort). For the sake of argument, let our first vertex be 3 and the second 6. Then `lvnu`$= \{1, 2, 4, 5, 7, 8\}$ and we identify the latter with $\{1, 2, 3, 4, 5, 6\}$ of which there are $(2 \cdot 6)!/2^6 6! = 10395$ possible chord diagrams on the set. Once such pairing is $\{(1, 4), (2, 6), (3, 5)\}$ which under our dictionary is the pairing $\{(1, 5), (2, 8), (4, 7)\}$.

For the sake of computational ease, the forested edges that are part of the linear forest are not listed in the data. Hence the data passed to the boundary operator routine would consist of the list of forested edges `ft` and non-forested

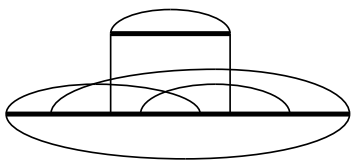


Figure 3.10: Attachment of a two-legged Morita monster

edges `nft` in the diagram. In the case of our example, `ft` $= \{(9,10)\}$, `nft` $= \{(1,5),(2,8),(3,9),(4,7),(6,10),(9,10)\}$, and the list of edges that will be forested after applying ∂_E are $\{(3,9),(6,10)\}$.

It should be noted that the calculations that have been made in this example and the following examples are correct up to a sign since most steps require a reordering of the forest, an application of the IHX relation, etc. and hence incur a sign change for the orientation. The interested reader is invited to peruse the heavily-commented code of each program to understand the sign conventions and how they are applied.

5. `str.py`

Given a diagram with a piece of the forest "growing" from the linear forest, we repeatedly apply the IHX relation to reduce the diagram to a sum of chord diagrams as in Lemma 3.1.3. Thus when the result of the monster-creator program `mc_*#.py` of item 4 is fed into this program, we get a linear combination of chord diagrams and hence a relation of chord diagrams. We comment that the program name `str` is for sprouting tree reducer for the visual feature that the tree sprouts from the forest and is afterwards reduced via the IHX relation.

It is clear by the definition of the IHX relation that a diagram with a piece of the nonlinear forest of length m will reduce to a linear combination of at most 2^m diagrams.

The result of processing a monster of type `*` with `#` legs is stored to

`~/mo*#relns/mo*#_reln_$`

where

- `*` is the monster type
- `#` is the number of monster legs
- `$` is the relation number for that particular kind of monster

Example 3.3.2. We carry on with our previous example and reduce our relation of ∂_E-operated diagrams to a linear combination of chord diagrams. After the application of ∂_E, we have a sum of two diagrams $D_1 = [\texttt{ft1}, \texttt{nft}]$ and $D_2 = [\texttt{ft2}, \texttt{nft}]$ where $\texttt{ft1} = \{(3,9),(9,10)\}$ and $\texttt{ft2} = \{(6,10),(9,10)\}$.

The diagram D_i is fed into a routine which determines the "leftmost" operable forest edge and an IHX relation is applied to the edge to get a sum of two diagrams where the number of forested edges not part of the linear forest is reduced by one. In the case of D_1, the first operated edge would be $(3,9)$ and a

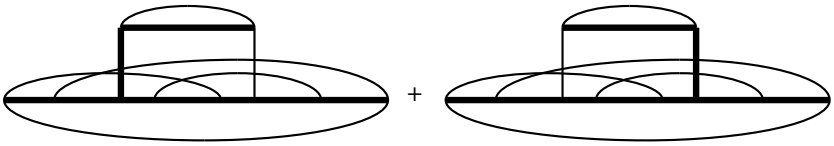


Figure 3.11: The boundary of the monstered diagram

single application of the IHX relation yields the sum of diagrams $D_{1,H} + D_{1,X}$ illustrated in the second row of Figure 3.12.

The second and final IHX application yields four diagrams $D_{1,HH} + D_{1,HX} + D_{1,XH} + D_{1,XX}$. The final relation after reducing D_1 and D_2 is shown in Figure 3.13.

6. `good_diag_enum.py`

   From the space of chord diagrams with `N` chords, the good chord diagrams are extracted, enumerated (according to a lexicographic ordering of the vertex list), and saved to the text file `genumN` in the format `'diagram|#'`, e.g. `'[5, 4, 6]|3'`. The pipe symbol appears merely as a forced partitioning character for the python string preprocessor `.split()`.

7. `bad_to_good.py`

   Since the space of chord diagrams is generated by the space of good chord diagrams (Lemma 3.2.10), we take each linear combination of diagrams determined from `str.py` and make it into a new linear combination of good diagrams. This is accomplished by defining a class called `Diagram` which carries the stubs `.g`, `.orien`, `.c`, and the subroutine `.get_chords()`. When the first term from a relation in the input file is read into `.g`, the coefficient is placed in the stub `.orien`, and then the subroutine processes the chord diagram to test if there is a chord which is bad. If there is a bad chord, the result is stored to the stub `.c`. Otherwise, the subroutine returns zero and the program knows that the diagram is then good.

   At runtime, the enumerated list generated by `good_diag_enum.py` is loaded and translated into the `dict` type so that the final linear combination of good chord diagrams can be converted into a row vector.

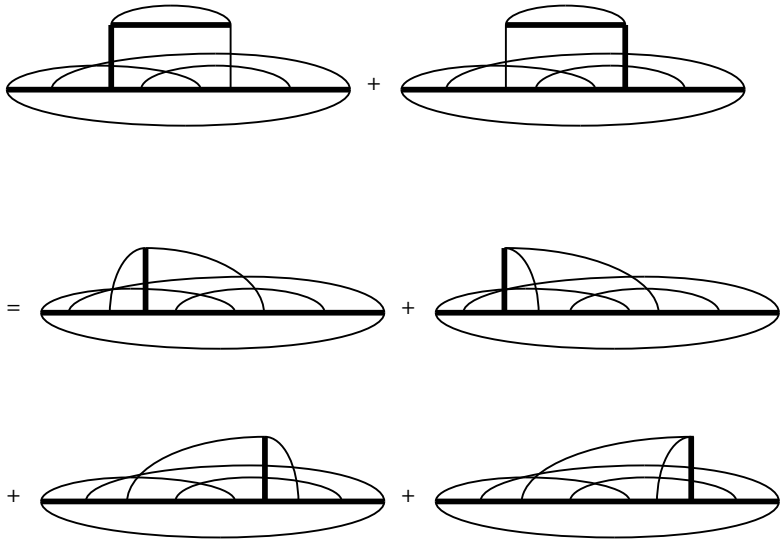


Figure 3.12: One application of IHX to the sum $D_1 + D_2$

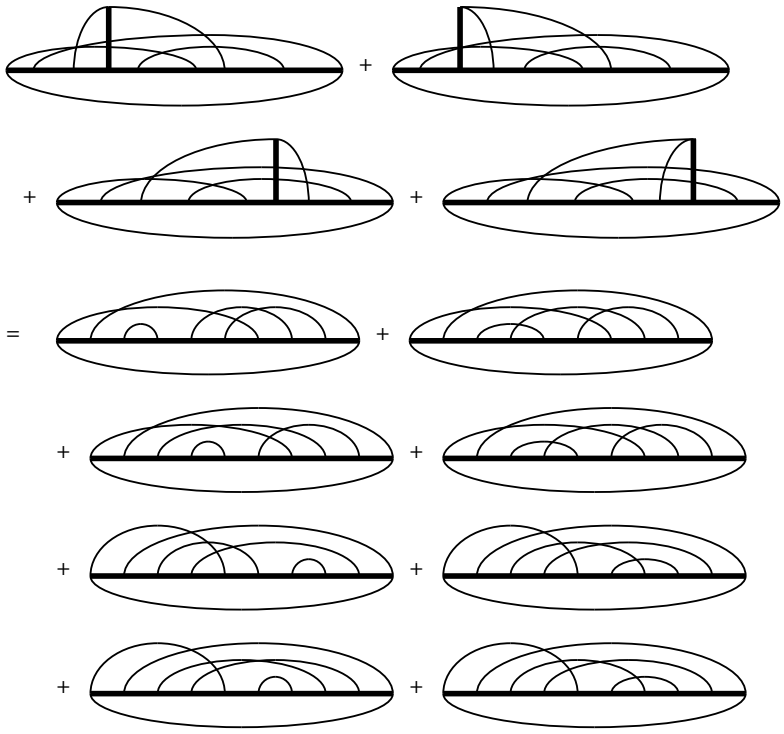


Figure 3.13: Two applications of IHX yields a sum of chord diagrams

In an effort to reduce computational intensity, after each application for the 6T relation, the resulting linear combination of diagrams is tested for isolated chords and long chords.

After the relation `mo*#_reln_$` from `str.py` has been fully processed, it is stored as a row vector in the plain text file `~/mo*#rows/r_$`.

Example 3.3.3. We reduce the diagram in the $(1, 2)$ array position, relative to the equality, in Figure 3.13. Note that the first bad pair is $(2, 4), (5, 9)$. A 6T relation applied to the chosen chords yields a sum of five chord diagrams that are all good (Figure 3.14).

8. `cat_rows.py`

   Processing the immense number of relations gives an equally large amount rows for our "relation matrix." To combine the rows generated by `bad_to_good.py`, a call to the UNIX command `cat` is utilized; the resulting matrix is stored as `rows$` where `$` is the number of chords.

3.3.2 Results, comments, and further directions

Upon generating a (hopefully) sufficient number of relations, the matrix generated in 8 is loaded into Octave and the routine `rank` is called to determine if our matrix is of full rank. In the event it is not, we process more relations by generating more monsters. As an indicator of the amount of data used to determine the $k = 4$ case, the matrix used had more than

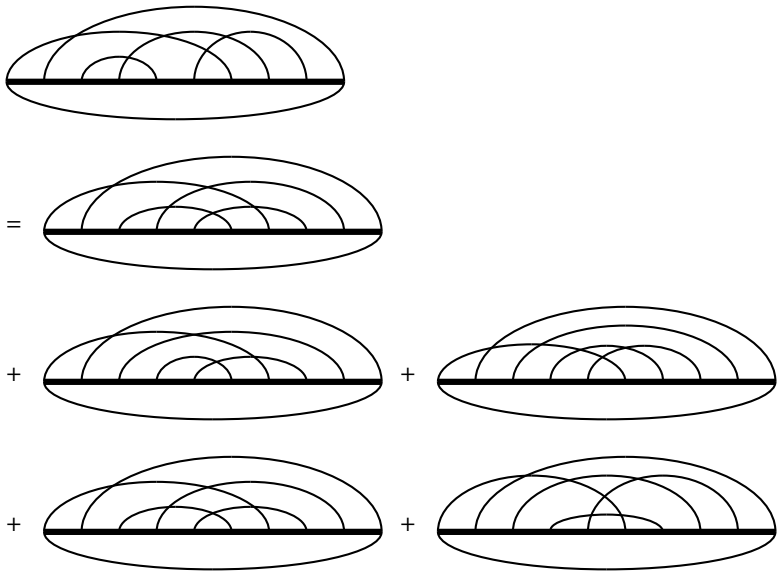


Figure 3.14: Reduction of a bad diagram via the 6T relation

40 millions entries and the [compressed] file size is over 100MB. The files used to determine that $\dim \partial(\ker \tau_8) = 5040$ can be found at `http://www.math.utk.edu/`gray/code/ .

Theorem 3.3.4. $H^{12}(Out(F_8)) \neq 0$

Proof. Using Octave, one can show that the subspace spanned isolated chord relations, parallel chord relations, and (almost) double transversal relations is not sufficient to trivialize the quotient. If we also allow for trace-type relations induced from three, four, and five-legged Morita monsters (see Figure 3.15), then we note that these relations *do* trivialize the quotient. That is, taking the relations generated by the Morita monsters along with the before-mentioned relations, one finds that the spanned subspace is of dimension 5040. Hence the Morita class is nontrivial and we may conclude the result. □

We comment that the code used to determine the nontriviality of the cohomology of *Out* can be applied to that of *Aut* in some cases if one does not utilize the almost double transversal relation [note that in the proof, the IHX relation is applied to the rightmost edge and the result causes the basepoint to now fall onto the chord hanging beneath the last diagram in Figure 3.6(b)].

The entirety of the code written was developed on the author's personal computer and then transferred to departmental servers for execution. The native version of Python on

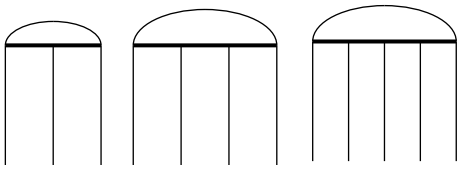


Figure 3.15: 3, 4, and 5-legged Morita monsters

the departmental servers was a version behind the original scripting ($2.6 \to 2.5$) and so speed and efficiency reductions were noticed. A portion of this was alleviated thanks to the generosity of M. Thistlethwaite; he allowed the author use of a personal workstation which allowed [immense] parallel computation.

The code as presented was optimized via bottlenecking utilities, but additional optimizations can be made. The most fruitful change to the code would be to create a "top-down" database of the bad chord diagrams which consists of its expansion as good chord diagrams. Using this database, it would be possible to explicitly describe the cocycle generator of $H^{12}(Out(F_8)) \neq 0$ which was nontractable due to computing constraints. Another avenue of improvement is to rewrite portions of the code in the hybrid Python-C language Cython; preliminary tests showed speed improvements of up to 10x.

Chapter 4

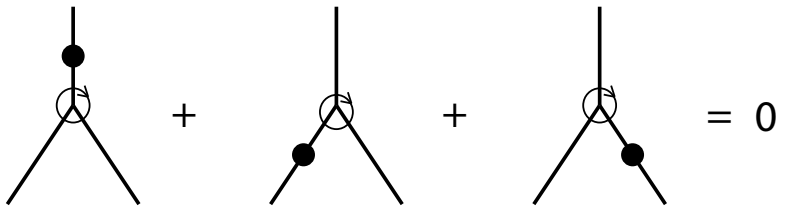
The Graph Homology of $Aut(F_n)$

4.1 The operad $p\mathcal{L}$

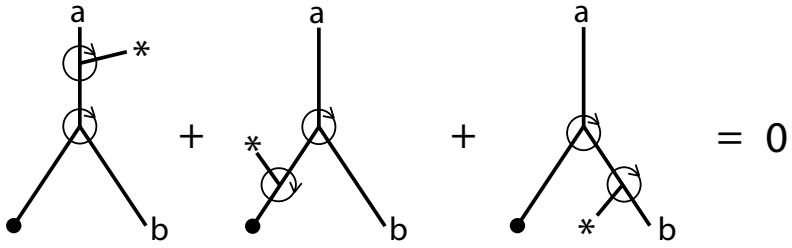
Recall the Lie operad is the cyclic operad where $\mathcal{L}[n]$ is generated by all Lie bracket sequences on $[n] = \{1, 2, \cdots, n\}$, or equivalently the space generated by all (rooted) binary planar trees with n numbered leaves modulo the relations AS and IHX. We wish to define a basepointed version of $\mathcal{L}$.

Given an element L from the Lie operad, we place distinguished basepoints on the tree. To mimic the IHX relation of $\mathcal{L}$, we impose the relation Aut-IHX illustrated in Figure 4.1(a).

It is useful to think of the basepoints on an operad element as sitting on the ends of "hairs" protruding from the basepoints of the binary tree. With this identification, we note that the Aut-IHX relation coincides with the IHX relation discussed earlier. To explain the identification, consider Figure 4.1(b).



(a) The Aut-IHX relation



(b) A hairy IHX relation

Figure 4.1: The IHX relation in the Aut case

With the hairs grown with the coherent orientation shown (the opposite coherent orientation of the grown edges differs by a global sign), we note this translates to the linear combination of bracket sequences

$$[[a,*],b]+[*,[a,b]]+[a,[b,*]]=0. \tag{4.1}$$

If we consider the standard IHX relation from $\mathcal{L}$, we have

$$\begin{aligned}0&=[[a,b],*]-[a,[b,*]]+[b,[a,*]]\\&=-[[a,*],b]-[*,[a,b]]-[a,[b,*]]\\&=[[a,*],b]+[*,[a,b]]+[a,[b,*]]\end{aligned}$$

which is precisely equation 4.1. Thus the Aut-IHX relation of hairy trees can be interpreted as the standard IHX relation in $\mathcal{L}$.

Proposition 4.1.1. *The species derived from adding distinguished basepoints to the elements of $\mathcal{L}$ modulo the Aut-IHX, AS, and IHX relations forms a cyclic operad denoted $p\mathcal{L}$.*

Proof. We show $p\mathcal{L}$ is a monoid in the category of species and hence is an operad. Define an operation on pointed species elements by the standard mating of their underlying Lie species structure, i.e. mate the spiders along a chosen leg pair and if the legs to be mated have l and m basepoints (respectively), then the edge formed from mating now has $l+m$ basepoints. The identity of this monoid is the same as that of $\mathcal{L}$. Note that a Σ_{n+1} action on the underlying Lie operad structure extends to a Σ_{n+1} action on the pointed counterpart and so $p\mathcal{L}$ is cyclic. In particular, if the basepoints are identified with hairs growing from the basepoint, the Aut-IHX relation becomes the standard IHX relation of $\mathcal{L}$ and hence the Aut-IHX subspace is preserved by the Σ_{n+1} action. □

4.2 The Lie module $p_1\ell_\infty$

In specializing our definitions from Chapter 2 to the operad $p\mathcal{L}$, we note that the basepoint/hair identification does not induce a labeling of the hair per the Σ_n-invariant construction. That is, we do not label the hair corresponding to the basepoint, nor do we "attach" an element from our symplectic vector space to the hair in the case of the symplectospider construction.

We define the *space of basepointed spiders* with n legs to be $p\mathcal{LS}[n]=\left(\bigoplus_{\mathcal{L}}\mathcal{O}[n-1]\right)_{\Sigma_n}$ and we define the *total space of basepointed spiders* to be $p\mathcal{LS}=\bigoplus_{n\geq 1}p\mathcal{LS}[n]$. The *Lie algebra of basepointed symplectospiders* is defined as in Definition 2.6.5 and will be denoted $p\ell_\infty$.

We write $p_k\ell_\infty$ to denote the subspace of $p\ell_\infty$ with symplectospiders possessing k basepoints. Care must be exercised for while $p\ell_\infty$ is a Lie algebra (since $p\mathcal{L}$ is a cyclic operad), it is not the case that our object of interest $p_1\ell_\infty$ is a Lie algebra. This is easy to see because mating preserves the number of basepoints: the mating (that is, the bracket) of two single-haired spiders would result in a two-haired spider which is not contained in $p_1\ell_\infty$. All is not lost, however, as is indicated in the following lemma.

Lemma 4.2.1. *The space $p_1\ell_\infty$ is a Lie ℓ_∞-module.*

Proof. In order for the vector space $p_1\ell_\infty$ to be a Lie ℓ_∞-module, we must show that there is a map

$$\begin{aligned} \ell_\infty \otimes p_1\ell_\infty &\stackrel{\cdot}{\longrightarrow} p_1\ell_\infty \\ \mathbb{S} \otimes p\mathbb{S} &\mapsto \mathbb{S} \cdot p\mathbb{S} \end{aligned}$$

that satisfies

$$[\mathbb{S}_1, \mathbb{S}_2] \cdot p\mathbb{S} = \mathbb{S}_1 \cdot (\mathbb{S}_2 \cdot p\mathbb{S}) - \mathbb{S}_2 \cdot (\mathbb{S}_1 \cdot p\mathbb{S}). \tag{4.2}$$

Let $\mathbb{S} \cdot p\mathbb{S}$ be defined by $[\mathbb{S}, p\mathbb{S}]$ where the bracket $[-,-]$ is inherited from $p\ell_\infty$. It is clear that $[-,-]$ is bilinear. Let $p\mathbb{S} \in p_1\ell_\infty$ and $\mathbb{S}_1, \mathbb{S}_2 \in \ell_\infty$, then by the Jacobi identity we have

$$[p\mathbb{S}, [\mathbb{S}_1, \mathbb{S}_2]] + [\mathbb{S}_1, [\mathbb{S}_2, p\mathbb{S}]] + [\mathbb{S}_2, [p\mathbb{S}, \mathbb{S}_1]] = 0.$$

A few applications of the antisymmetry property gives

$$-[[\mathbb{S}_1, \mathbb{S}_2], p\mathbb{S}] + [\mathbb{S}_1, [\mathbb{S}_2, p\mathbb{S}]] - [\mathbb{S}_2, [\mathbb{S}_1, p\mathbb{S}]] = 0$$

and then we note

$$[\mathbb{S}_1, \mathbb{S}_2] \cdot p\mathbb{S} - \mathbb{S}_1 \cdot (\mathbb{S}_2 \cdot p\mathbb{S}) + \mathbb{S}_2 \cdot (\mathbb{S}_1 \cdot p\mathbb{S}) = 0$$

is equation 4.2. Therefore $p_1\ell_\infty$ is a Lie ℓ_∞-module. □

Remark 4.2.2. The action of $p_1\ell_\infty$ on ℓ_∞ is very natural in light of Kontsevich's definition of ℓ_∞. Given a symplectic vector space (V_n, ω) with basis $\mathcal{B} = \{p_1, \cdots p_n, q_1, \cdots, q_n\}$ (see Definition 2.6.4), we let $\mathcal{L}_{2n}$ be the free Lie algebra generated by V_n. Recall that a derivation is a map $\mathcal{L}_{2n} \xrightarrow{D} \mathcal{L}_{2n}$ that respects the bracket of the Lie algebra:

$$D[x, y] = [Dx, y] + [x, Dy].$$

As such, the collection of derivations forms a Lie algebra with bracket $[D, D'] = D \circ D' - D' \circ D$ denoted $Der(\mathcal{L}_{2n})$. Given an element x of $\mathcal{L}_{2n}$, we note that we can define the adjoint ad_x by $ad_x(v) = [x, v]$ where $v \in \mathcal{L}_{2n}$. Defined in this way, the adjoint is a derivation. If we then consider the elements of $Der(\mathcal{L}_{2n})$ which kill the special element $\sum_i [p_i, q_i]$, then this is Kontsevich's Lie algebra ℓ_n. With the basepoint-hair identification, the space $p_1\ell_\infty$ is isomorphic to the free Lie algebra on $\mathcal{L}_{2n}$. Since ℓ_∞ is a Lie algebra of derivations, the standard action with the adjoint representation is $ad_X v = [X, v]$ where $X \in \ell_\infty$ and $v \in \mathcal{L}_{2n} \cong p_1\ell_\infty$. Note that this action of ℓ_∞ on $p_1\ell_\infty$ in the world of free Lie algebras is the translation of the action presented in Lemma 4.2.1 to the world of Lie modules.

To define the Lie algebra homology of ℓ_∞ with coefficients in $p_1\ell_\infty$ we must establish that $p_1\ell_\infty$ is a representation of ℓ_∞, that is, we must establish that $p_1\ell_\infty$ is an ℓ_∞-module. This is precisely what we showed in the preceding lemma and so we define the homology of the "Lie algebra" of singly basepointed spiders as $H_\bullet(\ell_\infty; p_1\ell_\infty)$.

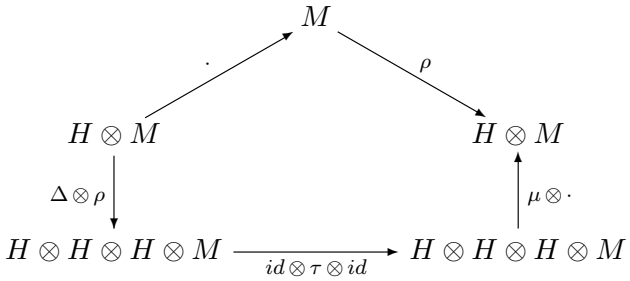
Recall from [7] that $H_\bullet(\ell_\infty)$ was shown to have the structure of a Hopf algebra. We recall the definition of a Hopf module from §4.1 of [34].

Definition 4.2.3. Let H be a Hopf algebra over the field k with multiplication μ and comultiplication Δ. Then the k-module M is called a *Hopf module over* H if

1. M is a left H-module with module action $h \cdot m$ for $m \in M$ and $h \in H$.
2. M is a left H-comodule with comodule action $M \xrightarrow{\rho} H \otimes M$ given by $m \mapsto m_{(0)} \otimes m_{(1)}$ for $m_{(0)} \in H$ and $m_{(1)} \in M$. (Note we are using the sumless Sweedler notation.)
3. The following compatibility relation holds

$$\rho(h \cdot m) = h_{(1)} \cdot m_{(0)} \otimes h_{(2)} m_{(1)},$$

or equivalently the following diagram commutes (the map $H \otimes H \xrightarrow{\tau} H \otimes H$ is the twist map defined by $x \otimes y \mapsto y \otimes x$).



We adjust these operations for the context of the Koszul complex and the result is

Lemma 4.2.4. *The space $H_\bullet(\ell_\infty; p_1\ell_\infty)$ is a Hopf module over the Hopf algebra $H_\bullet(\ell_\infty)$.*

Proof. We completely describe the product μ and coproduct Δ from [7]. In the Lie algebra ℓ_∞, the symplectospiders are decorated with elements from the symplectic basis $\mathcal{B}_\infty$. We define the two maps on $\mathcal{B}_\infty$

$$\begin{aligned} \mathcal{B}_\infty &\xrightarrow{E} \mathcal{B}_\infty & \qquad \mathcal{B}_\infty &\xrightarrow{O} \mathcal{B}_\infty \\ p_i &\longmapsto p_{2i} & p_i &\longmapsto p_{2i-1} \\ q_i &\longmapsto q_{2i} & q_i &\longmapsto q_{2i-1} \end{aligned}$$

The product on $H_\bullet(\ell_\infty)$ is then given by $S \otimes S' \mapsto E(S) \wedge O(S')$ where S and S' are wedges of spiders. We give the caveat that this product is associative only up to invariants, but this is fine as it was shown in [7] that the complex of invariants carries the homology of the full complex.

To define the coproduct, we consider unordered partitions of spiders. Given a wedge of spiders $S_1 \wedge \cdots \wedge S_k$, we define S_I to be the wedge $S_{i_1} \wedge \cdots \wedge S_{i_{|I|}}$ where $i_1 < \cdots < i_{|I|}$ and $I \subset \{1, \ldots, k\} = [k]$. Given an unordered partition $I \cup J = [k]$, we write $S_1 \wedge \cdots \wedge S_k \stackrel{?}{=} \epsilon(I, J) S_I \wedge S_J$ where $\epsilon(I, J)$ is the sign required to make this an equality. Given these definitions, the coproduct Δ is given by

$$S_1 \wedge \cdots \wedge S_k \mapsto \sum_{[k]=I \cup J} \epsilon(I, J) S_I \otimes S_J.$$

We are now in a position to define a module and comodule action for $\wedge\ell_\infty$ on $\wedge\ell_\infty \otimes p_1\ell_\infty$; they are completely analogous to the multiplication and comultiplication in ℓ_∞.

Let $S_1, \cdots, S_k, S_{k+1}, \cdots, S_t$ be Lie symplectospiders and let P be a pointed Lie symplectospider. The module action $\cdot$ will be given by

$$(S_1 \wedge \cdots \wedge S_k) \otimes [(S_{k+1} \wedge \cdots \wedge S_t) \otimes P] \mapsto E(S_1 \wedge \cdots \wedge S_k) \wedge O(S_{k+1} \wedge \cdots \wedge S_t) \otimes O(P)$$

while the comodule action ρ will be given by

$$S_1 \wedge \cdots \wedge S_k \otimes P \mapsto \sum_{[k]=I \cup J} \epsilon(I,J) S_I \otimes (S_J \otimes P).$$

We shall now verify that the Hopf module compatibility diagram commutes:

$$\rho(- \cdot -) = (\mu \otimes \cdot)(id \otimes \tau \otimes id)(\Delta \otimes \rho)(- \otimes -)$$

Let $(S_1 \wedge \cdots \wedge S_k) \otimes (S_{k+1} \wedge \cdots \wedge S_t \otimes P)$ be an element of $\wedge \ell_\infty \otimes (\wedge \ell_\infty \otimes p_1 \ell_\infty)$. Then

$$(S_1 \wedge \cdots \wedge S_k) \cdot (S_{k+1} \wedge \cdots \wedge S_t \otimes P) \overset{\rho}{\longmapsto} \sum_{I,J} \epsilon(I,J) S_I \otimes (S_J \otimes O(P))$$

where S_I and S_J consist of (wedges of) elements from $\{E(S_1), \cdots E(S_k), O(S_{k+1}), \cdots, O(S_t)\}$. This completes the left-hand side of the equality.

Turning to the right-hand side of the equality, we note

$$\begin{aligned} &\Delta(S_1 \wedge \cdots \wedge S_k) \otimes \rho(S_{k+1} \wedge \cdots \wedge S_t \otimes P) \\ &\qquad = \sum_{I',J',I'',J''} \epsilon(I',J')\epsilon(I'',J'') S_{I'} \otimes S_{J'} \otimes S_{I''} \otimes (S_{J''} \otimes P) \end{aligned}$$

which followed by $(id \otimes \tau \otimes id)$ gives

$$\begin{aligned} &\sum_{I',J',I'',J''} \epsilon(I',J')\epsilon(I'',J'') S_{I'} \otimes S_{I''} \otimes S_{J'} \otimes (S_{J''} \otimes P) \\ &= (\dagger) \end{aligned}$$

Finally, applying $(\mu \otimes \cdot)$ to $\dagger$ has the result

$$\sum_{I',J',I'',J''} \epsilon(I',J')\epsilon(I'',J'')(E(S_{I'}) \wedge O(S_{I''})) \otimes (E(S_{J'}) \wedge O(S_{J''}) \otimes O(P)).$$

Note that the partitions are selected in a manner so that the indices are strictly increasing. The consequence of this is that when one traverses the top of the pentagon for the compatibility diagram, the selection of "even and odd" colored elements is restricted so that if the partition $I \cup J$ is so that say the first element in I is an "odd" element, then *all* elements from the set I must be from the odd-colored component. Hence, once an index for an odd-colored element is selected for I it must be that the remaining elements elements are odd-colored. This exactly describes the above summand. It remains then to show the corresponding coefficients agree. To see this, note that the signs agree up to the last $t-k+1$ terms, but it is precisely the shuffle $\epsilon(I'', J'')$ along with the twist map τ that form the correction in sign. $\square$

4.3 The graph homology of $p_1\mathcal{L}$

From the space of basepointed spiders, we wish to create a graph complex as in Proposition 2.5.8.

Definition 4.3.1. Let G be an oriented graph with all vertices at least bivalent. If we denote the vertices of G by $v_1, \cdots, v_m$ and the respective valence of each vertex by $d_1, \cdots, d_m$, then a basepointed graph is an identification of each vertex v_i with a spider from $p\mathcal{LS}[d_i - 1]$.

Since $p\mathcal{L}$ is a cyclic operad, $p\mathcal{LG}$ is a $p\mathcal{L}$-graph complex and so we retain the same conventions as in Chapter 2 regarding the calculation of graph homology. As we are interested in the homology of $Aut(F_n)$, we focus on the situation when our basepointed graphs have a single hair and henceforth when we write "basepointed graph" it will be understood that the graph contains a single basepoint.

Denote the subspace of $p\mathcal{LG}$ consisting of graphs which possess a single basepoint by $p_1\mathcal{LG}$. This means the degree k part of $p_1\mathcal{LG}$ consists of a single vertex decorated by a pointed spider along with $k-1$ vertices decorated by non-pointed spiders. With this definition of $p_1\mathcal{LG}$, it is a subcomplex:

Proposition 4.3.2. *The subspace $p_1\mathcal{LG}$ is a subcomplex of $p\mathcal{LG}$.*

Proof. We must show ∂_E has degree -1, carries single basepointed graphs to single basepointed graphs, and $\partial_E^2 = 0$ on this subspace.

Let P be a pointed Lie graph in $p_1\mathcal{LG}_k$, that is, there are k vertices on the underlying oriented graph of P and exactly one of the k vertices is decorated with a pointed Lie spider with a single basepoint. Then $\partial_E(P)$ is the sum over all edge collapses of P and consequent mating of the spiders decorating the edge vertex endpoints, i.e. $\partial_E(P)$ is a linear combination of decorated graphs with $k-1$ vertices. Note that the mating of a Lie spider with a singly pointed Lie spider results in a singly pointed Lie spider and so, in particular, the edge collapse will not result in a graph with more than one basepoint nor zero basepoints (unless the collapses causes a loop in the quotient graph). The fact that $\operatorname{im} \partial_E \subset \ker \partial_E$ follows as in the proof of Proposition 2.5.8. □

We recall that if $\mathcal{O}$ is a cyclic operad, then $\mathcal{OG}$ has a Hopf algebra structure as detailed in Proposition 2.5.9: the product of two graphs is given by their disjoint union and a coalgebra structure is defined so that the connected graphs are the primitives. Since $p_1\mathcal{LG}$ is not a cyclic operad, we are led to the question: What kind of structure does the space $p_1\mathcal{LG}$ carry? Following in the trend for the space of pointed Lie symplectospiders we assert

Proposition 4.3.3. *The space $p_1\mathcal{LG}$ is a Hopf module over the Hopf algebra $\mathcal{LG}$.*

Proof. To establish the result, we must define a module action, a comodule action, and then verify a compatibility relation.

Define an $\mathcal{LG}$-module action $\mathcal{LG} \otimes p_1\mathcal{LG} \xrightarrow{\cdot} p_1\mathcal{LG}$ as in the $\mathcal{LG}$ case by disjoint union. That is, if G is a Lie graph and P is a pointed Lie graph, then $G \cdot P = G \sqcup P$ where $\sqcup$ is the disjoint union.

We define $p_1\mathcal{LG} \xrightarrow{\rho} \mathcal{LG} \otimes p_1\mathcal{LG}$ in the spirit of the operation Δ of the Hopf algebra $\mathcal{LG}$ where Δ is defined so that the connected graphs are the primitive elements. If P is a connected, basepointed Lie graph and 1 denotes the empty graph, then $P \xmapsto{\rho} 1 \otimes P$ and extend linearly over the product.

Let us verify the compatibility condition. Let G be a Lie graph and let P be a pointed Lie graph. Tracing along the top of the pentagon of the diagram in Definition 4.2.3,

$$\begin{aligned}\rho(G \cdot P) &= \rho(G \sqcup P) \\ &= 1 \otimes (G \sqcup P) + G \otimes P. \end{aligned} \tag{4.3}$$

On the other hand, going down the leftmost vertical map of the pentagon gives

$$\begin{aligned}(\Delta \otimes \rho)(G \otimes P) &= \Delta(G) \otimes \rho(P) \\ &= (G \otimes 1 + 1 \otimes G) \otimes (1 \otimes P) \\ &= G \otimes 1 \otimes 1 \otimes P + 1 \otimes G \otimes 1 \otimes P \\ &= (\dagger)\end{aligned}$$

and then traversing the bottom yields

$$\begin{aligned}(id \otimes \tau \otimes id)(\dagger) &= G \otimes 1 \otimes 1 \otimes P + 1 \otimes 1 \otimes G \otimes P \\ &= (\dagger\dagger)\end{aligned}$$

Finally, we go up the rightmost map of the pentagon to note

$$\begin{aligned}(\mu \otimes \cdot)(\dagger\dagger) &= (G \sqcup 1) \otimes (1 \sqcup P) + (1 \sqcup 1) \otimes (G \sqcup P) \\ &= G \otimes P + 1 \otimes (G \sqcup P)\end{aligned} \tag{4.4}$$

Note that equations 4.3 and 4.4 coincide; our compatibility condition is verified and therefore $p_1\mathcal{LG}$ is a Hopf module over the Hopf algebra $\mathcal{LG}$. □

Lemma 4.3.4. *The map*

$$\wedge\ell_\infty \otimes p_1\ell_\infty \xrightarrow{\psi_n} p_1\mathcal{LG}$$

is a chain map, that is, the following diagram is commutative.

$$\begin{array}{ccc} \wedge\ell_\infty \otimes p_1\ell_\infty & \xrightarrow{\psi_n} & p_1\mathcal{LG} \\ \Big\downarrow{\scriptstyle \partial_n} & & \Big\downarrow{\scriptstyle \partial_E} \\ \wedge\ell_\infty \otimes p_1\ell_\infty & \xrightarrow{\psi_n} & p_1\mathcal{LG} \end{array}$$

Proof. Let $S = S_1 \wedge \cdots \wedge S_n \otimes P \in \wedge\ell_\infty \otimes p_1\ell_\infty$. We show $\partial_E\psi_n(X) = \psi_n\partial_n(X)$ (recall ∂_E is the graph-complex boundary map and ∂_n is the boundary map for Lie algebra homology). Consider the left hand side:

$$\begin{aligned}\partial_E\psi_n(X) &= \partial_E \sum_\pi w(\pi)(S_1 \wedge \cdots \wedge S_n \otimes P)^\pi \\ &= \sum_e \sum_\pi w(\pi)(S_1 \wedge \cdots \wedge S_n \otimes P)^\pi_e\end{aligned}$$

where the latter sum is over all edges in the graph $(S_1 \wedge \cdots \wedge S_n \otimes P)^\pi$ with π a pairing of the half edges of the spider legs. We partition this sum into two classes: edges which have

a half-edge which is a leg of P and edges which do not.

$$\cdots = \sum_{e \in P} \sum_{\pi} w(\pi)(S_1 \wedge \cdots \wedge S_n \otimes P)^{\pi}_{e} + \sum_{e \notin P} \sum_{\pi} w(\pi)(S_1 \wedge \cdots \wedge S_n \otimes P)^{\pi}_{e}.$$

It was shown in [7] that the the second summand corresponds to

$$\sum_{i<j} (-1)^{i+j} [S_i, S_j] \wedge S_1 \cdots \wedge \widehat{S}_i \wedge \cdots \wedge \widehat{S_j} \wedge \cdots \wedge S_n \otimes P$$

under the image of ψ_n and so it remains to show the correspondence between

$$\sum_{e \in P} \sum_{\pi} w(\pi)(S_1 \wedge \cdots \wedge S_n \otimes P)^{\pi}_{e} \tag{4.5}$$

and

$$\psi_n \sum_{i=1}^{n} (-1)^i S_1 \wedge \cdots \wedge \widehat{S}_i \wedge \cdots \wedge S_n \otimes [S_i, P]. \tag{4.6}$$

If we expand (4.6) using the definition of the bracket and then apply the map ψ_n we have

$$\begin{aligned} &\psi_n \sum_{i=1}^{n} \sum_{\alpha \in S_i, \beta \in P} (-1)^i S_1 \wedge \cdots \wedge \widehat{S}_i \wedge \cdots \wedge S_n \otimes (S_i, \alpha)!(P, \beta) \\ &= \sum_{i=1}^{n} \sum_{\alpha \in S_i, \beta \in P} \sum_{\pi} (-1)^i w(\pi)(S_1 \wedge \cdots \wedge \widehat{S}_i \wedge \cdots \wedge S_n \otimes (S_i, \alpha)!(P, \beta))^{\pi}. \end{aligned} \tag{4.7}$$

We manipulate (4.5) to bring it into the form of (4.7). Note that the legs (half-edges) α and β comprise an edge of the graph $(S_1 \wedge \cdots \wedge S_n \otimes P)^{\pi}$ if $(\alpha, \beta) \in \pi$. Moreover, we may assume α and β are legs from distinct spiders for if they are not, the graph vanishes under ∂_E (recall that ∂_E is defined so that it vanishes on loops). With this in mind, we recast (4.5) in the following form (with the understanding all the edges e in the inner summand are necessarily incident to P):

$$\begin{aligned} &\sum_{\pi} \sum_{e \in (S_1 \wedge \cdots \wedge S_n \otimes P)^{\pi}} w(\pi)(S_1 \wedge \cdots \wedge S_n \otimes P)^{\pi}_{e} \\ &= \sum_{\pi} \sum_{(\alpha, \beta) \in \pi} w(\pi)(S_1 \wedge \cdots \wedge S_n \otimes P)^{\pi}_{\alpha \cup \beta} \\ &= \sum_{\pi} \sum_{i=1}^{n} \sum_{\alpha \in S_i, \beta \in P, (\alpha, \beta) \in \pi} w(\pi)(S_1 \wedge \cdots \wedge S_n \otimes P)^{\pi}_{\alpha \cup \beta}. \end{aligned} \tag{4.8}$$

We must now translate the pairing π into a spider mating. To accomplish this, we define a new pairing $\widetilde{\pi}$ which is the paring on the legs other than α and β induced by π, i.e.,

$\widetilde{\pi} = \pi - (\alpha, \beta)$. Working with the summand in (4.6):

$$\begin{aligned} &w(\pi)(S_1 \wedge \cdots \wedge S_i \wedge \cdots \wedge S_n \otimes P)^{\pi}_{\alpha \cup \beta} \\ &\quad = w(\pi)\omega(v_\alpha, v_\beta)(-1)^i (S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge S_n \otimes (S_i, \alpha)!(P, \beta))^{\widetilde{\pi}}. \end{aligned}$$

Note $w(\widetilde{\pi}) = w(\pi)\omega(v_\alpha, v_\beta)$ and so if we change the order of summation and substitute in our new summand into (4.8) we get

$$\cdots = \sum_{i=1}^{n} \sum_{\alpha \in S_i, \beta \in P} \sum_{\widetilde{\pi}} w(\widetilde{\pi})(-1)^i (S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge S_n \otimes (S_i, \alpha)!(P, \beta))^{\widetilde{\pi}},$$

as desired. □

4.4 An analysis of the $\mathfrak{sp}(2n)$-invariants in $\wedge \ell_n \otimes p_1 \ell_n$

Recall in Chapter 2, we defined a Lie symplectospider as an element of

$$\ell_n = \bigoplus_{m \geq 2} \left(\mathcal{LS}[m] \otimes V_n^{\otimes m} \right)_{\Sigma_m} = \bigoplus_{m \geq 2} \ell_n^m.$$

Using the fact $p_1 \ell_n$ is a ℓ_n-module, we form the chain groups $\wedge \ell_n \otimes p_1 \ell_n$ of the Chevalley-Eilenberg complex and then decompose the chain groups into a direct sum of terms $p\ell_{k,m}$ of wedges of $k-1$ Lie symplectospiders tensored with one singly-pointed Lie symplectospider which altogether have m legs in total (note we are suppressing the subscript of 1 on $p\ell_{k,m}$ in the interest of keeping the notation manageable). That is,

$$\begin{aligned} \wedge \ell_n \otimes p_1 \ell_n &= \bigoplus_{k,m} p\ell_{k,m} \\ &= \bigoplus_{\substack{k,\, m \\ m_1 + \cdots + m_k = m}} \ell_n^{m_1} \wedge \cdots \wedge \ell_n^{m_{k-1}} \otimes p_1 \ell_n^{m_k}. \end{aligned}$$

Note that the $\mathfrak{sp}(2n)$-action on $p_1 \ell_n$ and ℓ_n is via the mating of two-legged spiders with either a pointed or unpointed spider. We say more regarding the $\mathfrak{sp}(2n)$-invariants of $\wedge \ell_n \otimes p_1 \ell_n$: first, $(\wedge \ell_n \otimes p_1 \ell_n)^{\mathfrak{sp}(2n)}$ forms a subcomplex of $\wedge \ell_n \otimes p_1 \ell_n$. Indeed, we have

Lemma 4.4.1. *The boundary map of the Chevalley-Eilenberg complex is an $\mathfrak{sp}(2n)$-module morphism.*

Proof. Recall that the boundary map

$$\wedge^r \ell_n \otimes p_1 \ell_n \xrightarrow{\partial} \wedge^{r-1} \ell_n \otimes p_1 \ell_n$$

is given by

$$S_1 \wedge \cdots \wedge S_r \otimes P \mapsto \sum_{i=1}^{r} (-1)^i S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge S_r \otimes (S_i \cdot P) + \sum_{i<j} (-1)^{i+j} [S_i, S_j] \wedge S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge \widehat{S_j} \wedge \cdots \wedge S_r \otimes P.$$

Consider the element $S_1 \wedge \cdots \wedge S_r \otimes P$ of $\wedge^r \ell_n \otimes p_1 \ell_n$ and the element ξ of $\mathfrak{sp}(2n)$; we show $\partial(\xi \cdot (S_1 \wedge \cdots \wedge S_r \otimes P)) = \xi \cdot \partial(S_1 \wedge \cdots \wedge S_r \otimes P)$ for the linearity is clear by the definition of ∂. Note that we can expand the left hand side as

$$\begin{aligned}
&\partial(\xi \cdot (S_1 \wedge \cdots \wedge S_r \otimes P)) \\
&\quad = \partial \left(S_1 \wedge \cdots \wedge S_r \otimes (\xi \cdot P) + \sum_{i=1}^{r} S_1 \wedge \cdots \wedge (\xi \cdot S_i) \wedge \cdots \wedge S_r \otimes P \right) \\
&\quad = \partial \left(S_1 \wedge \cdots \wedge S_r \otimes (\xi \cdot P) \right) \\
&\qquad + \sum_{i=1}^{r} \partial \left(S_1 \wedge \cdots \wedge (\xi \cdot S_i) \wedge \cdots \wedge S_r \otimes P \right) \\
&\quad = \sum_{i=1}^{r} (-1)^i S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge S_r \otimes (S_i \cdot (\xi \cdot P)) \\
&\qquad + \sum_{i<j} (-1)^{i+j} [S_i, S_j] \wedge S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge \widehat{S_j} \wedge \cdots \wedge S_r \otimes (\xi \cdot P) \\
&\qquad + \sum_{\alpha=1}^{r} \left(\sum_{i=1}^{r} (-1)^i S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge (\xi \cdot S_\alpha) \wedge \cdots \wedge S_r \otimes (S_i \cdot P) \right) \\
&\qquad + \sum_{\alpha=1}^{r} \left(\sum_{i<j} (-1)^{i+j} [S_i, S_j] \wedge S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge (\xi \cdot S_\alpha) \wedge \cdots \wedge \widehat{S_j} \wedge \cdots \wedge S_r \otimes P \right)
\end{aligned}$$

and the right hand side (the details follow similarly) as

$$\begin{aligned}
&\xi \cdot \partial(S_1 \wedge \cdots \wedge S_r \otimes P) \\
&\quad = \sum_{i=1}^{r} (-1)^i S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge S_r \otimes (\xi \cdot (S_i \cdot P)) \\
&\qquad + \sum_{i<j} (-1)^{i+j} [S_i, S_j] \wedge S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge \widehat{S_j} \wedge \cdots \wedge S_r \otimes (\xi \cdot P) \\
&\qquad + \sum_{i=1}^{r} \left(\sum_{\alpha=1}^{r-1} (-1)^i S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge (\xi \cdot S_\alpha) \wedge \cdots \wedge S_r \otimes (S_i \cdot P) \right) \\
&\qquad + \sum_{i<j} \left(\sum_{\alpha=1}^{r-2} (-1)^{i+j} [S_i, S_j] \wedge S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge (\xi \cdot S_\alpha) \wedge \cdots \wedge \widehat{S_j} \wedge \cdots \wedge S_r \otimes P \right).
\end{aligned}$$

Subtracting like quantities, we reduce the lemma to verifying

$$
\begin{aligned}
&\sum_{i=1}^{r}(-1)^i S_1 \wedge \cdots \wedge \widehat{S_i} \wedge \cdots \wedge S_r \otimes (S_i \cdot (\xi \cdot P)) \\
&\quad + \sum_{\alpha=1}^{r}\left(\sum_{i=1}^{r}(-1)^i S_1 \wedge \cdots \wedge \widehat{S}_i \wedge \cdots \wedge (\xi \cdot S_\alpha) \wedge \cdots \wedge S_r \otimes (S_i \cdot P)\right) \\
&\quad + \sum_{\alpha=1}^{r}\left(\sum_{i<j}(-1)^{i+j}[S_i, S_j] \wedge S_1 \wedge \cdots \wedge \widehat{S}_i \wedge \cdots \wedge (\xi \cdot S_\alpha) \wedge \cdots \wedge \widehat{S_j} \wedge \cdots \wedge S_r \otimes P\right) \\
&= \sum_{i=1}^{r}(-1)^i S_1 \wedge \cdots \wedge \widehat{S}_i \wedge \cdots \wedge S_r \otimes (\xi \cdot (S_i \cdot P)) \\
&\quad + \sum_{i=1}^{r}\left(\sum_{\alpha=1}^{r-1}(-1)^i S_1 \wedge \cdots \wedge \widehat{S}_i \wedge \cdots \wedge (\xi \cdot S_\alpha) \wedge \cdots \wedge S_r \otimes (S_i \cdot P)\right) \\
&\quad + \sum_{i<j}\left(\sum_{\alpha=1}^{r-2}(-1)^{i+j}[S_i, S_j] \wedge S_1 \wedge \cdots \wedge \widehat{S}_i \wedge \cdots \wedge (\xi \cdot S_\alpha) \wedge \cdots \wedge \widehat{S_j} \wedge \cdots \wedge S_r \otimes P\right).
\end{aligned}
$$

The latter equality is seen to hold by an appeal to the algebraic structure of our objects. Since $p_1\ell_\infty$ is a Lie module over ℓ_∞, it must be the case that $[S_1, S_2] \cdot P = S_1 \cdot (S_2 \cdot P) - S_2 \cdot (S_1 \cdot P)$. If we decompose our sums in a "diagonal" (the pulled term), "lower diagonal" (precede the pulled term), and "upper diagonal" (follow the pulled term) manner in the topmost part of the equality, these terms coincide with the unexpanded (in reference to the module action) terms of the bottom portion of the equality. We also comment that this lemma follows directly from the fact that the boundary operator commutes with the adjoint action.

□

Since the boundary operator is an $\mathfrak{sp}(2n)$-module morphism, we may conclude that $(\wedge\ell_n \otimes p_1\ell_n)^{\mathfrak{sp}(2n)}$ is a subcomplex of $\wedge\ell_n \otimes p_1\ell_n$. Given the decomposition of $\wedge\ell_n \otimes p_1\ell_n$ into the subspaces of k spiders with a total of m legs $\bigoplus_{k,m} p\ell_{k,m}$, we form the spaces

$$
\begin{aligned}
Z_{k,m} &= \ker \partial_n \cap p\ell_{k,m} \\
B_{k,m} &= \text{im } \partial_n \cap p\ell_{k,m} \\
H_{k,m}(\ell_n; p_1\ell_n) &= Z_{k,m}/B_{k,m}
\end{aligned}
$$

so that the Lie homology $H_k(\ell_n; p_1\ell_n)$ can be expressed as $\bigoplus_m H_{k,m}(\ell_n; p_1\ell_n)$.

The map ψ_n of Lemma 4.3.4 provides us with a way to produce a pointed Lie graph from a collection of spiders. The reverse map is given as φ_n in [7]; we extend the definition here to the pointed case:

Definition 4.4.2. Let P be a basepointed Lie graph with underlying oriented graph X. For each edge of P, break the edge in half so as to get a disjoint union of spiders. Now, to turn this into an element of $\wedge\ell_n \otimes p_1\ell_n$ we wedge the spiders in the order given by the orientation (recall an orientation gives a vertex ordering and a direction associated to each edge) and assign two elements of $\mathcal{B}_n$ and a sign to each edge as follows: Let e be an edge

of P which consists of the two half-edges e_0 and e_1 (in order from the orientation of the half-edges). Place $v \in \{p_i, q_i\}$ on the half-edge e_0 and $v' \in \{p_i, q_i\} - \{v\}$ on the half-edge e_1. If the half-edge e_0 is labeled with p_i (hence e_1 carries q_i), then the edge has a positive sign associated to it. If the initial half-edge e_0 is labeled with q_i, then we associate a minus sign to the edge. An assignment of an element of $\mathcal{B}_n$ to each half-edge and along with a sign for each edge following the previous description is called a *state* of P. We define the *sign* $\sigma(s)$ of the state s to be the product of the signs from the edges given by the state s. As defined, a state s gives rise to an element P_s of $\wedge\ell_n \otimes p_1\ell_n$. Now, we define the map $p_1\mathcal{LG} \xrightarrow{\varphi_n} \wedge\ell_n \otimes p_1\ell_n$ by

$$P \mapsto \sum_s \sigma(s) P_s.$$

Now, by the work of the previous lemma and the second part of Proposition 8 of [7], we have that the inclusion of the $\mathfrak{sp}(2n)$-invariants of $\wedge\ell_n \otimes p_1\ell_n$ into $\wedge\ell_n \otimes p_1\ell_n$ induces an isomorphism on homology.

Since the tree decorating the interior of the spider is invariant under this action, the $\mathfrak{sp}(2n)$ action only affects the tensor power of $V^{\otimes m}$ in ℓ_n^m and $p_1\ell_n^m$. It thus suffices to identify the $\mathfrak{sp}(2n)$-invariants of $V^{\otimes m}$. In [10], it is shown that the invariants can be given by chord diagrams. The details of this identification are given by

Proposition 4.4.3. *The image of the map φ_n coincides with the invariants in $\wedge\ell_n \otimes p_1\ell_n$, that is, $\operatorname{im}\varphi_n = (\wedge\ell_n \otimes p_1\ell_n)^{\mathfrak{sp}(2n)}$.*

Proof. The idea behind this proof will be to supplement the determination of invariants of $\wedge\ell_n$ from [7] for our purposes. We recreate much of the proof here for the benefit of the reader.

Let $T\ell_n \xrightarrow{\tau} \wedge\ell_n$ be the standard quotient map from the tensor algebra to the alternating algebra and define a map ι by $S_1 \wedge \cdots \wedge S_k \mapsto \frac{1}{k!}\sum_{\sigma\in\Sigma_k}(-1)^{|\sigma|}S_{\sigma(1)} \otimes \cdots \otimes S_{\sigma(k)}$. These maps fit into a sequence

$$\wedge\ell_n \xrightarrow{\iota} T\ell_n \xrightarrow{\tau} \wedge\ell_n$$

of $\mathfrak{sp}(2n)$-module morphisms. Furthermore, we have $\tau\iota = id$ and so it follows τ is a epimorphism of $\mathfrak{sp}(2n)$-modules.

Recall that ℓ_n is defined as the sum of the Σ_m-coinvariants in $\mathcal{LS}[m] \otimes V_n^{\otimes m}$, that is

$$\ell_n = \bigoplus_{m\geq 2} (\mathcal{LS}[m] \otimes V_n^{\otimes m})_{\Sigma_m}.$$

Let $\widehat{\ell}_n = \bigoplus_{m\geq 2}(\mathcal{LS}[m] \otimes V_n^{\otimes m})$; we again pass to the quotient $T\widehat{\ell}_n \xrightarrow{\gamma} T\ell_n$ and note that if we let $\widehat{\iota}$ be induced by the map

$$\begin{aligned}(\mathcal{LS}[m] \otimes V_n^{\otimes m})_{\Sigma_m} &\longrightarrow \mathcal{LS}[m] \otimes V_n^{\otimes m} \\ S &\longmapsto \frac{1}{m!}\sum_{\sigma\in\Sigma_m} \sigma \cdot S\end{aligned}$$

we get a similar sequence as before

$$T\ell_n \xrightarrow{\widehat{\iota}} T\widehat{\ell}_n \xrightarrow{\gamma} T\ell_n$$

The maps compose to the identity and are $\mathfrak{sp}(2n)$-module morphisms. Hence γ is an epimorphism of $\mathfrak{sp}(2n)$-modules.

Putting this together, we have a new sequence of $\mathfrak{sp}(2n)$-module epimorphisms

$$T\widehat{\ell}_n \xrightarrow{\gamma} T\ell_n \xrightarrow{\tau} \wedge\ell_n.$$

Note that $p_1\ell_n \xrightarrow{id} p_1\ell_n$ is an $\mathfrak{sp}(2n)$-module epimorphism and since the tensor of epimorphisms is an epimorphism, we have that $\tau \otimes id$ is an epimorphism. Similar to the $\widehat{\ell}_n$ construction above, we form $p_1\widehat{\ell}_n = \bigoplus_{m\geq 2}(p_1\mathcal{LS}[m] \otimes V_n^{\otimes m})$ and observe we again get an $\mathfrak{sp}(2n)$-module epimorphism $p_1\widehat{\ell}_n \xrightarrow{\eta} p_1\ell_n$. This fits nicely together in

$$T\widehat{\ell}_n \otimes p_1\widehat{\ell}_n \xrightarrow{\gamma\otimes\eta} T\ell_n \otimes p_1\ell_n \xrightarrow{\tau\otimes id} \wedge\ell_n \otimes p_1\ell_n,$$

which we note $(\tau \otimes id) \circ (\gamma \otimes \eta) = (\tau \circ \gamma) \otimes (id \circ \eta)$ implies the composite is an epimorphism since each term is an epimorphism. It now suffices to identify the invariants in $T\widehat{\ell}_n \otimes p_1\widehat{\ell}_n$.

To compute the invariants in $T\widehat{\ell}_n \otimes p_1\widehat{\ell}_n$ we decompose $T\widehat{\ell}_n \otimes p_1\widehat{\ell}_n$ as

$$T\widehat{\ell}_n \otimes p_1\widehat{\ell}_n = \bigoplus_{k\geq 2, m\geq 1} p\widehat{\ell}_{k,m}$$

with

$$p\widehat{\ell}_{k,m} = \bigoplus_{m_1+\cdots+m_k=m} (\mathcal{LS}[m_1] \otimes V_n^{\otimes m_1}) \otimes \cdots \otimes (\mathcal{LS}[m_{k-1}] \otimes V_n^{\otimes m_{k-1}}) \otimes (p\mathcal{LS}[m_k] \otimes V_n^{\otimes m_k}).$$

Recall that the $\mathfrak{sp}(2n)$-action is via a two-legged symplectospider mating and that this mating does not affect the underlying spider structure but does change the symplectic labeling of the spider. To be precise, the $\mathfrak{sp}(2n)$-action on $\mathcal{LS}[m_i] \otimes V^{\otimes m_i}$ is only effective on the $V^{\otimes m_i}$ term (the same is true for the pointed case). With this, we then are led to

$$(p\widehat{\ell}_{k,m})^{\mathfrak{sp}(2n)} = \bigoplus_{m_1+\cdots+m_k=m} (V_n^{\otimes m_1}\otimes\cdots\otimes V_n^{\otimes m_k})^{\mathfrak{sp}(2n)} \otimes (\mathcal{LS}[m_1]\otimes\cdots\otimes\mathcal{LS}[m_{k-1}]\otimes p\mathcal{LS}[m_k]).$$

The determination of the $\mathfrak{sp}(2n)$-invariants in $V_n^{\otimes m_1} \otimes \cdots \otimes V_n^{\otimes m_k}$ is a result that goes back Weyl. We will describe the process as given in [10]. First, we write $V_n^{\otimes m_1}\otimes\cdots\otimes V_n^{\otimes m_k} = V^{\otimes m}$. It turns out that a base for the invariants in $V^{\otimes m}$ is given by oriented chord diagrams on m vertices. An oriented chord diagram is a pairing of the m vertices into ordered pairs $v_i \to v_j$ (that is, directed edges) where we consider v_i to be the "tail" and v_j to be the "head" of the directed edge.

Now, to every pairing $v_i \to v_j$ we define a bijection $\{v_i, v_j\} \leftrightarrow \{p_k, q_k\}$ where the elements $\{p_1, \cdots, p_n, q_1, \cdots, q_n\}$ form a basis for the symplectic vector space V_n. We impose the condition that if the element q_k is identified with v_i while p_k is identified with v_j, then this directed edge carries a minus sign. That is, if a directed edge is given by $(head) \to (tail)$, then $(head) \leftrightarrow p_k$ and $(tail) \leftrightarrow q_k$ causes the edge to carry a "+" sign whereas $(head) \leftrightarrow q_k$ and $(tail) \leftrightarrow p_k$ causes the edge to carry a "−" sign.

Define a state to be an established bijection between every directed edge and $\{p_k, q_k\}$ in our chord diagram together with an associated sign given by the product of the signs attached to the directed edges. Then the invariant is the sum over all possible states of the

symplectic basis elements representing the vertices for $V^{\otimes m}$.

We have established the invariants in $T\widehat{\ell}_n \otimes p_1\widehat{\ell}_n$, but we need to identify the invariants in $\wedge\ell_n \otimes p_1\ell_n$. Earlier in the proof we concluded that

$$T\widehat{\ell}_n \otimes p_1\widehat{\ell}_n \xrightarrow{\gamma\otimes\eta} T\ell_n \otimes p_1\ell_n \xrightarrow{\tau\otimes id} \wedge\ell_n \otimes p_1\ell_n,$$

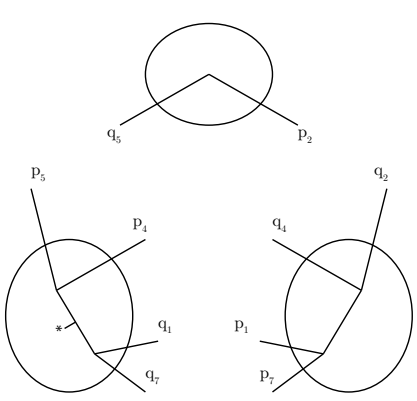
is a sequence of $\mathfrak{sp}(2n)$-module epimorphisms. Thus the maps restrict to the sequence of epimorphisms

$$(T\widehat{\ell}_n \otimes p_1\widehat{\ell}_n)^{\mathfrak{sp}(2n)} \xrightarrow{\gamma\otimes\eta} (T\ell_n \otimes p_1\ell_n)^{\mathfrak{sp}(2n)} \xrightarrow{\tau\otimes id} (\wedge\ell_n \otimes p_1\ell_n)^{\mathfrak{sp}(2n)}.$$

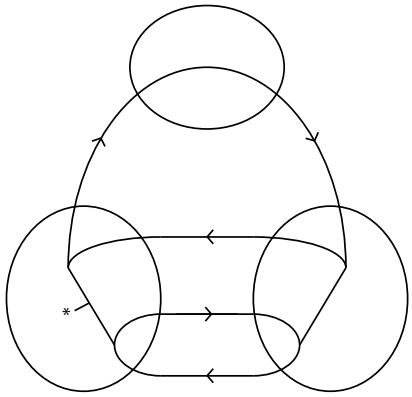
Let $\xi \otimes S_1 \otimes \cdots \otimes S_{k-1} \otimes P$ be an element in $(V_n^{\otimes m_1} \otimes \cdots \otimes V_n^{\otimes m_k})^{\mathfrak{sp}(2n)} \otimes (\mathcal{LS}[m_1] \otimes \cdots \otimes \mathcal{LS}[m_{k-1}] \otimes p\mathcal{LS}[m_k])$ where ξ is an $\mathfrak{sp}(2n)$-invariant in $V_n^{\otimes m_1} \otimes \cdots \otimes V_n^{\otimes m_k}$. The image of this element under the map $(\tau\otimes id)\circ(\gamma\otimes\eta)$ can be identified with the image of a pointed Lie graph G as follows. The $\mathfrak{sp}(2n)$-invariant ξ is identified with its chord diagram counterpart and when we map $\xi\otimes S_1\otimes\cdots\otimes S_{k-1}\otimes P$ to $\wedge\ell_n\otimes p_1\ell_n$, the result is attaching the symplectic elements dictated by the pairing of ξ to the legs of the spiders $S_1, \cdots, S_{k-1}, P$. Further, the diagram invariant ξ gives the exact way to attach the spider legs to give rise to a pointed Lie graph. Note that each possible state of the oriented chord diagram ξ gives rise to a state of the resulting graph. If we sum over all possible states ξ, this corresponds to a sum over all states of a graph, i.e., it is precisely the image of φ_n.

□

We illustrate the process of creating a pointed Lie graph from an $\mathfrak{sp}(2n)$-invariant for $V^{\otimes 10} = (V^{\otimes 2})^{\otimes 1} \otimes (V^{\otimes 4})^{\otimes 2}$. The way to interpret the individual tensor powers is $(V^{\otimes i})^{\otimes j}$ means there are j vertices in a particular Lie graph which contain a planar binary tree with i leaves. A possible pairing could be given by



(a) Three symplectospiders



(b) Gluing spider legs together based on chord diagram invariant gives a hairy Lie graph

Figure 4.2: How an $\mathfrak{sp}(2n)$-invariant gives rise to a hairy Lie graph

$$\begin{array}{ll} 1 \to 10 & 8 \to 2 \\ 3 \to 7 & 6 \to 4 \ . \\ 9 \to 5 & \end{array}$$

Next, we establish a bijection between the vertex pairs and the basis elements for V_n

$$\begin{array}{lll} 1 \to 10 & 8 \to 2 & \\ 3 \to 7 & 6 \to 4 & \longleftrightarrow \\ 9 \to 5 & & \end{array} \begin{array}{ll} p_5 \to q_5 & q_4 \to p_4 \\ q_1 \to p_1 & p_7 \to q_7 \ . \\ p_2 \to q_2 & \end{array}$$

The directed edges $q_1 \to p_1$ and $q_4 \to p_4$ each carry a "$-$" sign and so one state for $V^{\otimes 10}$ would be $(-1)^2\, p_5 \otimes p_4 \otimes q_1 \otimes q_7 \otimes q_2 \otimes p_7 \otimes p_1 \otimes q_4 \otimes p_2 \otimes q_5$. A pictorial representation is given in Figure 4.2.

Proposition 4.4.4. *The spaces $H_\bullet(\ell_\infty; p_1\ell_\infty)$ and $H_\bullet(p_1\mathcal{LG})$ are isomorphic as Hopf modules.*

Proof. To see that ψ_∞ (of Lemma 4.3.5) is a morphism of Hopf modules, consider the diagram

$$\begin{array}{ccccc} (\wedge\ell_\infty) \otimes (\wedge\ell_\infty \otimes p_1\ell_\infty) & \xrightarrow{\psi\otimes\psi_\infty} & \mathcal{LG} \otimes p_1\mathcal{LG} & \xrightarrow{\psi^{-1}\otimes id} & \wedge\ell_\infty \otimes p_1\mathcal{LG} \\ \downarrow & & \downarrow & \nearrow & \\ \wedge\ell_\infty \otimes p_1\ell_\infty & \xrightarrow{\psi_\infty} & p_1\mathcal{LG} & & \end{array}$$

where the vertical maps are given by the corresponding module actions and the map ψ is defined in [7] and is shown to be an isomorphism. We note that the composition of the two top morphisms gives $id \otimes \psi_\infty$ (as is necessary in the definition of a Hopf module morphism) and the right diagonal map translates the $\mathcal{LG}$ action into an action of $\wedge\ell_\infty$ on $p_1\mathcal{LG}$.

Let $S_I \otimes (S_J \otimes P)$ be an element of $(\wedge\ell_\infty) \otimes (\wedge\ell_\infty \otimes p_1\ell_\infty)$. Then tracing the diagram down the first vertical map gives $E(S_I) \wedge O(S_J) \otimes O(P)$. Recall that the action of ψ_∞ on a wedge of spiders is to pair the spider legs according to their symplectic labels. Since the parity of the terms $E(S_I)$ and $O(S_J) \otimes O(P)$ differ, the result is that ψ_∞ gives the disjoint union of the latter two. This is the precise description of the second vertical map in the diagram.

Now consider the diagram

$$\begin{array}{ccccc} \wedge\ell_\infty \otimes p_1\ell_\infty & \xrightarrow{\psi_\infty} & p_1\mathcal{LG} & & \\ \downarrow & & \downarrow & \searrow & \\ (\wedge\ell_\infty) \otimes (\wedge\ell_\infty \otimes p_1\ell_\infty) & \xrightarrow{\psi\otimes\psi_\infty} & \mathcal{LG} \otimes p_1\mathcal{LG} & \xrightarrow{\psi^{-1}\otimes id} & \wedge\ell_\infty \otimes p_1\mathcal{LG} \end{array}$$

where the vertical maps are given by the corresponding comodule actions, the bottom two morphisms compose to give the morphism $id \otimes \psi_\infty$, and the right diagonal map extends the comodule action. Analyzing the definitions of the maps, we note that the commutativity of this diagram results by a correspondence between the pairing via ψ_∞ and the partitioning in the comodule action of $\wedge\ell_\infty$ on $\wedge\ell_\infty \otimes p_1\ell_\infty$. This follows as in the nonpointed case because a pairing π induced by ψ_∞ corresponds to a partition $I \sqcup J$ which yields the splitting of π into π_I and π_J as in Lemma 4.3.4. $\square$

4.5 An analysis of the pointed Lie graph complex

In order to draw the conclusions we desire, we must get to the connected graphs of $H_\bullet(p_1\mathcal{LG})$. Recall that the coaction is defined so that the connected graphs are those such that $P \mapsto 1 \otimes P$. In the theory of A-comodules, this subspace has a special name, the coinvariants, and it is indicated by a superscript co A. If we denote the subspace generated by connected graphs by $H_\bullet(P(p_1\mathcal{LG}))$, then, noting ψ_∞ is an isomorphism, we let $H_\bullet(\ell_\infty; p_1\ell_\infty)^{\mathrm{co}\,\ell_\infty}$ be the corresponding subspace of $H_\bullet(\ell_\infty; p_1\ell_\infty)$.

The subspace of connected pointed Lie graphs splits as $\mathcal{P} \oplus P(p_1\mathcal{LG})/\mathcal{P}$ where

- $\mathcal{P}$ is spanned by graphs with only bivalent vertices (polygons)
- $P(p_1\mathcal{LG})/\mathcal{P}$ is spanned by graphs with at least one (≥ 3)-valent vertex

Recall that the space of two-legged symplectospiders is isomorphic to $\mathfrak{sp}(2n)$. If we decorate the vertices of a graph with these two-legged spiders, we note the correspondence in the graph setting:

Lemma 4.5.1. *$H_\bullet(\mathcal{P}) \cong H_\bullet(\mathfrak{sp}(2\infty))$*

Proof. We will consider the spaces $\mathcal{P}_j$ of $\mathcal{P} = \bigoplus \mathcal{P}_j$ based upon the congruence of j modulo 4 as particular symmetries will be able to be exploited. Let us set up some notation: the space $\mathcal{P}_j$ is spanned by at most the j oriented polygons $p_j^1, \cdots, p_j^j$ where p_j^i represents a polygon with j sides and a distinguished basepoint on the ith vertex.

Suppose $j \equiv 0 \bmod 4$. If we rotate p_j^i by $2\pi/j$ and suitably relabel so that the orientation of $exp(2\pi i/j)p_j^i$ is coherent with that of p_j^{i+1}, we note that $p_j^i = -p_j^{i+1}$. Hence $\mathcal{P}_j$ is generated by p_j^1. Now, rotate p_j^1 about the axis through the basepoint that bisects the polygon. This reverses the orientation and so $\mathcal{P}_j = 0$ for $j \equiv 0 \bmod 4$.

If $j \equiv 1 \bmod 4$, then $p_j^i = 0$ for all i. To see this, rotate p_j^i about the axis passing through the basepoint on vertex i and the edge "opposite" it on the polygon. This action reverses the orientation since there is an odd number of sides to reverse and an even number of vertex pairs to interchange. Therefore $p_j^i = 0$.

For $j \equiv 2 \bmod 4$, we can use the argument presented for the previous scenario except now, there is an even number of edge orientation swaps and odd number of vertex pair swaps. Finally, when $j \equiv 3 \bmod 4$, we pick up the generator p_j^1. Rotation by $2\pi/j$ preserves the orientation, and so the polygons $p_j^1, \cdots, p_j^j$ are all congruent: $p_j^1 = p_j^2 = \cdots = p_j^j$.

We then have the chain complex

$$\cdots \to 0 \to \mathbb{Q}\left\{p_{4j-1}^1\right\} \to 0 \to \cdots$$

It follows that $\mathcal{P}$ only has nontrivial homology only if $j \equiv 3 \bmod 4$ and in this case it is seen to be cyclic and coincides with the homology of $\mathfrak{sp}(2\infty)$, as was to be shown. □

The previous lemma details graphs in which all vertices are bivalent. We now analyze the effect of $\mathfrak{sp}(2n)$-colored vertices in a general pointed Lie graph. To begin, we consider a special type of graph and then create a chain complex with boundary operator given by the standard graph complex boundary $\partial = \partial_E$. For $k \geq 0$, let V_k^0 denote the (linear) graph with k internal vertices and hence $k-1$ internal edges. Now, take V_k^0 and add a distinguished basepoint to the ith vertex; denote this new graph by V_k^i. The space $\mathcal{V}_k$ is then generated by $V_k^1, \cdots V_k^k$. We claim

Lemma 4.5.2. *The graph homology of $\mathcal{V}_k$ vanishes except at $k = 0$ when it is $\mathbb{R}$.*

Proof. It is clear by the definitions of $\mathcal{V}_\bullet$ and ∂ that $H_0(\mathcal{V}_\bullet) = \mathbb{R}$. Note that for k odd, we have $\partial(V_k^i)$ vanishes for i odd and $\partial(V_k^i) = V_{k-1}^i - V_{k-1}^{i+1}$ for i even. For k even, $\partial(V_k^i)$ and $\partial(V_k^{i+1})$ have image V_{k-1}^i where i is odd. Direct computation now yields the result. □

Hence, aside from dimension 0, the addition of a bivalent vertex to a pointed Lie graph contributes nothing to homology. Recall that the space $P(p_1\overline{LG})$ is spanned by connected graphs with no bivalent vertices and $P(p_1LG)/\mathcal{P}$ is spanned by connected graphs with at least one (≥ 3)-valent vertex. In the interest of relating $P(p_1\overline{LG})$ and $P(p_1LG)/\mathcal{P}$, let us filter $P(p_1LG)/\mathcal{P}$ by the number of non-bivalent vertices (call this filter F_g). Define $E^0_{g,b}$ to be the span of graphs with g (≥ 3)-valent vertices and b bivalent (i.e., identity-colored) vertices. We take the vertical boundary maps $E^0_{g,b} \xrightarrow{\partial_y} E^0_{g,b-1}$ as the standard graph complex boundary maps except the edge contractions only occur at edges incident to a bivalent vertex. Defined as such, we claim that

Lemma 4.5.3. *The spaces $H_\bullet(P(p_1\overline{LG}))$ and $H_\bullet(P(p_1LG)/\mathcal{P})$ are isomorphic.*

Proof. Note that the definition of $E^0_{g,b}$ given corresponds to filtering the space $P(p_1LG)/\mathcal{P}$ by the number of non-bivalent vertices and so the spectral sequence attached to $E^0_{g,b}$ is the quotient complex $F_g(P(p_1LG)/\mathcal{P})_{g+b}/F_{g-1}(P(p_1LG)/\mathcal{P})_{g+b}$. Let us analyze the vertical complexes $E^0_{g,\bullet}$.

Define an equivalence relation on connected graphs with at least one (≥ 3)-valent vertex by $G \sim G'$ if G and G' are isomorphic after the removal of all bivalent identity-colored vertices. Observe that $E^0_{g,0}$ consists of graphs with no bivalent vertices and can therefore be identified with one of our complexes of interest, $P(p_1\overline{LG})$.

We decompose the vertical complex as $E^0_{g,\bullet} = \bigoplus_G E_g(G)$ where the index G runs over isomorphism classes of graphs G from $E^0_{g,\bullet}$, that is a graph G of $E^0_{g,\bullet}$ serves as a representative for its equivalence class in $E^0_{g,\bullet} = P(p_1\overline{LG})$. Given a representative G, let $\widehat{G}$ be the graph G except it possesses only trivial automorphisms. Since $E_g(G)$ is free, we have $E_g(G)/Aut(G) = E_g(\widehat{G})$ by (I.4.2) of [5]. Moreover, since we are working with real coefficients, we have $H_b(E_g(G)) \cong H_b(E_g(\widehat{G}))$, provided one of the spaces is trivial.

We decompose further: $E_g(\widehat{G}) = \bigoplus E_g(\widehat{G}; S_1, \cdots, S_g)$ where the indexing is over a basis for $p_1\mathcal{LS}/\mathbb{R}\{1_{\mathcal{LS}}\}$. The latter decomposition follows from the fact ∂_y is restricted to the bivalent vertices and so essentially, only the non-$1_{\mathcal{LS}}$ colored vertices can contribute to homology. Hence, when $b > 0$ (that is, there exist bivalent vertices), we have $E_g(\widehat{G}; S_1, \cdots, S_g) \cong \bigotimes_{e \in E(\widehat{G})} \mathcal{E}_\bullet$ where the tensor argument $\mathcal{E}_\bullet$ is acyclic except in dimension 0 by Lemma 4.5.2. If we appeal to the Kunneth formula, we note $\bigotimes_{e \in E(\widehat{G})} \mathcal{E}_\bullet$ is acyclic except in dimension 0, as well. Hence, $H_b(E_\bullet(\widehat{G})) = 0$ when $b > 0$ and the only contribution to homology occurs on the g-axis (when $b = 0$).

Now that we have determined that the homology is concentrated along the g-axis, note that our sequence collapses at the E^1 page. We further note that the boundary operator for the E^1 page coincides with the boundary operator for the point Lie graph complex, ∂_E and the result follows. □

So far in this section we have established

(Lemma 4.5.2) $H_\bullet(P(p_1\overline{LG})) \cong H_\bullet(P(p_1LG)/\mathcal{P})$ and

(Lemma 4.5.3) $H_\bullet(\mathcal{P}) \cong H_\bullet(\mathfrak{sp}(2\infty))$.

Recalling that our space of interest $P(p_1\mathcal{LG})$ decomposes into $\mathcal{P}$ and $P(p_1\mathcal{LG})/\mathcal{P}$, we conclude

Proposition 4.5.4. $H_\bullet(P(p_1\mathcal{LG})) \cong H_\bullet(\mathfrak{sp}(2\infty) \oplus H_\bullet(P(p_1\overline{LG}))$.

4.6 The pointed forested graph complex

In §2.7.1 we defined the forested graph complex $f\mathcal{G}_k$ and set the foundation for establishing a natural isomorphism between it and the connected Lie graphs without bivalent vertices, $P\overline{\mathcal{LG}}_k$.

Definition 4.6.1. Let G be a finite connected trivalent graph together with a basepoint $*$ and an oriented subforest Φ which contains all of the vertices of G. We will call this object $(G, \Phi, *)$ a *pointed forested graph*. Denote the space spanned by pointed forested graphs whose subforest contains r edges (or *trees*) quotiented by the relation $(G, -\Phi, *) = -(G, \Phi, *)$ by $\widetilde{pf\mathcal{G}}_r$.

We impose another relation on pointed forested graphs which will correspond to the IHX/Aut-IHX relations from the pointed Lie operad. To start, we define the subspace of basic IHX relators on $\widetilde{pf\mathcal{G}}_r$.

Consider the graph in Figure 4.3(a). There are three ways to grow the four-valent vertex v into a new edge of the forest and create an element of $\widetilde{pf\mathcal{G}}_4$. These three ways are: grow it "vertically," grow it "horizontally," and grow it "horizontally" with a twist, see Figures 4.3(b), 4.3(d), and 4.3(f), respectively. The sum of these three terms will be known as a basic IHX relator and the space spanned by these relators will be denoted IHX_r.

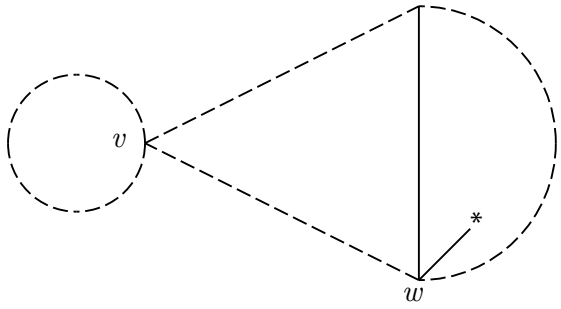
We now define the subspace of pointed IHX relators, $p\mathrm{IHX}_r$. In Figure 4.3(a), consider the vertex w where we have continued our convention of growing the basepoint out from the graph on a "hair." As such, the three blow-ups of this "four"-valent vertex follow in suit with the previous paragraph as shown in Figures 4.3(c), 4.3(e), and 4.3(g).

Define $pf\mathcal{G}_r$ to be $\widetilde{pf\mathcal{G}}_r$ modulo the subspaces IHX_r and $p\mathrm{IHX}_r$. The boundary map on the *unpointed* forested graph complex $f\mathcal{G}_r$ was given in 2.7.1 as the sum over all ways of adding an edge to the subforest so that the addition of an edge e to Φ remains acyclic. With this identification, $f\mathcal{G}_r$ becomes a chain complex which is naturally isomorphic (as a vector space) to $P\overline{\mathcal{LG}}_r$, see [7]. Along these lines, we claim:

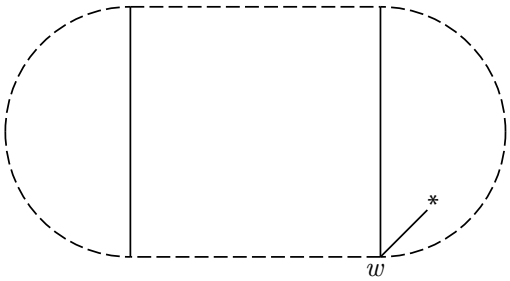
Proposition 4.6.2. *There is a natural vector space isomorphism between the pointed forested graph complex $pf\mathcal{G}_r$ and the space $Pp_1\overline{\mathcal{LG}}_r$ of connected pointed Lie graphs without bivalent vertices.*

Proof. We define a morphism $\widetilde{pf\mathcal{G}}_r \xrightarrow{\varphi} Pp_1\overline{\mathcal{LG}}_r$ as follows:

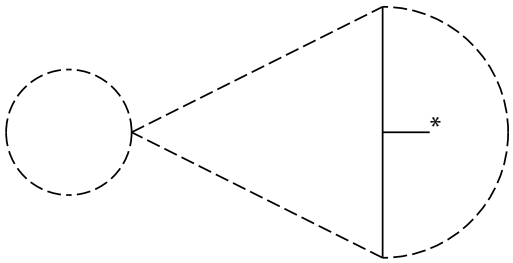
1. Take a pointed forested graph (G, Φ) in $\widetilde{pf\mathcal{G}}_r$.
2. For every connected component of Φ, take an ϵ neighborhood of the component that does not intersect any other neighborhood of a component.
3. Collapse each of these neighborhoods to create a vertex in the quotient graph $\widehat{G}$. For each vertex v of $\widehat{G}$ we associate to it the component T_v which existed before the collapse. Note that we "lose" the basepoint in the quotient graph, but it is retained in the tree associated to a collapsed vertex in $\widehat{G}$.



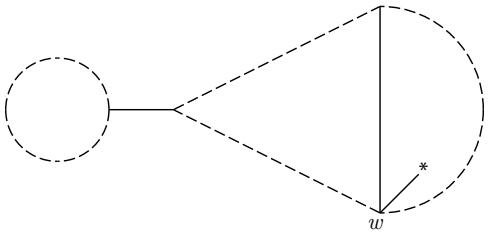
(a) A basepointed graph. The solid edges indicate components of the subforest



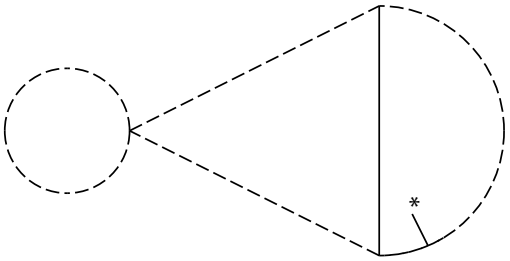
(b) Vertical growth "I" of the vertex v



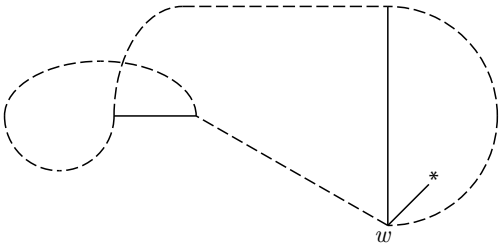
(c) Component one of pIHX for the vertex w



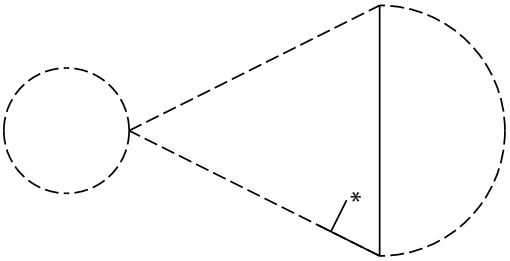
(d) Horizontal growth "H" of the vertex v



(e) Component two of pIHX for the vertex w



(f) Horizontal growth with a twist "X" of the vertex v



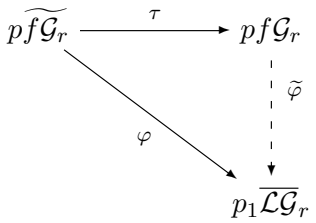
(g) Component three of pIHX for the vertex w

Figure 4.3: IHX and pIHX relators

We comment that the coherency of orientation between these objects follows as in Proposition 21 of [7] (or see Lemma 2.7.4) once we grow out our basepoint to the tip of a hair. The orientation of a pointed forested graph is then given by a number of the edges of Φ. Note that the choice of orientation is canonical.

Each T_v is a binary tree and we can identify the leaves of T_v with the half edges incident to v. We also identify the interior edges of each T_v with their forest preimage, Φ_v, so that Φ can be expressed as the union over the vertices $v \in V(\widehat{G})$ of Φ_v. It follows that the image of a pointed forested graph under φ can be expressed as $(\widehat{G}, \{T_v\})$.

Let us verify that the IHX and pIHX relators factor through the quotient. If we let $\widetilde{pf\mathcal{G}_r} \xrightarrow{\tau} f\mathcal{G}_r$ be the quotient map, then we wish verify there exists a map $\widetilde{\varphi}$ such that the diagram below commutes.



By the universal mapping property of the quotient, it then suffices to show that IHX_r and $p\mathrm{IHX}_r$ are contained in the kernel of φ. That IHX_r is contained in $\ker\varphi$ was shown in [7]; we consider the pointed IHX subspace. Suppose $(G_I, \Phi_I) + (G_H, \Phi_H) + (G_X, \Phi_X)$ is a pointed IHX relator. The image of this relator under φ is the linear combination $\pm(\widehat{G}_I, \{T_v\}_I) \pm (\widehat{G}_H, \{T_v\}_H) \pm (\widehat{G}_X, \{T_v\}_X)$. In order for this sum to vanish, it must be the case that the signs are $(+,-,+)$ so that the pIHX relator maps to the Aut-IHX relation of $p_1\overline{\mathcal{LG}}_r$. Using our convenient trick of turning the basepoint resulting from our distinguished basepoint into a (original valence + 1) vertex with the basepoint on the tip of a hair, we note that indeed this pIHX relator can be viewed as an IHX relator and hence corresponds to the IHX relation in the graph complex. Reidentifying our hair as a hairy IHX relation as in Figure 4.1(b), we conclude that the pIHX relator maps into the Aut-IHX subspace and so $(G_I, \Phi_I) + (G_H, \Phi_H) + (G_X, \Phi_X) \longmapsto (\widehat{G}_I, \{T_v\}_I) - (\widehat{G}_H, \{T_v\}_H) + (\widehat{G}_X, \{T_v\}_X) = 0$. Thus the subspaces IHX_r and $p\mathrm{IHX}_r$ are in the kernel of φ and there exists a map $pf\mathcal{G}_r \xrightarrow{\widetilde{\varphi}} p_1\overline{\mathcal{LG}}_r$ such that the diagram above commutes.

It now remains to show $\widetilde{\varphi}$ is a bijection. We do so by constructing an inverse to $\widetilde{\varphi}$. Given a pointed Lie graph, we decompose it as an oriented graph together with a vertex-spider association $(G, \{S_v\})$. If we take the tree which comprises the interior of each spider as being a component of the forest, then we have Φ is the union of the interiors of the S_v. Since the choice of orientation of the vertices and edges is canonical, this completely determines the edge numbering in the resulting forested graph. $\square$

The boundary operator on the pointed Lie graph complex is defined as the sum of all possible edge collapses between the spider-decorated vertices. Upon collapsing an edge of a graph, we note that the two half-edges (spider legs) adjacent to the edge collapse are identified at their endpoints so as to create a new edge on the interior of the mating spiders. If we instead view this edge collapse as the addition of an edge between the vertices (with subsequent edge numbering falling after all previous numberings), we note that there is an induced boundary map on $pf\mathcal{G}$ from the boundary map on $p\overline{\mathcal{LG}}$. As such, we now speak

of the *pointed forested graph complex* where the boundary operator is defined as described above.

4.7 The main result

An analysis of the pointed forested graph complex is the last step in identifying the cohomology of Auter space with $H_\bullet(\ell_\infty; p_1\ell_\infty)$. In particular, we prove

Theorem 4.7.1. $H_k(pf\mathcal{G}^r) \cong H^{2r-1-k}(Aut(F_r))$

so that together with Propositions 4.4.4 and 4.5.4 we have the Kontsevich-type isomorphism

Theorem 4.7.2. $H_k(\ell_\infty; p_1\ell_\infty)^{\mathrm{co}\,\ell_\infty} \cong H_k(\mathfrak{sp}(2\infty)) \oplus \bigoplus_{r\geq 2} H^{2r-1-k}(Aut(F_r))$

We summarize the process by which we will determine Theorem 4.7.1. First, we define a filtration on $S\mathbb{A}_r$ and analyze the cohomology spectral sequence associated to the filtration. This spectral sequence collapses to a cochain complex which calculates the cohomology of $Aut(F_r)$. Finally, analysis of this cochain complex will allow us to relate it to the pointed forested graph complex.

Filter the spine $S\mathbb{A}_r$ by the number of vertices in a marked graph so that $F_0S\mathbb{A}_r$ consists of all trivalent (aside from the basepoint) graphs with $2r-1$ vertices. In general, we let $F_pS\mathbb{A}_r$ be generated by graphs with $2r-p-1$ vertices along with $F_{p-1}S\mathbb{A}_r$. The filtration is independent of the marking and so when we quotient by the Aut action, we preserve the filtration and so we have $F_0Q_r \subset F_1Q_r \subset \cdots \subset F_{2r-2}Q_r = Q_r$. With trivial $\mathbb{R}$-coefficients, the spectral sequence attached to this filtration has first page E_1 given by $H^{p+q}(F_pQ_r, F_{p-1}Q_r)$ and the spectral sequence converges to $H^*(Aut(F_r))$.

Note that if all of the terms on the E_1 page vanish except for those along the p-axis, then the spectral sequence collapses to the cochain complex

$$0 \to H^0(F_0Q_r) \to H^1(F_1Q_r, F_0Q_r) \to \cdots \to H^{2r-2}(F_{2r-2}Q_r, F_{2r-3}Q_r) \to 0$$

which calculates the cohomology of $Aut(F_r)$. Let us accept this fact for now and use it to verify

Theorem. $H_k(pf\mathcal{G}^r) \cong H^{2r-1-k}(Aut(F_r))$

Proof. For convenience, let $Q_p^{(r)}$ denote F_pQ_r. We analyze the complex

$$0 \to H^0(Q_0^{(r)}) \to H^1(Q_1^{(r)}, Q_1^{(r)}) \to \cdots \to H^{2r-2}(Q_{2r-2}^{(r)}, Q_{2r-3}^{(r)}) \to 0.$$

Note that at the pth node, we have

$$H^p(Q_p^{(r)}, Q_{p-1}^{(r)}) = \frac{ker(C^p(Q_p^{(r)}, Q_{p-1}^{(r)}) \xrightarrow{\delta} C^{p+1}(Q_p^{(r)}, Q_{p-1}^{(r)}))}{im(C^{p-1}(Q_p^{(r)}, Q_{p-1}^{(r)}) \xrightarrow{\delta} C^p(Q_p^{(r)}, Q_{p-1}^{(r)}))}.$$

Note that the $Aut(F_n)$-stabilizer of a cube (g, G, Φ) in the spine consists of the basepoint-preserving graph automorphisms of G that fix the forest. We say a basepoint-preserving graph automorphism is *odd* if it induces an odd permutation on the edges of the forest;

otherwise we say it is *even*. When we quotient a cube by its stabilizer, the result is either a cone on a rational homology ball (if there is an odd automorphism) or a cone on a rational homology sphere (if all automorphisms are even) by Corollary 3.2 of [17] so that $H^p(Q_p^{(r)}, Q_{p-1}^{(r)}) = C^p(Q_p^{(r)}, Q_{p-1}^{(r)})/im(C^{p-1}(Q_p^{(r)}, Q_{p-1}^{(r)}) \xrightarrow{\delta} C^p(Q_p^{(r)}, Q_{p-1}^{(r)}))$.

We claim

1. $C^p(Q_p^{(r)}, Q_{p-1}^{(r)})$ is the dual of the span of all basepointed graphs whose non-basepoint vertices are trivalent along with a p-edged forest modulo the relation AS given as $(G, -\Phi) = -(G, \Phi)$ and that
2. the image of δ is spanned by the dual of the IHX and pIHX relators.

To see that the result follows from the establishment of these two claims, note that the space spanned in either case can be identified with its dual and so we have the result

$$H^p(Q_p^{(r)}, Q_{p-1}^{(r)}) = \mathbb{R}\{(G, \Phi, *)\}/(pIHX, IHX, AS)$$

which one can observe is precisely the *vector space* $pf\mathcal{G}_{2r-1-p}$. Recall that the lattice points on the E_1 page are given by $H^{p+q}(Q_p^{(r)}, Q_{p-1}^{(r)})$ and so, in particular, the coboundary map on this page has bidegree $(1, 0)$. The E_1 page vanishes off the p-axis and so the coboundary map along this remaining cochain complex is the sum over adding an edge e to the forest Φ of the cube $[G, \Phi]$ in all possible ways so that $\Phi \cup e$ is a forest. This exactly describes the boundary map on the forested graph complex. That is, the cochain complex $H^\bullet(Q_p^{(r)}, Q_{p-1}^{(r)})$ is isomorphic to the chain complex $pf\mathcal{G}_{2r-1-p}$.

Now that we see how the theorem follows from the two claims, let us verify them; we start with the first. Note that the dimension of the simplex corresponding to an element of $Q_{p-1}^{(r)}$

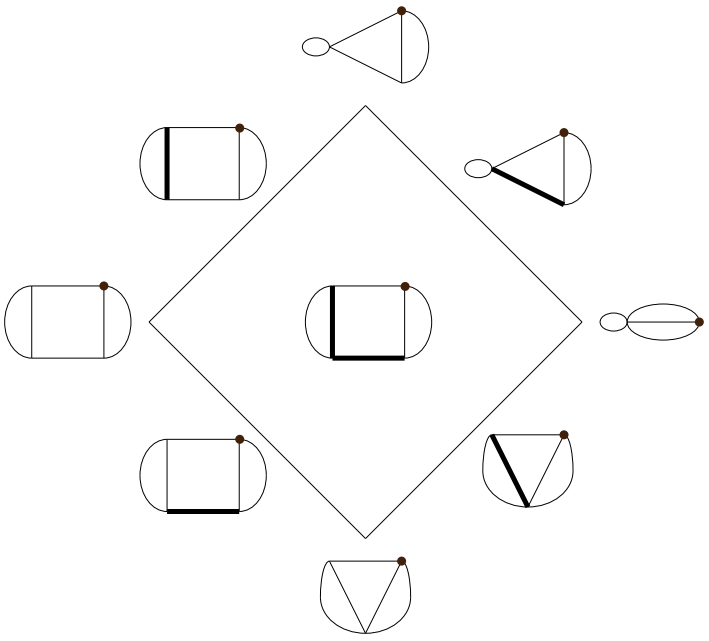


Figure 4.4: A cube in the 3-spine

has dimension $p-1$ and $C^p(Q_p^{(r)}, Q_{p-1}^{(r)})$ and may be identified with $C^p(Q_p^{(r)})$. Recall that $Q_\bullet^{(r)}$ is defined by a quotient: in the Aut action on a cube (g, G, Φ), the diagonal between $[G/\Phi, \emptyset]$ and $[G, \emptyset]$ is fixed. We then note there is a one-to-one correspondence between the cubes of $Q_p^{(r)}$ and the graphs that are basepointed trivalent (aside from the basepoint) with a forest Φ with p edges. The antisymmetry property follows by the combinatorial construction of the spine and the forest ordering. We have identified the cochain groups with the dual of the vector space spanned by basepointed trivalent graphs (excluding the basepoint) with a subforest modulo an antisymmetry relation.

We consider the image of the coboundary map. To see that the image coincides with IHX-type relators, we start at the level of the spine filtration and then pass to the quotient filtration. Suppose we remove the forest designation of an edge $e \in \Phi$ from the cube (g, G, Φ) in $F_pS\mathbb{A}_r$, that is, we consider $(\delta_R)|_e(g, G, \Phi)$. The image $(\delta_R)|_e(g, G, \Phi)$ is codimension one to the face (g, G, Φ). In addition, there is a codimension one face $(\delta_C)|_e(g, G, \Phi)$ of (g, G, Φ) for every face of the form $(\delta_R)|_e(g, G, \Phi)$ and this face is directly opposite it in the cube. Succinctly stated, there is a bijective correspondence between the codimension one faces resulting from contracting or removing an edge in the cube (g, G, Φ). See Figures 2.3 and 4.4 for a pictorial idea of the correspondence. By definition, $C^{p-1}(Q_p^{(r)}, Q_{p-1}^{(r)})$ consists of maps of dimension $p-1$ cubes in $Q_p^{(r)}$ that are zero on $Q_{p-1}^{(r)}$. By the bijective correspondence of the codimension one faces and the definition of the relative cochain groups, we may form a basis which consists of indicator functions on the dimension $p-1$ cubes of the form $(\delta_C)|_e(g, G, \Phi)$. Consider the vertex resulting from the collapse of the edge e in $(\delta_C)|_e(g, G, \Phi)$. There are precisely three codimension one faces which share this collapsed cube as a common face, see Figure 4.5. Erasing the marking by taking into account the Aut action, we see that these three blow-ups correspond to the three terms of an IHX relator in $pf\mathcal{G}$. Application of the coboundary map from the cochain complex to the contracted cube then gives $\delta((\delta_C)|_e(G, \Phi)) = (G, \Phi) + (G', \Phi') + (G'', \Phi'')$. Furthermore, if the basepoint is at a 3-valent vertex of the forest, we can grow out the hair for the basepoint and proceed as above so as to get the dual of an pIHX relator in the image of δ. Hence the image of δ is generated by the indicator functions on the IHX and pIHX relators.

□

Following [7], we say a complex is *k-spherical* if it is of dimension k and is homotopy equivalent to a wedge of k-spheres.

Lemma 4.7.3. *On the E_1 page, we have $H^{p+q}(Q_p^{(r)}, Q_{p-1}^{(r)})$ is zero off the p-axis.*

Proof. Since Aut acts with finite stabilizers and preserves the filtration, the homology of $Q_p^{(r)}/Q_{p-1}^{(r)}$ coincides with the homology of $F_pS\mathbb{A}_r/F_{p-1}S\mathbb{A}_r$ (modulo the Aut action). It therefore suffices to consider the pair $(F_pS\mathbb{A}_r, F_{p-1}S\mathbb{A}_r)$.

We want to characterize those elements in $F_pS\mathbb{A}_r - F_{p-1}S\mathbb{A}_r$. Given $(G, \Phi, *)$, grow out the basepoint $*$ to the end of a hair. If we consider the sum $\sum_v(|v| - 3)$ (exclude the univalent vertex at the tip of the hair), then the result is p if (G, Φ) is an element of $F_pS\mathbb{A}_r - F_{p-1}S\mathbb{A}_r$.

Suppose we have a basepointed graph where the basepoint sprouts off from a trivalent vertex, that is, prior to the hair growth, the vertex was bivalent. In this case, we may proceed exactly as in Proposition 22 of [7] by shrinking back down our hair so as to return

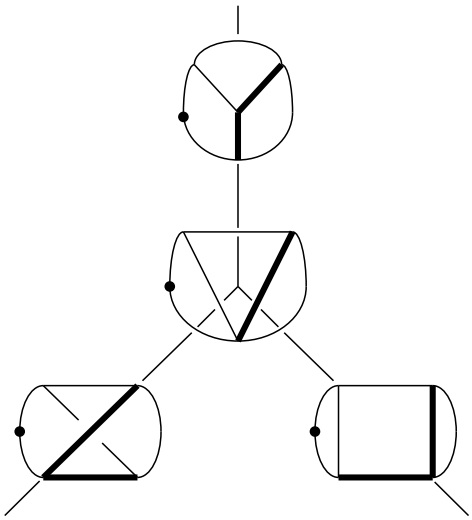


Figure 4.5: The three codimension-one faces of $(\delta_C)|_e(g, G, \Phi)$

the graph back to its locally bivalent state. This is because upon consideration of the blow-up complex, we need only consider vertices of valence larger than 3 and this case is handled by the proposition. Now, suppose our graph has a basepoint sprouting off from a vertex with valence $k \geq 4$ (so that the vertex was at least trivalent prior to the hair growing). The blow-up complex for such a vertex is $(k-4)$-spherical by Proposition 4.1 of [4]. A combinatorial interpretation of this is as follows: Prior to the hair growth, the blow-up complex of an at least trivalent vertex which coincides with the basepoint consists all blow-ups of the vertex in the non-basepoint sense and then a "distribution" of the basepoint to expand the blow-up further. When we grow the basepoint to the tip of a hair and consider the blow-ups of this (original valence + 1) vertex, we will distribute the hair across the blow-ups – this is precisely the same as the previous scenario. Now, further appeal to the cited result gives us that the quotient $F_pS\mathbb{A}_r/F_{p-1}S\mathbb{A}_r$ is p-spherical (the link of a basepointed marked graph $(g, G, *)$ is $(p-1)$-spherical in the $(p-1)$st level of the filtration and there are no edges between the elements of $F_pS\mathbb{A}_r - F_{p-1}S\mathbb{A}_r$). Since we are working with coefficients over the field $\mathbb{R}$ and $F_pS\mathbb{A}_r/F_{p-1}S\mathbb{A}_r$ is p-spherical, the cohomology group $H^{p+q}(Q_p^{(r)}, Q_{p-1}^{(r)})$ is nonzero only when $q = 0$, that is along the p-axis.

□

Bibliography

[1] M. Aguiar and S. Mahajan. *Monoidal Functors, Species and Hopf Algebras.*

[2] F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial Species and Tree-like Structures.* Encyclopedia of Mathematics and its Applications.

[3] F. Borceux. *Handbook of Categorical Algebra 1.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, New York, 1994.

[4] M. Bridson and K. Vogtmann. The symmetries of outer space. *Duke Math J*, 106(2):391–409, 2001.

[5] K. Brown. *Cohomology of Groups*, volume 87. Springer-Verlag, 1982.

[6] J. Conant and K. Vogtmann. Infinitesimal operations on graphs and graph homology.

[7] J. Conant and K. Vogtmann. On a theorem of Kontsevich. *Algebraic & Geometric Topology*, 3:1167–1224, 2003.

[8] J. Conant and K. Vogtmann. Morita classes in the homology of automorphism groups of free groups. *Geometry & Topology*, 8:1471–1499, December 2004.

[9] M. Culler and K. Vogtmann. Moduli of graphs and automorphisms of free groups. *Invent. Math.*, 84:91–119, 1986.

[10] W. Fulton and J. Harris. *Representation Theory: A First Course.* Readings in Mathematics. Springer-Verlag, New York, 1991.

[11] F. Gerlits. *Invariants in Chain Complexes of Graphs.* PhD thesis, Cornell, 2002.

[12] E. Getzler and M. Kapranov. Cyclic operads and cyclic homology. *Conf. Proc. Lecture Notes Geom. Topology, IV*, pages 167–201, 1995.

[13] A. Hagberg, D. Schult, and P. Swart. Exploring network structure, dynamics, and function using networkx. *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Aug. 2008.

[14] A. Hatcher. Homological stability for automorphism groups of free groups. *Comment. Math. Helv.*, 70(1):39–62, 1995.

[15] A. Hatcher. *Algebraic Topology.* Cambridge University Press, New York, seventh edition, 2006.

[16] A. Hatcher and K. Vogtmann. Cerf theory for graphs. *J. London Math Soc.*, 58:633–655, 1998.

[17] A. Hatcher and K. Vogtmann. Rational homology of $Aut(F_n)$. *Math Research Letters*, (5):759–780, 1998.

[18] A. Hatcher, K. Vogtmann, and N. Wahl. Erratum to: Homology stability for outer automorphisms groups of free groups. *Alg. & Geom. Top*, 6:573–579, 2006.

[19] A. Joyal. Une théorie combinatoire des séries formelles. *Adv. in Math.*, 42:1–82, 1981.

[20] A. Knapp. *Lie Groups, Lie Algebras, and Cohomology*, volume 34 of *Mathematical Notes*. Princeton University Press, Princeton, New Jersey, 1988.

[21] M. Kontsevich. Formal (non)commutative symplectic geometry. In *Gelfand Mathematical Seminars (1990-1992)*, pages 173–187. Birkhauser, 1993.

[22] M. Kontsevich. Feynmann diagrams and low-dimensional topology. In *Progr. Math.*, volume II of *First European Congress of Mathematics (Paris, 1992)*, pages 97–121. Birkhauser, 1994.

[23] S. M. Lane. *Category Theory for the Working Mathematician*. Springer-Verlag, 1971.

[24] T. Leinster. *Higher Operads, Higher Categories*, volume 298 of *London Mathmatical Society Lecture Note Series*. Cambridge University Press, 2004.

[25] S. Mahajan. Symplectic operad geometry and graph homology, Nov. 2002.

[26] M. Markl, S. Shnider, and J. Stasheff. *Operads in Algebra, Topology and Physics*, volume 96 of *Mathematical Surveys and Monographs*. AMS, Providence, RI, 2002.

[27] J. May. *The Geometry of Iterated Loop Spaces*, volume 271 of *Lecture Notes in Mathematics*. Springer-Verlag, New York, 1972.

[28] J. McCleary. *A User's Guide to Spectral Sequences*. Cambridge University Press, second edition, 2001.

[29] J. Milnor and J. Moore. On the structure of Hopf algebras. *The Annals of Mathematics*, 81(2):211–264, March 1965.

[30] B. Mitchell. *Theory of Categories*, volume 17. Academic Press, 1965.

[31] S. Morita. Structure of the mapping class groups of surfaces: a survey and a prospect. *Geom. Topol. Monogr.*, (2):349–406, 1999.

[32] R. Ohashi. The rational homology group of $Out(F_n)$ for $n \leq 6$. *Experiment. Math*, 17(2):167–179, 2008.

[33] J.-P. Serre. *Lie algebras and Lie groups*. Benjamin, 1965.

[34] M. Sweedler. *Hopf Algebras*. W. A. Benjamin Inc., New York, 1969.

[35] D. Thurston. Integral expressions for the Vassiliev knot invariants.

[36] K. Vogtmann. Automorphisms of free groups and outer space.

[37] C. Weibel. *An Introduction to Homological Algebra*, volume 38 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1994.

[38] A. Whitehead. *A Treatise on Universal Algebra*. Cambridge University Press, 1898.

Appendices

Appendix A

Python Code

A.1 diagram_gr.py

```
#!/usr/bin/env python
# encoding: utf-8
'''
Created by Jonathan Gray .
Copyright (c)  Jonathan Gray. All rights reserved.
'''

import sys
import os
import glob

# Chord Diagram Generator
# len(graph)=num chords in particular case

def cd_genr(num_chords):
  name_of_file="fg_"+str(num_chords)
  f=open(name_of_file, 'w')

  for i in xrange(2,2*num_chords+1,1):
    fn2="fg_"+str(num_chords-1)
    file2=open(fn2, 'r')

    for row in file2:
      graph=row.split()
      index=0

      for entry in graph:
        graph[index]=int(entry)
        index=index+1

      for j in range(0,len(graph),2):
        if graph[j]>i-2:
          graph[j]=graph[j]+2
        else:
          graph[j]=graph[j]+1
        if graph[j+1]>i-2:
          graph[j+1]=graph[j+1]+2
        else:
          graph[j+1]=graph[j+1]+1

      graph.insert(0,i)
      graph.insert(0,1)

      for entry in graph:
        f.write("%s " % entry)
      f.write("\n")

  f.close()

######################

top_case=4

print "The spaces fg_2, ... , fg_{0} will be writted to disk.\n".format(
    top_case)
```

```
contents=glob.glob('fg*') #pulls all files prefixed with fg from working dir

if 'fg_1' not in contents:
  print "\nFile fg_1 is missing... Creating fg_1 for you...\n"

  f=open("fg_1", 'w')
  f.write("1 2")
  f.close()

for index in xrange(2,top_case+1):
  if ("fg_"+str(index)) not in contents:
    print "Writing fg_{0} ...".format(index)
    diagrams = cd_genr(index)
  else:
    print "File fg_{0} already exists.  Jumping to next case...".format(index)
```

A.2 good_diagram_genr.py

```
#!/usr/bin/env python
# encoding: utf-8
'''
Created by Jonathan Gray .
Copyright (c)  Jonathan Gray. All rights reserved.
'''

import sys
import os
import glob

def testgood(graph,case):
  want=range(1,case+1)

  verts=[]
  for i in xrange(0,len(graph),2):
    verts.append(graph[i])
  if verts == want:
    return True
  else:
    return False

# Good Chord Diagram Generator
def cd_genr(num_chords):
  name_of_file="g"+str(num_chords)
  f=open(name_of_file, 'w')

  for i in xrange(2,2*num_chords+1,1):
    fn2="g"+str(num_chords-1)
    file2=open(fn2, 'r')

    for row in file2:
      graph=eval(row)

      for j in range(0,len(graph),2): #
        if graph[j]>i-2:
          graph[j]=graph[j]+2
        else:
          graph[j]=graph[j]+1
        if graph[j+1]>i-2:
          graph[j+1]=graph[j+1]+2
        else:
          graph[j+1]=graph[j+1]+1

      graph.insert(0,i)
      graph.insert(0,1)

      if testgood(graph,num_chords) == True:
        f.write("{0}\n".format(graph))

  f.close()
```

```
########################

top_case=8

print "The spaces g2, ... , g{0} will be writted to disk.\n".format(
    top_case)

contents=glob.glob('g*') #pulls all files prefixed with g from working
    dir

if 'g1' not in contents:
  print "\nFile g1 is missing... Creating g1 for you...\n"

  f=open("g1", 'w')
  f.write("[1, 2]")
  f.close()

for index in xrange(2,top_case+1):
  if ("g"+str(index)) not in contents:
    print "Writing g{0} ...".format(index)
    diagrams = cd_genr(index)
  else:
    print "File g{0} already exists.  Jumping to next case...".format(
        index)
```

A.3 reformat_and_filter.py

```
#!/usr/bin/env python
'''
Created by Jonathan Gray .
Copyright (c)  Jonathan Gray. All rights reserved.

Reads in good chord diagrams, reformats the list to endpoints only,
    kills diagrams with isolated or long chords, and keeps one in a pair
     of parallel chords [the longer one].
'''
import sys
import os
import glob

class Graph:
  "class to define a graph object with sign from IHX and forest
      relabelling"
  def __init__(self,graph,coeff):
    self.graph=graph
    self.coeff=coeff

def reformat_and_filter_core(num_chords):
  old_file="g"+str(num_chords)
  new_file="endpointg"+str(num_chords)

  f_old=open(old_file, 'r')
  f_new=open(new_file, 'w')

  for row in f_old:
    graph=eval(row)
    index=0

    for entry in graph:
      graph[index]=int(entry)
      index=index+1

    graph=reformat(graph)
    graph=Graph(graph,1)
    graph=quotient_out(graph,num_chords)

    if graph.coeff != 0:
      f_new.write("{0}|{1}\n".format(graph.graph, graph.coeff))

  f_new.close()
  f_old.close()

def reformat(graph):
  endpoints=[]

  for i in xrange(1,len(graph),2):
    endpoints.append(graph[i])
```

```
  return endpoints

def quotient_out(g,case):
  g=isolated_chord(g,case)
  g=long_chord(g,case)

  if g.coeff != 0:
    g=parallel_chords(g,case)
    g=parallel_chords(g,case)

  return g

def isolated_chord(g,case):
  if g.graph[-1] == case+1:
    g.coeff=0
    return g
  else:
    return g

def long_chord(g,case):
  v0=g.graph[0]
  v1=g.graph[1]

  if (v0 == 2*case) or (v0 == 2*case-1) or (v1 == 2*case):
    g.coeff=0
    return g
  else:
    return g

def parallel_chords(g,case):
  for i in xrange(case-1):
    left=g.graph[i]
    right=g.graph[i+1]

    if left==(right+1):
      g.graph[i]=right
      g.graph[i+1]=left
      g.coeff=-1*g.coeff
      break

  return g

########################
case=8
reformat_and_filter_core(case)
```

A.4 mc_*#.py

```
#!/usr/bin/env python
'''
Created by Jonathan Gray .

n_shoots_monster() describes the attaching of a monster M
to the linear tree L where there are n edges from M attaching to L.
E.g., in fG_2, a relation is generated via the 3-shoots-monster where
a length one forest edge is attached at the midpoint of a line segment
and two nonforest edges are attached symmetrically at the endpoints of
the line segment:  ft=[[a1,a2],[a2,a3],[a2,1]], nft=[[a1,1],[a3,1]]

Throughout,

Graph=[[ft],[nft]]
ft = forested edges of Graph given as a list
nft = nonforested edges of Graph given a sa list


'''

#from good_diag_enum import make_dict
#from math import factorial
import pickle
#import time
#import numpy.linalg as nla
#import numpy
#import scipy
#import pp
#import itertools
#import collections
import copy


class MonsteredGraph:
    '''
    class to define a graph of (top_dim-1) and list of edges that can be
        added
    via the bdy map d (d=add edge type bdy map from bicomplex)
    The .bdy stub holds the result of adding the forested edge via d and
    subsequently enumerating all possible diagrams with that monster.

    .graph=base graph with monster attached given as [[ft],[nft]]
    .edges=edges of monster that can be added to submaximal tree (list of
        tuples)
    .components=each of the terms from taking bdy map of monstered graph
    .lvnu=the vertices which can be utilized to create chords

    nonforested edge is initially stored in nft
    the edge is then remove via the del statements in the
        create_components method
```

```
    The .create_monstered_diagrams method has a special parsing in the
        event
    that the passed (non-forest) chord list for appending is empty.
    '''
    def __init__(self,ft,nft,viable_edges,components,lvnu,chords,monsters)
        :
      self.ft=ft
      self.nft=nft
      self.edges=viable_edges
      self.components=components
      self.lvnu=lvnu
      self.chords=chords
      self.monsters=monsters

    def create_components(self):
      for edge in self.edges:

        i=int(edge[0][1:])
        m=6 # 2*num_chords - 2
        n=2 # number of edges in forest of monster

        sign=(-1)**(n*(i+1)+m-i)

        new_graph_ft=self.ft[:]
        new_graph_nft=self.nft[:]

        ind=new_graph_nft.index(edge[0])

        del new_graph_nft[ind+1]
        del new_graph_nft[ind]

        new_graph_ft.extend(list(edge))
        new_graph=[new_graph_ft,new_graph_nft,sign]

        self.components.append(new_graph)

    def create_chord_configurations(self):
      bijection={}
      i=1

      for v in self.lvnu:
        bijection[i]=v
        i=i+1

      chord_list=[]
      dimn=len(self.lvnu)/2

      if dimn!=0:
        f=open("./include/fg_"+str(dimn))

        for line in f:
          line=line.split(' ')
          if line[-1]=='\n':
```

```
            line=line[:-1]
          line=self.remap(line,bijection)
          self.chords.append(line)

        f.close()

      if dimn==0:
        pass

    def remap(self,chords,bijection):
      new_chords=[0]*len(chords)
      i=0

      for entry in chords:
        new_chords[i]="a"+str(bijection[int(entry)])
        i=i+1

      return new_chords

    def create_monstered_diagrams(self):
      if self.chords==[]:
        monster_component=[]

        for comp in self.components:
          new_ft=comp[0]
          new_nft=comp[1][:]
          monster_component.append([new_ft,new_nft,comp[2]])

        self.monsters.append(monster_component)

      else:
        for c in self.chords:
          monster_component=[]

          for comp in self.components:
            new_ft=comp[0]
            new_nft=comp[1][:]
            new_nft.extend(c)
            monster_component.append([new_ft,new_nft,comp[2]])

          self.monsters.append(monster_component)

# yes, cheesy definition.
def factorial(n):
  if n==1:
    return 1
  if n==2:
    return 2
  if n==3:
    return 6
  if n==4:
    return 24
  if n==5:
```

```
    return 120
  if n==6:
    return 720
  if n==7:
    return 5040

def icr(my_dict,good_diags,num_chords,slots):
  new_rows=[]

  for graph in good_diags:
    if graph[-1] == (num_chords+1):
      new_row=[0]*slots
      new_row[my_dict[str(graph)]]=1
      new_rows.append(new_row)

  file_icr="ic_"+str(num_chords)
  f1=open(file_icr,'w')

  for reln in new_rows:
    for entry in reln:
      f1.write("{0} ".format(entry))
    f1.write("\n")
  f1.close()

def lcr(my_dict,good_diags,num_chords,slots):
  new_rows=[]

  for graph in good_diags:
    v0=graph[0]
    v1=graph[1]

    if (v0 == 2*num_chords) or (v0 == 2*num_chords-1) or (v1 == 2*
        num_chords):
      new_row=[0]*slots
      new_row[my_dict[str(graph)]]=1
      new_rows.append(new_row)

  file_lcr="lc_"+str(num_chords)
  f2=open(file_lcr,'w')

  for reln in new_rows:
    for entry in reln:
      f2.write("{0} ".format(entry))
    f2.write("\n")
  f2.close()

def pcr(my_dict,good_diags,num_chords,slots):
  new_rows=[]

  for graph in good_diags:
    new_row=para_parse(graph,my_dict,good_diags,num_chords,slots)
    if new_row != False:
      new_rows.append(new_row)
```

```
  file_pcr="pc_"+str(num_chords)
  f3=open(file_pcr,'w')

  for reln in new_rows:
    for entry in reln:
      f3.write("{0} ".format(entry))
    f3.write("\n")
  f3.close()

def get_diags(num_chords):
  f=open("endpointg"+str(num_chords))
  good_diags=[]

  for line in f:
    line=line.split('|')
    good_diags.append(eval(line[0]))

  return good_diags

def para_parse(graph,my_dict,good_diags,num_chords,slots):
  for i in xrange(num_chords-1):
    left=graph[i]
    right=graph[i+1]

    if left==(right-1):
      other_graph=make_other(graph,left,right,i)
      new_row=[0]*slots
      new_row[my_dict[str(graph)]]=1
      new_row[my_dict[str(other_graph)]]=1
      return new_row

  return False

def make_other(graph,left,right,i):
  other_graph=[]

  for v in graph:
    other_graph.append(v)

  other_graph[i]=right
  other_graph[i+1]=left

  return other_graph

def flatten(array):
    i=0

    while i<len(array):
        while isinstance(array[i], list):

            if not array[i]:
                array.pop(i)
```

```
                i -= 1
                break
            else:
                array[i:i+1] = array[i]

        i += 1

    return array

def two_shoots_morita_monster(num_chords,slots):
  # requires two chords/four vertices
  # range for lvnu is because of assumption two "top" vertices in
  # forest carry the labels 2n-1 and 2n where n=num_chords

  monsters=[]

  for a in range(1,2*num_chords-2):
    for b in range(a+1,2*num_chords-1):
      lvnu=range(1,2*num_chords-1)
      lvnu.remove(a)
      lvnu.remove(b)
      monsters.append([
      ["a"+str(a),2*num_chords-1],
      ["a"+str(b),2*num_chords],
      lvnu])

  all_monsters=[]

  v0=2*num_chords-1
  v1=2*num_chords
  ft=[v0,v1]

  for monster in monsters:
    nft=flatten(monster[:-1])
    nft.extend([v0,v1])
    ve=monster[:-1]
    MonsterInstance=MonsteredGraph(ft,nft,ve,[],monster[-1],[],[])
    MonsterInstance.create_components()
    MonsterInstance.create_chord_configurations()
    MonsterInstance.create_monstered_diagrams()
    all_monsters.append(MonsterInstance)

  return all_monsters

def three_shoots_monster(num_chords,slots):

  # requires two chords/four vertices

  monsters=[0]*(((num_chords-1)*(4*(num_chords-1)**2-1))/3)

  # three for loops:  each picks a vertex use one less vertex for "
      forest vert"
  # equivalent to generate all tuples (a,b,c) s.t. a < b < c
```

```
    # there are n(4n^2-1)/3 such tuples

    i=0

    for v0 in range(1,2*num_chords-2):
      for v1 in range(v0+1,2*num_chords-1):
        for v2 in range(v1+1,2*num_chords):
          lvnu=range(1,2*num_chords)
          lvnu.remove(v0)
          lvnu.remove(v1)
          lvnu.remove(v2)
          monsters[i]=[
          ["a"+str(v0),2*num_chords],
          ["a"+str(v1),2*num_chords],
          ["a"+str(v2),2*num_chords],
          lvnu]
          i=i+1

    all_monsters=[] #all class instances of the monsters
    ft=[]

    for monster in monsters:
      nft=flatten(monster[:-1])
      ve=monster[:-1]
      MonsterInstance=MonsteredGraph(ft,nft,ve,[],monster[-1],[],[])
      MonsterInstance.create_components()
      MonsterInstance.create_chord_configurations()
      MonsterInstance.create_monstered_diagrams()
      all_monsters.append(MonsterInstance)

    return all_monsters

def three_shoots_morita_monster(num_chords,slots):
    # requires three chords/six vertices
    # range for lvnu is because of assumption two "top" vertices in forest
    # carry the labels 2n-1 and 2n where n=num_chords

    monsters=[]
    choose=range(1,2*num_chords-2)
    for a in choose:
      for b in choose:
        for c in choose:
          if (a<c) and (a!=b) and (a!=c) and (b!=c):
            lvnu=range(1,2*num_chords-1)
            lvnu.remove(a)
            lvnu.remove(b)
            lvnu.remove(c)
            monsters.append([
            ["a"+str(a),2*num_chords-2],
            ["a"+str(b),2*num_chords-1],
            ["a"+str(c),2*num_chords],
            lvnu])
```

```
    all_monsters=[]

    v0=2*num_chords-2
    v1=2*num_chords-1
    v2=2*num_chords
    ft=[v0,v1,v1,v2]

    for monster in monsters:
      nft=flatten(monster[:-1])
      nft.extend([v0,v2])
      ve=monster[:-1]
      MonsterInstance=MonsteredGraph(ft,nft,ve,[],monster[-1],[],[])
      MonsterInstance.create_components()
      MonsterInstance.create_chord_configurations()
      MonsterInstance.create_monstered_diagrams()
      all_monsters.append(MonsterInstance)

    return all_monsters

def four_shoots_monster(num_chords,slots):
    # requires three chords/six vertices
    # method = generate all tuples ([2n],[2n],[2n],[2n]) subject to the
    # topological sort detailed in the if statement
    # range for lvnu is because of assumption two "top" vertices in forest
    # carry the labels 2n-1 and 2n where n=num_chords

    monsters=[]
    choose=range(1,2*num_chords-1)

    for a in choose:
      for b in choose:
        for c in choose:
          for d in choose:
            if (a<c<d) and (a<b) and (a!=b) and (b!=c) and (c!=d) and (b!=
                d):
              lvnu=range(1,2*num_chords-1)
              lvnu.remove(a)
              lvnu.remove(b)
              lvnu.remove(c)
              lvnu.remove(d)
              monsters.append([
              ["a"+str(a),2*num_chords-1],
              ["a"+str(b),2*num_chords-1],
              ["a"+str(c),2*num_chords],
              ["a"+str(d),2*num_chords],
              lvnu])

    all_monsters=[]
    v0=2*num_chords-1
    v1=2*num_chords
    ft=[v0,v1]

    for monster in monsters:
```

```
    nft=flatten(monster[:-1])
    ve=monster[:-1]
    MonsterInstance=MonsteredGraph(ft,nft,ve,[],monster[-1],[],[])
    MonsterInstance.create_components()
    MonsterInstance.create_chord_configurations()
    MonsterInstance.create_monstered_diagrams()
    all_monsters.append(MonsterInstance)

  return all_monsters

def four_shoots_morita_monster(num_chords,slots):
  # requires 4 chords/eight vertices
  # range for lvnu is because of assumption two "top" vertices in forest
  # carry the labels 2n-1 and 2n where n=num_chords

  monsters=[]
  choose=range(1,2*num_chords-3)
  for a in choose:
    for b in choose:
      for c in choose:
        for d in choose:
          if (a<d) and (a!=b) and (a!=c) and (b!=c) and (b!=d) and (c!=d
              ):
            lvnu=range(1,2*num_chords-3)
            lvnu.remove(a)
            lvnu.remove(b)
            lvnu.remove(c)
            lvnu.remove(d)
            monsters.append([
            ["a"+str(a),2*num_chords-3],
            ["a"+str(b),2*num_chords-2],
            ["a"+str(c),2*num_chords-1],
            ["a"+str(d),2*num_chords],
            lvnu])

  all_monsters=[]

  v0=2*num_chords-3
  v1=2*num_chords-2
  v2=2*num_chords-1
  v3=2*num_chords
  ft=[v0,v1,v1,v2,v2,v3]

  for monster in monsters:
    nft=flatten(monster[:-1])
    nft.extend([v0,v3])
    ve=monster[:-1]
    MonsterInstance=MonsteredGraph(ft,nft,ve,[],monster[-1],[],[])
    MonsterInstance.create_components()
    MonsterInstance.create_chord_configurations()
    MonsterInstance.create_monstered_diagrams()
    all_monsters.append(MonsterInstance)
```

```
  return all_monsters

##########################################################################

num_chords=4
slots=factorial(num_chords) #slots = number of diagrams per reln

relations=[]

#edit the following to get desired monster type generated

#mon=four_shoots_monster(num_chords,slots)
#mon=four_shoots_morita_monster(num_chords,slots)
mon=three_shoots_morita_monster(num_chords,slots)

reln_num=1
folder_number=1

for monster_type in mon:
  for bundle in monster_type.monsters:
    po=open("./include/4/mor3/"+"/"+str(num_chords)+"mor3_"+str(reln_num
        ),'wb')
    pickle.dump(bundle,po)
    po.close()
    reln_num+=1
```

A.5 str.py

```
#!/usr/bin/env python
# encoding: utf-8

'''
Created by Jonathan Gray .
Copyright (c)  Jonathan Gray. All rights reserved.

 Usage:
 Vertices of linear forest must be entered as "strings" while vertices
    of sprouting
 tree are all given as numbers.  Further, the assumption is made that
    the ordering
 of the forest proceeds linearly from left to right.  As in, the edge
    before the
 tree is n-1, the first edge in the tree is n, and the edge after the
    tree is n+k
 where k-1 is the number of (forested) edges in the tree.

 The vertices of the sprouted tree are numbered consecutively from left
    to right



 The class Graph has three stubs:
 graph.g = networkx instance of the graph structure, complete with
    forest labelling
 graph.orien = a \pm 1 corresponding to orien determined in the end by
    foo


 CAUTION:
 when breaking down sprouts, the instance of an isolated chord on a
    sprouting tree
 is not permitted.  Does not remove generality either, for such diagrams
     are automatically zero.
'''

import pickle
from copy import *
from networkx import *
import sys
#import profile
#import time
#import networkx as nx
#import matplotlib.pyplot as plt
#import Tkinter

class Graph:
  "class to define a graph object with sign from IHX and forest
     relabelling"
  def __init__(self,graph,p_m_one):
```

```
    self.g=graph
    self.orien=p_m_one

def are_there_isolated_long(graph,case):
  print graph

  for i in range(0,2*case,2):
    v0=graph[i]
    v1=graph[i+1]

    if abs(v0-v1)==1 or abs(v0-v1)==(2*case-1) or abs(v0-v1)==(2*case-2)
        :
      return True

  return False

def filter_small_chords(to_filter,case):
  for graph in to_filter:
    for edge in graph.g:
      v0=edge[0]
      v1=edge[1]

      if abs(v0-v1)<=2:
        return []

  return to_filter

def are_there_isolated_long_tuple(graph,case):
  for edge in graph:
    v0=edge[0]
    v1=edge[1]

    if abs(v0-v1)==1 or abs(v0-v1)==(2*case-1) or abs(v0-v1)==(2*case-2)
        :
      return True

  return False

def fix_nbr_edge_for_orientation(nbr_edge):
  # returns edge so that
  # (str1,str2) is s.t. str1<str2
  # (str,int) is returned if one is int and the other str
  # (int1,int2) is s.t. int1 < int2

  v0=nbr_edge[0]
  v1=nbr_edge[1]
  ef=nbr_edge[2]
  tv0=type(v0)
  tv1=type(v1)

  if (tv0 is str) and (tv1 is str):
    v0n=int(v0[1:])
    v1n=int(v1[1:])
```

```
      if v0n<v1n:
        return (v0,v1,ef)
      if v0n>v1n:
        return (v1,v0,ef)

    if (tv0 is str) and (tv1 is int):
      return (v0,v1,ef)

    if (tv0 is int) and (tv1 is str):
      return (v1,v0,ef)

    if (tv0 is int) and (tv1 is int):
      if v0<v1:
        return (v0,v1,ef)
      if v0>v1:
        return (v1,v0,ef)

def reorder_nbr1_nbr2(nbr1,nbr2):
  #We assume forest edge will always be nbr2 if one is forest and the
      other not.

    if nbr1[2]=='y' and nbr2[2]=='n':
      return nbr2,nbr1

    if nbr1[2]=='n' and nbr2[2]=='y':
      return nbr1,nbr2

    else:
      return nbr1,nbr2

def fix_orientation(graph,vert,vep):
    '''
    The sign is dependent on the position of the tree relative to the
       placement of the new
    forest edge.  If the tree is placed after the new forest edge in the
       linear forest, then
    the orientation remains the same.  If the tree is place before the new
        linear forest edge,
    then one needs to count the number of forest edges in the tree (
       excluding the edge just
    operated upon) and determine how many transpositin are required to
       reorder the tree in a
    manner consistent with the rest of the graph.  In the case of a two
       sprout with non-forest
    edge to the right the H-term requires an orientation fix because the
       "2" will fall after
    the sprouting tree.  It requires the transposition (2 3) to reorder
       the graph and so H will
    carry a negative sign.  The X-term caused the "2" to precede the tree
       and hence no reordering
    is necessary.
```

```
The algorithm then becomes:  find first operable edge from "left to
    right" and then count the
number of forested edges in sprouting tree --above the operable edge
    --.  Then determine which
of the H/X terms will cause the new linear forested edge to precede
    the sprouting tree.  The
H/X term with the edge that succeeds the tree will carry the sign
                (-1)**(# forested edges in tree above operable edge).
'''

G=graph.g

nbrs_of_vep=[]

for nbr in G[vep]: #want (str,int)
  if type(nbr) is str:
    nbrs_of_vep.append((nbr,vep,G[nbr][vep][0]['forest']))
  if type(nbr) is int:
    nbrs_of_vep.append((vep,nbr,G[nbr][vep][0]['forest']))

nbrs_of_vep.remove((vert,vep,'y'))

# get all forested edges:

forested_edges=[(e[0],e[1]) for e in G.edges(data=True) if e[2]['
    forest']=='y']

# remove operable edge:
try:
  forested_edges.remove((vert,vep))
except:
  forested_edges.remove((vep,vert))

#create new graph from the forested edges:
component_graph=MultiGraph()

component_graph.add_edges_from(forested_edges)

components=connected_component_subgraphs(component_graph)

# pick component which connected to operable edge:
relevant_component=[]

for component in components:
  for edge in component.edges():
    if vep in edge:
      relevant_component.extend(component.edges())
      break

try:
  nbr1=nbrs_of_vep[0]
  nbr2=nbrs_of_vep[1]
except:
```

```
  return graph.orien,graph.orien

endpoints_of_nbrs_edges=[nbr1[1],nbr2[1]]

if nbr1[2]=='y' and nbr2[2]=='y':
  problem_edges=[sorted([edge[0],edge[1]]) for edge in
      relevant_component if vep in edge]

  LR_graph=MultiGraph()
  LR_graph.add_edges_from(relevant_component)

  LR_graph.remove_edge(problem_edges[0][0],problem_edges[0][1])
  LR_graph.remove_edge(problem_edges[1][0],problem_edges[1][1])

  LR_graph.add_edge('l',min(endpoints_of_nbrs_edges))
  LR_graph.add_edge('r',max(endpoints_of_nbrs_edges))

  LR_components=connected_component_subgraphs(LR_graph)

  for component in LR_components:
    for edge in component.edges():
      if 'l' in edge:
        L_length=len(component)/2

      if 'r' in edge:
        R_length=len(component)/2

  pm_one_h=graph.orien*(-1)**(L_length)
  pm_one_x=graph.orien*(-1)**(R_length+L_length)

  return pm_one_h,pm_one_x

if nbr1[2]=='y' and nbr2[2]=='n':
  '''
  figure out if non-forest edge attaches to lin-forest, or sprouting
      tree.
  if to lin-forest, then determine on which side of vert it appears.
  '''

  if type(nbr2[0]) == int and type(nbr2[1]) == int: #hence, connects
      to sprouting tree
    pm_one_gh=graph.orien
    pm_one_gx=graph.orien*(-1)**len(relevant_component)

    return pm_one_gh,pm_one_gx

  if type(nbr2[0]) == str and type(nbr2[1]) == int: #hence, connects
      to lin forest
    nbr2_num=int(nbr2[0][1:])
    vert_num=int(vert[1:])

    if nbr2_num<vert_num:
      pm_one_gh=graph.orien
```

```
        pm_one_gx=graph.orien*(-1)**len(relevant_component)

        return pm_one_gh,pm_one_gx

      if nbr2_num>vert_num:
        pm_one_gh=graph.orien*(-1)**len(relevant_component)
        pm_one_gx=graph.orien

        return pm_one_gh,pm_one_gx

    if type(nbr2[0]) == int and type(nbr2[1]) == str: #hence, connects
        to lin forest
      # ERROR, order on edges not respected somehow
      # Cannot reach this case
      nbr2_num=int(nbr2[0][1:])
      vert_num=int(vert[1:])



if nbr1[2]=='n' and nbr2[2]=='y':
    '''
    figure out if non-forest edge attaches to lin-forest, or sprouting
        tree.
    if to lin-forest, then determine on which side of vert it appears.
    '''

    if type(nbr1[0]) == int and type(nbr1[1]) == int: #hence, connects
        to sprouting tree
      pm_one_gx=graph.orien
      pm_one_gh=graph.orien*(-1)**len(relevant_component)

      return pm_one_gh,pm_one_gx

    if type(nbr1[0]) == str and type(nbr1[1]) == int: #hence, nbr1
        connects to lin forest
      nbr1_num=int(nbr1[0][1:])
      vert_num=int(vert[1:])

      if nbr1_num<vert_num:
        pm_one_gh=graph.orien
        pm_one_gx=graph.orien*(-1)**len(relevant_component)

        return pm_one_gh,pm_one_gx

      if nbr1_num>vert_num:
        pm_one_gh=graph.orien*(-1)**len(relevant_component)
        pm_one_gx=graph.orien

        return pm_one_gh,pm_one_gx


    if type(nbr1[0]) == int and type(nbr1[1]) == str:
      print "PROBLEM!, about line 211 "
```

```
        #will never hit this case
        return 1,1

    if nbr1[2]=='n' and nbr2[2]=='n':

      return graph.orien,graph.orien

def lists_to_networkx_graph(ft,nft):
    efl=[]
    enfl=[]

    #create tuples for edges

    for i in xrange(0,len(ft),2):
      efl.append((ft[i],ft[i+1]))

    for i in xrange(0,len(nft),2):
      enfl.append((nft[i],nft[i+1]))

    g0=MultiGraph()
    g0.add_edges_from(efl,forest="y")
    g0.add_edges_from(enfl,forest="n")

    return g0

def pickedge(G):
    # returns edge to operate upon with vep being of type int
    # First loop gets all edges that are operable and forces entries in
    # list of operable edges to be [str, int, str, int, ...]
    # numbers.append() in for loop just grabs numbers from str vertices
    # so the leftmost vertex may be picked via min().

    graph=G

    operable_verts=[]
    numbers=[]

    for e in graph.edges(data=True):
      e2for=e[2]['forest']=='y'

      if e2for and type(e[0]) is str and type(e[1]) is int:
        operable_verts.extend([e[0],e[1]])
        numbers.append(int(e[0][1:]))

      if e2for and type(e[1]) is str and type(e[0]) is int:
        operable_verts.extend([e[1],e[0]])
        numbers.append(int(e[1][1:]))

    leftmost_operable_vert="a"+str(min(numbers))
    index_vert_cnntd_to_leftmost_op_vert=operable_verts.index(
       leftmost_operable_vert)

    vert=leftmost_operable_vert
```

```
    vep=operable_verts[index_vert_cnntd_to_leftmost_op_vert+1]

    return vert,vep

def getnbrs(G,vert,vep):
    nbrs=[]

    for nbr in G.neighbors(vep):
        nbrs.append(nbr)

    bad_nbrs=[vertex for vertex in nbrs if type(vertex)==str]

    try:
        bad_nbrs.remove(vert)
    except:
        pass

    good_nbrs=[vertex for vertex in nbrs if type(vertex)!=str]

    return nbrs,bad_nbrs,good_nbrs

def ihx(graph):
    G=graph.g

    vert,vep=pickedge(G)


    ###
    ### Get nbrs
    ###

    nbrs=[]

    for nbr in G.neighbors(vep):
        nbrs.append(nbr)

    bad_nbrs=[vertex for vertex in nbrs if type(vertex)==str]

    try:
        bad_nbrs.remove(vert)
    except:
        pass

    good_nbrs=[vertex for vertex in nbrs if type(vertex)!=str]

    ###
    ### Procress edge
    ###

    vert_number=int(vert[1:])

    if len(good_nbrs)==0: # means vert,vep is a single sprout <=> bad_nbrs
        has length 2
```

```
    gh,gx=len_zero(G,vert,vep,bad_nbrs,good_nbrs,vert_number)

  if len(good_nbrs)==1:
    gh,gx=len_one(G,vert,vep,bad_nbrs,good_nbrs,vert_number)

  if len(good_nbrs)==2:
    gh,gx=len_two(G,vert,vep,bad_nbrs,good_nbrs,vert_number)

  gh_orien,gx_orien=fix_orientation(graph,vert,vep)

  ghc=Graph(gh,gh_orien) #ghc := gh class
  gxc=Graph(gx,gx_orien)

  ghc.orien=ghc.orien*(-1) # apply sign from I = -H - X
  gxc.orien=gxc.orien*(-1)

  ghc.g=renumber_tree_vertices_gh(ghc.g)
  gxc.g=renumber_tree_vertices_gx(gxc.g)

  return ghc,gxc

def comp(v1,v2):
  #Manual comparison of vertex -- via a proper order relation
  # python compares strings lexiographically, bad b/c then a11 < a2 !!!

  if (type(v1)==str) and (type(v2))==str:
    v1_number=int(v1[1:])
    v2_number=int(v2[1:])

    if v1_number < v2_number:
      return True
    if v2_number < v1_number:
      return False

  else:
    return (v1<v2)

def get_new_edges(G,vert_number,vert):
  valid_edges=[]
  edges_to_remove=[]

  for edge in G.edges(data=True):
    if comp(vert,edge[0]) and comp(vert,edge[1]):
      e0n=int(edge[0][1:])        #e0n=vert 0 number
      e1n=int(edge[1][1:])        #e1n=vert 1 number

      valid_edges.append(['a'+str(e0n+1),'a'+str(e1n+1),edge[2]['forest'
          ]])
      edges_to_remove.append([edge[0],edge[1]])

    elif comp(vert,edge[0]) and comp(edge[1],vert):
      e0n=int(edge[0][1:])
```

```
        valid_edges.append(['a'+str(e0n+1),edge[1],edge[2]['forest']])
        edges_to_remove.append([edge[0],edge[1]])

      elif comp(edge[0],vert) and comp(vert,edge[1]):
        e1n=int(edge[1][1:])

        valid_edges.append(['a'+str(e1n+1),edge[0],edge[2]['forest']])
        edges_to_remove.append([edge[0],edge[1]])

      elif comp(edge[0],vert) and comp(edge[1],vert):
        pass

  return valid_edges,edges_to_remove

def len_zero(G,vert,vep,bad_nbrs,good_nbrs,vert_number):
  # both edges hit the linear forest

  gh=deepcopy(G)
  gx=deepcopy(G)

  bad_nbrs=sorted(bad_nbrs,comp)

  n0=bad_nbrs[0]
  n1=bad_nbrs[1]

  n0_number=int(n0[1:])
  n0plusone='a'+str(n0_number+1)

  n1_number=int(n1[1:])
  n1plusone='a'+str(n1_number+1)

  vertplusone='a'+str(vert_number+1)

  gh.remove_edge(vert,vep)
  gx.remove_edge(vert,vep)

  gh.remove_edge(vep,n0)
  gh.remove_edge(vep,n1)

  gx.remove_edge(vep,n0)
  gx.remove_edge(vep,n1)

  new_edges,edges_to_remove=get_new_edges(gh,vert_number,vert)

  gh.remove_edges_from(edges_to_remove)
  gx.remove_edges_from(edges_to_remove)

  for e in new_edges:
    gh.add_edge(e[0],e[1],forest=e[2])
    gx.add_edge(e[0],e[1],forest=e[2])

  n0vert=comp(n0,vert)
  vertn1=comp(vert,n1)
```

```
    if n0vert and vertn1:
      gha=(n0,vert)
      ghb=(vertplusone,n1plusone)
      gxa=(n0,vertplusone)
      gxb=(vert,n1plusone)

    if comp(n1 , vert) and comp(vert , n0):
      gha=(n1,vert)
      ghb=(vertplusone,n0plusone)
      gxa=(n1,vertplusone)
      gxb=(vert,n0plusone)

    if comp(vert , n0) and comp(n0 , n1):
      gha=(vert,n1plusone)
      ghb=(vertplusone,n0plusone)
      gxa=(vert,n0plusone)
      gxb=(vertplusone,n1plusone)

    if vertn1 and comp(n1 , n0):
      gha=(vert,n0plusone)
      ghb=(vertplusone,n1plusone)
      gxa=(vert,n1plusone)
      gxb=(vertplusone,n0plusone)

    if comp(n0 , n1) and comp(n1 , vert):
      gha=(n0,vertplusone)
      ghb=(n1,vert)
      gxa=(n0,vert)
      gxb=(n1,vertplusone)

    if comp(n1 , n0) and n0vert:
      gha=(n1,vertplusone)
      ghb=(n0,vert)
      gxa=(n1,vert)
      gxb=(n0,vertplusone)

    gh.add_edge(*gha,forest='n')
    gh.add_edge(*ghb,forest='n')
    gx.add_edge(*gxa,forest='n')
    gx.add_edge(*gxb,forest='n')

    return gh,gx

def len_one(G,vert,vep,bad_nbrs,good_nbrs,vert_number):
    gh=deepcopy(G)
    gx=deepcopy(G)
    nbrs=[]

    if bad_nbrs!=[]:
      bad_nbr_number=int(bad_nbrs[0][1:])

    for nbr in G.neighbors(vep):
```

```
    nbrs.append(nbr)

  nbrs.remove(vert)

  gh.remove_edge(vert,vep)
  gx.remove_edge(vert,vep)

  if gh.number_of_edges(vep,nbrs[0])!=2:
    if bad_nbr_number>vert_number:

      fg=gh[vep][good_nbrs[0]][0]['forest']
      fb=gh[vep][bad_nbrs[0]][0]['forest']

      gh.remove_edge(vep,nbrs[0])
      gh.remove_edge(vep,nbrs[1])

      new_edges,edges_to_remove=get_new_edges(gh,vert_number,vert)

      gh.remove_edges_from(edges_to_remove)

      for e in new_edges:
        gh.add_edge(e[0],e[1],forest=e[2])

      gha=(vert,good_nbrs[0])
      ghb=('a'+str(vert_number+1),'a'+str(bad_nbr_number+1))
      gh.add_edge(*gha,forest=fg)
      gh.add_edge(*ghb,forest=fb)

    if bad_nbr_number<vert_number:########
      fg=gh[vep][good_nbrs[0]][0]['forest']
      fb=gh[vep][bad_nbrs[0]][0]['forest']

      gh.remove_edge(vep,nbrs[0])
      gh.remove_edge(vep,nbrs[1])

      new_edges,edges_to_remove=get_new_edges(gh,vert_number,vert)

      gh.remove_edges_from(edges_to_remove)

      for e in new_edges:
        gh.add_edge(e[0],e[1],forest=e[2])

      gha=(bad_nbrs[0],vert)
      ghb=('a'+str(vert_number+1),good_nbrs[0])
      gh.add_edge(*gha,forest=fb)
      gh.add_edge(*ghb,forest=fg)

  if gh.number_of_edges(vep,nbrs[0])==2:
    gh.remove_edge(vep,nbrs[0])

    new_edges,edges_to_remove=get_new_edges(gh,vert_number,vert)

    gh.remove_edges_from(edges_to_remove)
```

```
    for e in new_edges:
      gh.add_edge(e[0],e[1],forest=e[2])

    gha=(vert,nbrs[0])
    ghb=('a'+str(vert_number+1),nbrs[0])

    gh.add_edge(*gha,forest='n')
    gh.add_edge(*ghb,forest='y')

  if gx.number_of_edges(vep,nbrs[0])!=2:
    if int(bad_nbrs[0][1:])>vert_number:
      fg=gx[vep][good_nbrs[0]][0]['forest']
      fb=gx[vep][bad_nbrs[0]][0]['forest']

      gx.remove_edge(vep,nbrs[0])
      gx.remove_edge(vep,nbrs[1])

      new_edges,edges_to_remove=get_new_edges(gx,vert_number,vert)

      gx.remove_edges_from(edges_to_remove)

      for e in new_edges:
        gx.add_edge(e[0],e[1],forest=e[2])

      gxa=(vert,'a'+str(bad_nbr_number+1))
      gxb=('a'+str(vert_number+1),good_nbrs[0])
      gx.add_edge(*gxa,forest=fb)
      gx.add_edge(*gxb,forest=fg)

    if int(bad_nbrs[0][1:])<vert_number:
      fg=gx[vep][good_nbrs[0]][0]['forest']
      fb=gx[vep][bad_nbrs[0]][0]['forest']

      gx.remove_edge(vep,nbrs[0])
      gx.remove_edge(vep,nbrs[1])

      new_edges,edges_to_remove=get_new_edges(gx,vert_number,vert)

      gx.remove_edges_from(edges_to_remove)

      for e in new_edges:
        gx.add_edge(e[0],e[1],forest=e[2])

      gxa=(bad_nbrs[0],'a'+str(vert_number+1))
      gxb=(vert,good_nbrs[0])
      gx.add_edge(*gxa,forest=fb)
      gx.add_edge(*gxb,forest=fg)

  if gx.number_of_edges(vep,nbrs[0])==2:
    gx.remove_edge(vep,nbrs[0])

    new_edges,edges_to_remove=get_new_edges(gx,vert_number,vert)
```

```
    gx.remove_edges_from(edges_to_remove)

    for e in new_edges:
      gx.add_edge(e[0],e[1],forest=e[2])

    gxa=(vert,nbrs[0])
    gxb=('a'+str(vert_number+1),nbrs[0])
    gx.add_edge(*gxa,forest='y')
    gx.add_edge(*gxb,forest='n')

  return gh,gx

def len_two(G,vert,vep,bad_nbrs,good_nbrs,vert_number):
  #note there is ambiguity between H and X if vep has a forested and non
      -forested adj edge
  # to see this, consider the isolated chord on a straight three sprout.
      depending on
  # what side you put the bubble, H and X can differ (ref sheet #8)

  gh=deepcopy(G)
  gx=deepcopy(G)
  nbrs=[]

  if bad_nbrs!=[]:
    bad_nbr_number=int(bad_nbrs[0][1:])

  for nbr in G.neighbors(vep):
    nbrs.append(nbr)

  nbrs.remove(vert)

  n0=min(nbrs)
  n1=max(nbrs)


  vep_n0_forest=gh[vep][n0][0]['forest']
  vep_n1_forest=gh[vep][n1][0]['forest']

  #fix n0 so that it is always the ep of the forest edge if in the case
      of a ft/nft pair for
  # the case when both endpoints lie n sprouting tree (see ref #8)

  if vep_n0_forest == 'y' and vep_n1_forest=='n':
    pass

  if vep_n0_forest == 'n' and vep_n1_forest=='y':
    n0,n1=n1,n0
    vep_n0_forest,vep_n1_forest=vep_n1_forest,vep_n0_forest

  vertplusone='a'+str(vert_number+1)

  gh.remove_edge(vert,vep)
```

```
    gx.remove_edge(vert,vep)

    new_edges,edges_to_remove=get_new_edges(gh,vert_number,vert)

    gh.remove_edges_from(edges_to_remove)
    gx.remove_edges_from(edges_to_remove)

    for e in new_edges:
      gh.add_edge(e[0],e[1],forest=e[2])
      gx.add_edge(e[0],e[1],forest=e[2])

    gh.remove_edge(vep,n0)
    gh.remove_edge(vep,n1)
    gx.remove_edge(vep,n0)
    gx.remove_edge(vep,n1)

    gh.add_edge(vert,n0,forest=vep_n0_forest)
    gh.add_edge(vertplusone,n1,forest=vep_n1_forest)
    gx.add_edge(vert,n1,forest=vep_n1_forest)
    gx.add_edge(vertplusone,n0,forest=vep_n0_forest)

    return gh,gx

def renumber_tree_vertices_gh(gh):
    # grab all vertices in sprouting tree(s)
    # rebuild graph from vertices and count the number of components
    # (i.e., get number of sprouting trees)
    # If a vertex is of type int (i.e., not part of linear forest),
    # then add to special list.  Sort list and get its length.
    # Set up a bijection between elements of list and the set {1, ... ,
        len(list)}

    vert_to_change=[]
    old_graph=[]

    for edge in gh.edges(data=True):
      old_graph.append([edge[0],edge[1],edge[2]['forest']])

      if type(edge[0]) is int and edge[0] not in vert_to_change:
        vert_to_change.append(edge[0])
      if type(edge[1]) is int and edge[1] not in vert_to_change:
        vert_to_change.append(edge[1])

    lvtc=len(vert_to_change)
    new_verts=range(1,lvtc+1)

    pairs=[(x,y) for x,y in zip(vert_to_change,new_verts)]
    pairs_dict={}

    for pair in pairs:
      pairs_dict[pair[0]]=pair[1]

    for edge in old_graph:
```

```
    if edge[0] in vert_to_change:
      edge[0]=pairs_dict[edge[0]]
    if edge[1] in vert_to_change:
      edge[1]=pairs_dict[edge[1]]

  new_graph=MultiGraph() #Build graph for sprout comp edge count

  for e in old_graph:
    new_graph.add_edge(e[0],e[1],forest=e[2])

  return new_graph

def renumber_tree_vertices_gx(gx):
  vert_to_change=[]
  old_graph=[]

  for edge in gx.edges(data=True):
    old_graph.append([edge[0],edge[1],edge[2]['forest']])

    if type(edge[0]) is int and edge[0] not in vert_to_change:
      vert_to_change.append(edge[0])

    if type(edge[1]) is int and edge[1] not in vert_to_change:
      vert_to_change.append(edge[1])

  lvtc=len(vert_to_change)
  new_verts=range(1,lvtc+1)

  pairs=[(x,y) for x,y in zip(vert_to_change,new_verts)]
  pairs_dict={}

  for pair in pairs:
    pairs_dict[pair[0]]=pair[1]

  for edge in old_graph:
    if edge[0] in vert_to_change:
      edge[0]=pairs_dict[edge[0]]

    if edge[1] in vert_to_change:
      edge[1]=pairs_dict[edge[1]]

  new_graph=MultiGraph()          #Build graph for sprout comp edge
      count

  for e in old_graph:
    new_graph.add_edge(e[0],e[1],forest=e[2])

  return new_graph

def forest_left(G):
  y_counter=0
  G=G.g
```

```
    for edge in G.edges_iter(data=True):
      if edge[2]['forest']=='y':
        return True

    if y_counter==0:
      return False

def process_end_result(graph_list):
  #function to take final graphs and then make them "display" correctly
  g_almost_done=[]
  g_proper=[]

  for entry in graph_list:
    new_g=[]

    for edge in entry.g.edges():
      v1new=int(edge[0][1:])+1  #strip the number off vertex and add
          offset of 1
      v2new=int(edge[1][1:])+1
      new_g.append(min(v1new,v2new))
      new_g.append(max(v1new,v2new))

    entry.g=new_g
    g_almost_done.append(entry)

  used_verts=g_almost_done[0].g[:]
  used_verts=sorted(used_verts)
  good_vals=range(1,len(used_verts)+1)

  bij={}
  i=0

  for entry in used_verts:
    bij[entry]=good_vals[i]
    i=i+1

  for graph in g_almost_done:
    edit=graph.g
    new_g=[]

    for vert in edit:
      new_g.append(bij[vert])

    graph.g=new_g

  for graph in g_almost_done:
    new_graph=[]

    for i in range(0,2*case,2):
      new_graph.append((graph.g[i],graph.g[i+1]))

    graph.g=new_graph
```

```
    g_proper=[]

    for graph in g_almost_done:
      if are_there_isolated_long_tuple(graph.g,case) is False:
        g_proper.append(graph)

    for graph in g_proper:
      graph.g=sorted(graph.g)

    return g_proper

def end_sort(diag):
    # sorts diagram so the edges are "increasing"
    case=len(diag)/2

    for i in xrange(0,len(diag)-2,2):
      if diag[i]>diag[i+2]:
        s0=diag[i]
        s1=diag[i+1]
        s2=diag[i+2]
        s3=diag[i+3]
        diag[i]=s2
        diag[i+1]=s3
        diag[i+2]=s0
        diag[i+3]=s1

    return diag



def program_core(ft,nft,sign):
    g0=lists_to_networkx_graph(ft,nft)

    g_done=[]
    g_left=[]
    g_final=[]

    gdapp=g_done.append

    gc=Graph(g0,sign)

    for entry in ihx(gc):
      forest_left_result=forest_left(entry)

      if forest_left_result is True:
        g_left.append(entry)
      else:
        gdapp(entry)

    # while loop to process sprouts
    # terminates when there are no sprouts left

    while g_left!=[]:
```

```
    out=[]

    for entry in g_left:
      for output in ihx(entry):
        out.append(output)

    g_left=[]

    for entry in out:
      forest_left_result=forest_left(entry)

      if forest_left_result is True:
        g_left.append(entry)
      else:
        gdapp(entry)

  g_final=process_end_result(g_done)
  return g_final

def fix_ft_nft(ft,nft):
  str_verts=[]
  print "."
  num_chords= (len(ft)+len(nft)-2)/2
  good_range=range(1,num_chords)

  for e in nft:
    if type(e) is str:
      str_verts.append(int(e[1:]))

  for e in ft:
    if type(e) is str:
      str_verts.append(int(e[1:]))

  return ft,nft


##########################################################################
#########################___Test_Graphs___################################
##########################################################################
##########################################################################

#ft=["a1",1]                    # single sprout in middle
#nft=["a0",1,"a2",1]

#ft=["a1",1,1,2]                # two-sprout with non-forest going to right
    of sprout
#nft=["a0",2,"a2",1,"a3",2]

#ft=["a2",1,1,2]                # two-sprout with non-forest going to left
    of sprout
#nft=["a0",2,"a1",1,"a3",2]

#ft=["a1",1,1,2,1,3]            # Y-sprout in center
```

```
#nft=["a0",2,2,3,'a2',3]

#ft=["a3",1,1,2,1,3]     # sign ref sheet #5
#nft=["a0",2,"a1",2,"a2",3,"a4",3]

#ft=["a1",1,1,2,1,3,"a4",4]     # Y-sprout, spare edge to right --  to
    test orien
#nft=["a0",2,2,3,'a2',3,"a3",4,"a5",4]

#ft=["a1",1,1,2,1,3]       #  ref sheet #6
#nft=["a0",2,"a2",2,"a3",3,"a4",3]

#ft=["a2",1,1,2,1,3]         #  ref sheet #7
#nft=["a0",2,"a1",2,"a3",3,"a4",3]

#ft=["a1",1,1,2,2,3,2,4,4,5]       # ref #8
#nft=["a0",3,1,4,3,5,"a2",5]

#ft=["a1",1,1,2,2,3,3,4]  #bisected bubble on side of two-sprout
#nft=["a0",4,1,3,"a2",2,"a3",4]

#########################################################################
#########################################################################
#########################################################################
#########################################################################

num_chords=4
global case
case =4


num=int(sys.argv[1])

if num ==1:
  for i in range(1,9999):

    if i%1000==0: #command console trick to check progress
      print i

    try:
      pr=open("./include/4/mor3/7mor3_"+str(i),"rb")
    except:
      print "End of all files..."
      break

    diags=pickle.load(pr)
    pr.close()
    pre_write=[]
    to_write=[1]

    for term in diags:
      ft=term[0]
      nft=term[1]
```

```
        sign=term[2]
        final_result=program_core(ft,nft,sign)
        pre_write.extend(final_result)

      #to_write=filter_small_chords(pre_write,case)
      to_write=pre_write

      if to_write != []:
        pw=open("./include/4/mor3relns/mor3_reln_"+str(i),"wb")
        pickle.dump(to_write,pw)
        pw.close()

if num !=1:
  for i in range((num-1)*10000,num*10000-1):

      if i%1000==0:
        print i

      try:
        pr=open("./include/4/mor3/7mor3_"+str(i),"rb")
      except:
        print "End of all files ... (second if)"
        break

      diags=pickle.load(pr)
      pr.close()
      pre_write=[]
      to_write=[1]

      for term in diags:
        ft=term[0]
        nft=term[1]
        sign=term[2]
        final_result=program_core(ft,nft,sign)
        pre_write.extend(final_result)

      #to_write=filter_small_chords(pre_write,case)
      to_write=pre_write

      if to_write != []:
        pw=open("./include/4/mor3relns/mor3_reln_"+str(i),"wb")
        pickle.dump(to_write,pw)
        pw.close()
```

A.6 good_diag_enum.py

```
#!/usr/bin/env python
'''
Created by Jonathan Gray .
Copyright (c)  Jonathan Gray. All rights reserved.
'''

import pickle

def make_dict(num_chords):
  f="genum"+str(num_chords)
  g_enum=open(f, 'r')
  graph_dict={}

  for graph in g_enum:
    graph=graph.split('|')
    graph_dict[graph[0]]=int(graph[1])

  g_enum.close()

  return graph_dict

def main():
  pass

if __name__ == '__main__':
  main()

my_dict= make_dict(6)

po=open("g6dict",'wb')
pickle.dump(my_dict,po)
po.close()
```

A.7 bad_to_good.py

```
#!/usr/bin/env python
# encoding: utf-8

'''
Created by Jonathan Gray .
Copyright (c)  Jonathan Gray. All rights reserved.


Takes a diagram and rewrites it as a sum of good chord diagrams and then
    reduces the linear combination that results.
term1 = implicitly original diagram
term2 = cross inner edges
term3 = init vert of second chord is pulled to left of init vert of
   first chord
term4 = permute the two operating verts from term3
term5 = term vert of first chord is pulled to right of term vert of
   second chord
term6 = permute two operating verts from term5

'''

from collections import deque
import pickle
import sys
import os
import glob

class Graph:
  "class to define a graph object with sign from IHX and forest
     relabelling"
  def __init__(self,graph,p_m_one):
    self.g=graph
    self.orien=p_m_one

class Diagram:
  def __init__ (self,g,orien,chords):
    self.g=g
    self.orien=orien
    self.c=self.get_chords()

  def get_chords(self):
    case=len(self.g)
    for c1 in self.g:
      for c2 in self.g:
        if (c2[0]==c1[1]+1) and (c2[1]>case):
          return c1,c2 #means diagram is bad
    return 0 #means diagram is good

def sixt(D):
  edges=D.g
  output=deque()
```

```
    outappend=output.append

    c1=D.c[0]
    c2=D.c[1]

    a=c1[0]
    b=c1[1]
    b1=c2[0]
    c=c2[1]

#####  create terms
    D2=[]
    D3=[]
    D4=[]
    D5=[]
    D6=[]

    D2ae=D2.append
    D3ae=D3.append
    D4ae=D4.append
    D5ae=D5.append
    D6ae=D6.append

    for edge in edges:
      e0=edge[0]
      e1=edge[1]
      if (edge != c1) and (edge !=c2):
        D2ae(edge)


        # i.e., initial vertex of edge is between a and b
        v0_bw_ab=(a <= e0 <= b)
        v1_bw_ab=(a <= e1 <= b)

        # i.e., initial vertex of edge is between a and c
        v0_bw_ac=(b <= e0 <= c)
        v1_bw_ac=(b <= e1 <= c)

        if v0_bw_ab and v1_bw_ab:
          D3ae((e0+1,e1+1))
          D4ae((e0+1,e1+1))

        if v0_bw_ab and not v1_bw_ab:
          D3ae((e0+1,e1))
          D4ae((e0+1,e1))

        if not v0_bw_ab and v1_bw_ab:
          D3ae((e0,e1+1))
          D4ae((e0,e1+1))

        if not v0_bw_ab and not v1_bw_ab:
          D3ae(edge)
          D4ae(edge)
```

```
        if  v0_bw_ac and  v1_bw_ac:
          D5ae((e0-1,e1-1))
          D6ae((e0-1,e1-1))

        if  v0_bw_ac and not v1_bw_ac:
          D5ae((e0-1,e1))
          D6ae((e0-1,e1))

        if  not v0_bw_ac and  v1_bw_ac:
          D5ae((e0,e1-1))
          D6ae((e0,e1-1))

        if  not v0_bw_ac and not v1_bw_ac:
          D5ae(edge)
          D6ae(edge)

    D2ae((c1[0],b1))
    D2ae((b,c2[1]))

    D3ae((a,c))
    D3ae((a+1,b1))

    D4ae((a+1,c))
    D4ae((a,b1))

    D5ae((a,c))
    D5ae((b,c-1))

    D6ae((a,c-1))
    D6ae((b,c))

    if are_there_isolated_long(D2) is False:
      outappend(Diagram(D2,D.orien*(-1)**(0)*-1,0))

    if are_there_isolated_long(D3) is False:
      outappend(Diagram(D3,D.orien*(-1)**(a+b)*-1,0))

    if are_there_isolated_long(D4) is False:
      outappend(Diagram(D4,D.orien*(-1)**(a+b)*-1,0))

    if are_there_isolated_long(D5) is False:
      outappend(Diagram(D5,D.orien*(-1)**(c+b+1)*-1,0))

    if are_there_isolated_long(D6) is False:
      outappend(Diagram(D6,D.orien*(-1)**(c+b+1)*-1,0))

    return output

def get_dict(num_chords):
    pickle_file="./include/g"+str(num_chords)+"dict"
    po=open(pickle_file,'rb')
    mydict=pickle.load(po)
```

```
  po.close()
  return mydict

def are_there_isolated_long(graph):
  for edge in graph:
    v0=edge[0]
    v1=edge[1]
    if abs(v0-v1)==1 or abs(v0-v1)==(2*case-1) or abs(v0-v1)==(2*case-2)
        :
      return True
  return False

def program_core(D):
  badarray=[]
  goodarray=deque()

  for entry in sixt(D):
    result=entry.c
    if result == 0:
      goodarray.append(entry)
    else:
      badarray.append(entry)

  while(badarray!=[]):
    out=[]
    tempbad=[]
    outapp=out.append
    for entry in badarray:

      for output in sixt(entry):
        result=output.c
        if result == 0:
          goodarray.append(output)
        else:
          tempbad.append(output)
    badarray=tempbad

  return goodarray

def file_process(case,file_number):

  ## = rank case
  # * = r [morita] or n [normal]
  # fn1=./include/#/mo*#relns/mo*#_reln_
  # fn2=./include/#/mo*#rows/r_

  fo=open("fn1"+file_number,'rb')
  fw=open("fn2"+file_number,'wb')

  relns=pickle.load(fo)

  fo.close()
```

```
    my_dict=get_dict(case)

    goodarray=[]

    for reln in relns:
      diag=Diagram(reln.g,reln.orien,0)
      if diag.c!=0:
        goodarray=(program_core(diag))
      if diag.c==0:
        goodarray.append(diag)

    row=[0]*(6*5*4*3*2)

    #geplwor = graph endpoint labels with orientation

    for entry in goodarray:
      geplwor=[]
      entry.g=sorted(entry.g)
      for edge in entry.g:
        geplwor.append(edge[1])
      geplwor.append(entry.orien)
      row[my_dict[str(geplwor[0:-1])]]=row[my_dict[str(geplwor[0:-1])]]+
          geplwor[-1]

    for element in row:
      fw.write("%s " % element)
    fw.write("\n")
    fw.close()
    print "end -->",file_number

global case
case=6

file=open('./include/fg_'+str(case),r)

for line in file:
  line=line.split()
  print line
```

A.8 matrix.py

```
#!/usr/bin/env python
# encoding: utf-8
"""
Created by Jonathan Gray .
Copyright (c) Jonathan Gray. All rights reserved.
"""

import sys
import os
import cPickle

mat=[]

old=open("mon4rows",'r')

mat=[]

for row in old:
  mat.append(row)

old.close()
new=open("./include/7/mon4rows_split/rows_1",'w')

for i in xrange(1,17496+1):
  if i%500==0:
    new.close()
    fn=str(i/500+1)
    new=open("./include/7/mon4rows_split/rows_"+fn,'w')
  new.write("%s" % mat[i-1])

new.close()
```

A.9 cat_rows.py

```
#!/usr/bin/env python
# encoding: utf-8
"""
Created by Jonathan Gray .
Copyright (c)  Jonathan Gray. All rights reserved.
"""

import sys
import os
import glob

def main():
  pass


if __name__ == '__main__':
  main()


# grab a list of all files in directory
files_to_process=glob.glob('./include/7/mor4rows/*')


#create file to put rows in and then contatenate rows
os.system("touch s3")

cat="/bin/cat"
for entry in files_to_process:
  cmd=cat+" "+entry+" >> "+"mor4s3"
  os.system(cmd)
```

Vita

Jonathan Nathan Gray was born in Stuart, Florida. He spent his early years in south Florida and moved to the state of Tennessee in the summer of 1993. After graduating from high school in 2001, he started his undergraduate work in mathematics and physics at Florida State University and graduated in 2004. During fall 2004 to spring 2005, Jonathan taught high school mathematics and then left his position to begin doctoral studies at the University of Tennessee, Knoxville. He graduated in May 2011.