

COMPUTING THE GENERATING FUNCTION OF A  
COINVARIANTS MAP

BY

JESSE FROHLICH

A thesis submitted in conformity with  
the requirements for the degree of

Doctor of Philosophy

Graduate Department of Mathematics  
University of Toronto

© 2023 Jesse Frohlich

# ABSTRACT

---

Computing the generating function of a coinvariants map

Jesse Frohlich

Doctor of Philosophy

Graduate Department of Mathematics

University of Toronto

2023

A well-known source of strong link invariants comes from quantum groups. Typically, one uses a representation of a quantum group to build a computable invariant, though these computations require exponential time in the number of crossings. Recent work has allowed for direct and efficient computations within the quantum groups themselves, through the use of perturbed Gaußian differential operators. This thesis introduces and explores a partial expansion of the tangle invariant  $Z$  introduced by Bar-Natan and van der Veen [BNvdV]. We expand the use of the quantum group  $\mathcal{U}(\mathfrak{sl}_{2+}^0)$  to include its space of coinvariants, providing an extension  $Z^{\text{tr}}$  of  $Z$  from open tangles to links.

We compute a basis for the space of coinvariants, then compute a closed-form expression for the corresponding trace map in the form of an exponential generating function. The resulting function is not a compatible perturbed Gaußian with respect to the previous research. To respond to this limitation, we find a method of computing the link invariant for a subclass of links and write a program to compute the invariant  $Z^{\text{tr}}$  on two-component links. Contrary to expectations, we find that  $Z^{\text{tr}}$  is neither stronger nor weaker than the Multivariable Alexander polynomial. This unexplained behaviour

warrants further study into the invariant  $Z^{\text{tr}}$  and its relationship with other invariants.

*To someone,  
who did something nice.*

# ACKNOWLEDGEMENTS

---

Thank you to my advisor, Dror Bar-Natan, whose consistent meetings, guidance, and encouragement brought me to this point.

Thank you to my supervisory committee, Joel Kamnitzer and Eckhard Meinrenken, who helped connect me to more of the department and its research.

Thank you to to Jason Siefken and the other teaching mentors in the math department. You have made me a better educator and communicator.

Thank you to the math department, especially Jemima Merisca and Sonja Injac, whose friendly support through the manifold administrative details was invaluable.

Thank you to the people I met in the department, especially those who opened my thesis to confirm they were included here. Thank you specifically.

Thank you to Assaf Bar-Natan, Vincent Girard, Caleb Jonker, and Adriano Pacifico, whose friendship, bike trips, and cooking sessions provided much joy.

Thank you to the care-taking staff at the St. George campus, especially Maria and Maria, whose diligent work and kind words have made this campus beautiful.

For the Graduate Christian Fellowship community at the U of T: thank you for being a refreshing breadth of perspectives for a graduate student.

Thank you to Rosemary and Alan Johnstone, whose hospitality made me feel immediately welcome in a new city.

My family, who raised me and encouraged me in my academic pursuits from the earliest moments.

Thank you to my mom for embracing my atypical learning styles, for instilling my love of reading, and for teaching me how to think deeply.

Thank you to my dad for teaching me to ride a bike, cook, file taxes, and notice the invisible among us.

Thank you to my brother for all the laughter you bring to my life.

Thank you to the Perrins. You are my second family, and your support and love for me is precious.

Thank you to my fiancée, Emily Langridge, who helped me with the motivation to finish. I look forward to the next chapters of my life with you.

# ACRONYMS

---

**RVT** Rotational Virtual Tangle

**RVK** Rotational Virtual Knot

**RVL** Rotational Virtual Link

**MVA** Multivariable Alexander polynomial

# CONTENTS

---

1	Executive Summary	9
1.1	Context . . . . .	9
1.2	Extending the invariant to more knotted objects . . . . .	11
2	Tensor Products and Meta-Objects	14
2.1	Tensor product notation . . . . .	14
2.2	Meta-objects . . . . .	16
2.3	Algebraic definitions . . . . .	19
2.4	The meta-algebra of tangle diagrams . . . . .	23
2.5	The <i>ybox</i> meta-algebra . . . . .	32
2.6	Morphisms between meta-objects . . . . .	33
3	Perturbed Gaussians	36
3.1	Expressing morphisms as generating functions . . . . .	36
4	Constructing the Trace	42
4.1	Extending a pure tangle invariant to links and general tangles	42
4.2	The coinvariants of $U$ . . . . .	43
5	Conclusions	49
5.1	Comparison with the multivariable Alexander polynomial .	49
5.2	Further work . . . . .	50
A	Code	51
A.1	Implementation of the invariant $Z$ . . . . .	51
A.2	Implementation of the trace . . . . .	62
A.3	Implementation of rotation number algorithm . . . . .	73



# EXECUTIVE SUMMARY

---

## 1.1 CONTEXT

### *Understanding Knotted Objects*

In the field of knot theory, distinguishing between two knots or links has proven to be a difficult task. Computing strong invariants of knotted objects is a popular way to aid with the classification of these objects.

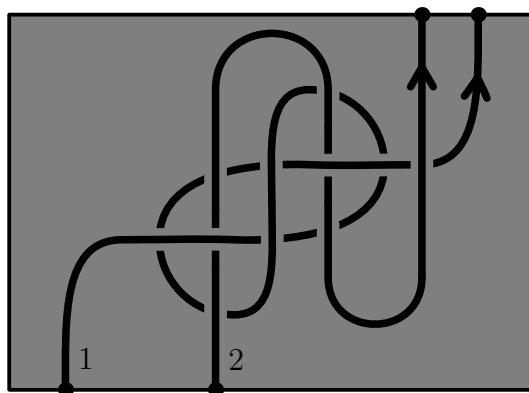


Figure 1.1: Two strings knotted together. Can they be disentangled?

Merely being able to distinguish between two knotted objects does not always provide us with enough information about these topological structures. For instance, one may ask if a particular link is a satellite of another (roughly: where one knot is embedded into a link by following one of its components), whether a knot is slice (i.e. it is the boundary of a disk in  $\mathbb{R}^4$ ), or whether it is ribbon (i.e. the boundary of a disk in  $\mathbb{R}^3$  with restricted types of singularities). Many interesting properties of knots can be phrased in terms of certain topological properties, such as strand doubling (taking a strand and replacing it with two copies of itself, as in figure 1.2) or strand stitching (joining two open components together to form one longer one).

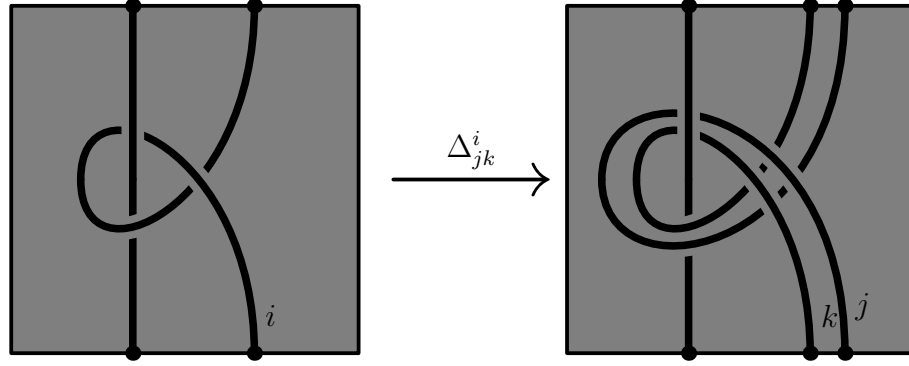


Figure 1.2: An example of strand-doubling.

Open problems such as the Ribbon Slice Conjecture (asking whether there exists a slice knot which is not ribbon) may be advanced by the development of “topologically aware” invariants—those which preserve topological data in a retrievable way.

### *Quantum Invariants*

One such class of topological invariants is derived from quantum groups, which are algebraic structures whose relations mimic those of knotted objects. With this approach, one takes a knotted object and decomposes it into a series of topological operations (such as stitching strands or doubling strands), then maps each of these operations to a corresponding algebraic operation. The composition of these algebraic operations is the value of the invariant.

More specifically, a quantum group (also called a Hopf algebra) is a vector space  $A$  together with several maps between various tensor powers (for instance, a multiplication map  $m: A \otimes A \rightarrow A$ ). The value of the invariant is a vector in some tensor power of  $A$ , with one factor assigned to each strand of the knotted object. It is computed by associating to each concatenation of strands with a multiplication of algebra elements. While this formulation is elegant, it has a notable drawback: computing the value of an invariant with many components requires manipulating large tensor powers of  $A$ . One remedy is to instead perform the computation in a representation of  $A$ , say a small vector space  $V$ , though the issue of exponential growth in complexity remains.

### *Images of the invariant*

To avoid the issue of exponential computational complexity, one can instead investigate the nature of the set of all values of the invariant as a subset of the algebra and its tensor powers. For a particular choice of algebra (namely  $\hat{\mathfrak{U}}(\mathfrak{sl}_{2+}^0)$ , as investigated by Dror Bar-Natan and Roland van der Veen in [BNvdV]) the space of values the corresponding invariant  $Z$  can take is significantly smaller than the whole space; the dimension of the space of values grows only quadratically with the number of crossings in the knotted object. In particular, by looking at the generating functions of the associated maps, instead of a generic power series, the value of  $Z$  on tangles always take the form of a (perturbed) Gaussian. Computationally, this means one needs only to keep track of the quadratic form and the perturbation. The invariant  $Z$ —which dominates the  $\mathfrak{sl}_2$ -coloured Jones polynomial—provides an efficient method of computing the Alexander polynomial.

### 1.2 EXTENDING THE INVARIANT TO MORE KNOTTED OBJECTS

The research program outlined by Bar-Natan and van der Veen computes  $Z$  only for (pure) tangles—that is, collections of open strands whose endpoints are fixed to a boundary circle. (Note that this includes long knots, which are exactly the one-component tangles.) This thesis is focused on extending  $Z$  and its computations to tangles with closed components.

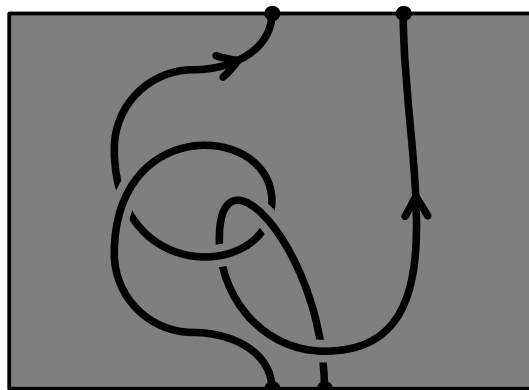


Figure 1.3: A pure tangle. All components intersect the boundary.

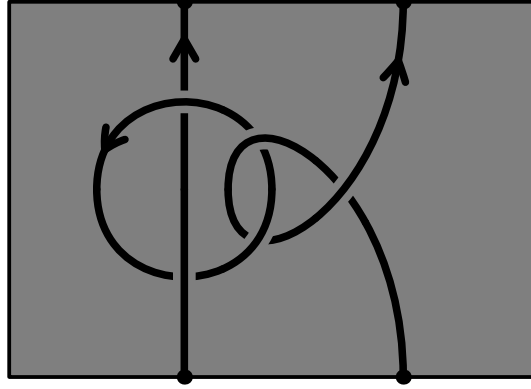


Figure 1.4: A tangle with a closed component.

### *Computing the extended map*

The first task is to determine the space in which the extended invariant, which we will call  $Z^{\text{tr}}$ , lives. One may observe that in a matrix algebra, one is able to contract two matrices together via matrix multiplication. When one wishes to contract a matrix along itself, one uses the trace map. Analogously, since two strands in a tangle corresponds to multiplication, closing a strand into a loop should correspond algebraically to a trace map.

In a generic algebra  $A$ , the trace map is defined as the projection onto the set of coinvariants:  $\text{tr}: A \rightarrow A_A = A/[A, A]$ . In order to extend the invariant in this framework, we must first compute the space of coinvariants for  $\hat{\mathcal{U}}(\mathfrak{sl}_{2+}^0)$ , then compute the coinvariants map, and write it as a generating function. (This is accomplished in section 4.2.)

### *Performing computations*

Unfortunately, the resulting trace map does not take the form of a perturbed Gaußian in a way that readily connects to the existing framework. In order to determine whether further study in this direction is merited, we must find an alternative computation method to get a preliminary sense of the strength of  $Z^{\text{tr}}$ .

For a subclass of links (which includes all two-component links), we compute an explicit closed form for the trace map, then implement a computer program to compute the value of  $Z^{\text{tr}}$  on all two-component links with up to 11 crossings. When applied to knots,  $Z$  computes the Alexander

polynomial. When applied to two-component tangles, one may expect that  $Z^{\text{tr}}$  would produce the natural generalization to multiple components: the Multivariable Alexander polynomial (MVA). Surprisingly, the MVA and  $Z^{\text{tr}}$  are incomparable, with each being able to distinguish pairs of links the other cannot. (See section 5.1 for more information.)

#### *Further study*

As  $Z^{\text{tr}}$  does not generalize  $Z$  in the manner expected, several interesting avenues of further research are opened. Firstly, determining what the relationship between  $Z^{\text{tr}}$  and the MVA is remains open. Second is the challenge of finding an efficient method for computing  $Z^{\text{tr}}$  on links with more than two components, which currently is mired in complications with the presence of non-elementary functions where quadratic forms normally appear. Third is the question of the existence of other viable trace candidates. In particular, it may be worth exploring whether a universal trace with respect to the perturbed Gaussian framework defines a sufficiently useful invariant.

# TENSOR PRODUCTS AND META-OBJECTS

---

## 2.1 TENSOR PRODUCT NOTATION

In what follows, we will extensively use tensor products, tensor powers, and generalizations thereof. We begin by introducing the notation that will be used first for traditional tensor products, then for their generalizations.

Let  $V$  be a  $\mathbb{k}$ -vector space (for the moment assumed to be finite dimensional). When working with a large tensor power  $V^{\otimes n}$  of  $V$ , it will often be more convenient to label tensor factors with elements of a finite set  $S$  (with  $|S| = n$ ) rather than by their position in a linear order.

For example, consider the vector  $u \otimes v \otimes w \in V^{\otimes 3}$ . Let us choose an index set  $S = \{1, 2, 3\}$ . We then may equivalently write this vector by labelling each tensor factor with one of the elements of  $S$ , say  $u_1 v_2 w_3$ . Since the labels serve to distinguish the separate factors, this vector may equivalently be written as  $u_1 v_2 w_3 = v_2 u_1 w_3 = w_3 u_2 u_1 \in V^{\otimes S}$ . We will write the set  $V^{\otimes S}$  with a subscript:  $V_S$ . We formalize the idea below:

**Definition 2.1** (indexed tensor powers). Let  $V$  be a vector space and  $S = \{s_1, \dots, s_n\}$  be a finite set. We define the indexed tensor power of  $V$  to be the collection of formal linear combinations of functions from  $S$  to  $V$

$$V_S := \text{span}\{f: S \rightarrow V\} / \sim \quad (2.1)$$

subject to the standard multilinear relations, namely multi-additivity and the factoring of scalars.

By multi-additivity, we mean that for each  $i \in S$  and  $f, g \in V_S$  satisfying  $f(i) = g(i) = v$ , we have:

$$f + g = \left( s \mapsto \begin{cases} v & \text{if } s = i \\ f(i) + g(i) & \text{otherwise} \end{cases} \right) \quad (2.2)$$

In practice, we will write such functions  $f: S \rightarrow V$  with  $f(s_i) = v_i$  in the following way:

$$(v_1)_{s_1} (v_2)_{s_2} \cdots (v_n)_{s_n} := f \quad (2.3)$$

With this notation, we may easily express the factoring of scalars as:

$$(v_1)_{s_1} (v_2)_{s_2} \cdots (\lambda v_i)_{s_i} \cdots (v_n)_{s_n} = \lambda \cdot (v_1)_{s_1} (v_2)_{s_2} \cdots (v_n)_{s_n} \quad (2.4)$$

Next, we introduce notation for maps between tensor powers so that we may unambiguously refer to appropriate tensor factors while defining morphisms. Let  $D$  and  $C$  be finite sets, and  $T: V_D \rightarrow V_C$ . We will denote  $T$  by  $T_C^D$ . It is important to note that when  $T$  is not symmetric in its arguments, the order of the indices in this notation matters.

**Example 2.2.** Let  $V = \mathbb{R}^2$ , and  $T_c^{a,b}$  defined by

$$\begin{aligned} T_c^{a,b}(\vec{v}_a(\vec{e}_1)_b) &= \vec{0}_c \\ T_c^{a,b}(\vec{v}_a(\vec{e}_2)_b) &= \vec{v}_c \end{aligned} \quad (2.5)$$

This function zeros out vectors whose  $b$ -component is  $\vec{e}_1$ . If we wish to define an analogous function for the  $a$ -component, we may simply reverse the order of the superscript:  $T_c^{b,a}$ , which sends  $\vec{v}_b(\vec{e}_1)_a = (\vec{e}_1)_a \vec{v}_b$  to  $\vec{0}_c$  and  $\vec{v}_b(\vec{e}_2)_a$  to  $\vec{v}_c$ .

Finally, we point out that any morphism  $T_C^D$  may be extended to one with larger domain and codomain. We introduce the notation  $T_C^D[S] := T_C^D \otimes \text{id}_S^S$  for this concept, though will also overload the notation  $T_C^D$  for the same map: for any  $v_D \in V_D$  and  $w_S \in V_S$ , we may also write  $(T_C^D)(v_D \otimes w_S) := (T_C^D v_D) \otimes w_S$ .

*Remark 2.3.* There are three special cases with this notation:

- Given a (multi)linear functional  $\phi: V_S \rightarrow \mathbb{k} \cong V_\emptyset$ , we will write  $\phi^S$  instead of  $\phi_\emptyset^S$ . The linear order on  $S$  remains in this notation.

- Elements  $v \in V_S$  will be interpreted as a map  $v: \mathbb{k} = V_\emptyset \rightarrow V_S$  written  $v_S$  instead of  $v_S^\emptyset$ .
- When only one index is present in a subscript or superscript, and its omission does not introduce an ambiguity in an expression, then it may be omitted to improve readability. For instance, a map  $\phi: V_{\{1,2\}} \rightarrow V_{\{3\}}$  may be written as  $\phi^{1,2}$  instead of  $\phi_3^{1,2}$ , with the canonical isomorphism  $V \cong V_{\{3\}}$  being suppressed.

When taking the tensor product of two such tensor powers, we follow [BNS] and use the notation “ $\sqcup$ ” instead of “ $\otimes$ ”:

$$V_X \sqcup V_Y := V_{X \sqcup Y} \quad (2.6)$$

Additionally, given  $\phi_{C_1}^{D_1}$  and  $\psi_{C_2}^{D_2}$  such that  $D_1 \cap D_2 = \emptyset = C_1 \cap C_2$ , we have a product morphism  $\phi_{C_1}^{D_1} \psi_{C_2}^{D_2} := \phi \otimes \psi: V_{D_1 \sqcup D_2} \rightarrow V_{C_1 \sqcup C_2}$ , which we also write with concatenation.

## 2.2 META-OBJECTS

### *Notation extension beyond vector spaces*

While the above notation is helpful when working with vector spaces, we are interested in also using the same notation to describe a tangle. Our formulation of tangles (introduced in section 2.4) is neither a tensor product nor a monoidal category, though it shares many similarities with both concepts. In particular, the domains and codomains of the maps we have discussed so far have only depended on the index set. With this observation, we replace the notation of tensor powers with that of a so-called meta-object:

**Definition 2.4** (Meta-object). Let  $\mathcal{C}$  be a category. A meta-object in  $\mathcal{C}$  is subcategory with objects indexed by the functor

$$\begin{aligned} A: \mathbf{FinSet} &\rightarrow \mathbf{Ob}(\mathcal{C}) \\ S &\mapsto A_S \end{aligned} \quad (2.7)$$

The homsets of this subcategory are indexed by pairs of finite sets  $D, C$ . Morphisms in these homsets will be denoted by  $\phi_C^D: A_D \rightarrow A_C$ . For each



such morphism  $\phi_C^D$ , there is a finite-set-indexed morphism  $\phi_C^D[\cdot]: \mathbf{FinSet} \rightarrow \mathbf{Hom}(\mathcal{C})$  such that

1.  $\phi[S]: A_{C \sqcup S} \rightarrow A_{D \sqcup S}$
2.  $\phi[\emptyset] = \phi$
3.  $(\phi[S])[T] = \phi[S \sqcup T]$

Note that the functor generalizes the map  $S \mapsto V^{\otimes S}$ , while the morphisms generalize the extension-by-identity  $\phi \otimes \text{id}_S^S$ .

Composition of morphisms  $\phi_{C_1}^{D_1}$  and  $\psi_{C_2}^{D_2}$  is defined when  $C_1 = D_2$ , and is written with the following concatenation operator: <sup>†1</sup>

$$\phi_{C_1}^{D_1} \parallel \psi_{C_2}^{D_2} := \psi_{C_2}^{D_2} \circ \phi_{C_1}^{D_1}: \mathcal{C}_{D_1} \rightarrow \mathcal{C}_{C_2} \quad (2.8)$$

In the general case, we still have the map  $\sqcup$  defined by  $A_S \sqcup A_T = A_{S \sqcup T}$ . Given morphisms  $\phi_{C_1}^{D_1}$  and  $\psi_{C_2}^{D_2}$  such that  $D_1 \cap D_2 = \emptyset = C_1 \cap C_2$ , we have a product morphism  $\phi_{C_1}^{D_1} \psi_{C_2}^{D_2} := \phi \otimes \psi: \mathcal{C}_{D_1 \sqcup D_2} \rightarrow \mathcal{C}_{C_1 \sqcup C_2}$ , which we write with concatenation.

*Remark 2.5.* To make expressions easier to read, in this paper we will introduce the domain extension implicitly in the following context: given morphisms  $\phi_{C_1}^{D_1}$  and  $\psi_{C_2}^{D_2}$  such that  $D_2 \subseteq C_1$  and  $C_2 \cap (C_1 \setminus D_2) = \emptyset = D_1 \cap (D_2 \setminus C_1)$ , we define:

$$\phi_{C_1}^{D_1} \parallel \psi_{C_2}^{D_2} := \phi_{C_1}^{D_1}[D_2 \setminus C_1] \parallel \psi_{C_2}^{D_2}[C_1 \setminus D_2] \quad (2.9)$$

The two extreme cases of this definition are:

- When  $C_1 \cap D_2 = \emptyset$ , equation (2.9) becomes  $\phi_{C_1}^{D_1} \psi_{C_2}^{D_2}$ .
- When  $C_1 = D_2$ , equation (2.9) becomes the composition  $\phi_{C_1}^{D_1} \parallel \psi_{C_2}^{D_2}$  exactly.

*Remark 2.6.* While the  $\parallel$  operator is associative, care must be taken that the compositions are well-defined in the presence of duplicated indices. While it is sufficient for all the finite sets in a composition to be pairwise disjoint, this condition will prove too restrictive for clear communication of formulae.

<sup>†1</sup> We denote left-to-right composition with the “ $\parallel$ ” symbol:  $f \parallel g := g \circ f$ . Writing function composition in this order assists with readability when there are many functions to apply.

### Defining a meta-group

To make the above definition more concrete, we will go through the process of defining a meta-group, which is a generalization of a group object. Traditionally, the data of a group object are the following:

- An object  $G$  in a category  $\mathcal{C}$ .
- A morphism  $m: G \times G \rightarrow G$  called “multiplication”.
- A “unit” morphism  $\eta: \{1\} \rightarrow G$ .<sup>†2</sup>
- An “inversion” morphism  $S: G \rightarrow G$ .
- A collection of relations between the morphisms, written as equalities of morphisms between Cartesian powers of  $G$ . For example, associativity may be written:

$$\begin{array}{ccc} G \times G \times G & \xrightarrow{m \times \text{id}} & G \times G \\ \text{id} \times m \downarrow & & \downarrow m \\ G \times G & \xrightarrow{m} & G \end{array} \quad (2.10)$$

Further, the data of these relations is extended to higher powers of  $G$  by acting on other components by the identity:

$$\begin{array}{ccc} G^{n+3} & \xrightarrow{m \times \text{id}^{n+1}} & G^{n+2} \\ \text{id} \times m \times \text{id}^n \downarrow & & \downarrow m \times \text{id}^n \\ G^{n+2} & \xrightarrow{m \times \text{id}^n} & G^{n+1} \end{array} \quad (2.11)$$

Let us alter how we package these data so as to maximize the clarity of the meta-group structure:

1. Instead of linear orders of factors  $G \times \cdots \times G$ , we will index factors by a finite set  $X$ , writing it  $G_X := \{f: X \rightarrow G\}$  in the style of equation (2.1).
2. The indexed factors will determine how the group operations act. For instance, multiplication of factor  $i$  and  $j$  together, with the result labelled in factor  $k$  is to be written  $m_k^{ij}: G_{\{i,j\}} \rightarrow G_{\{k\}}$ .

<sup>†2</sup> When  $\mathcal{C} = \mathbf{Set}$ , we usually write the unit as an element  $1 = \eta(1) \in G$

3. Instead of implicitly including extensions of morphisms to higher powers by the identity, we will parametrize the extension by finite sets by  $\phi_C^D[X] := \phi_C^D \times \text{id}_X^X$ . For example, multiplication  $m_k^{ij}: G_{\{i,j\}} \rightarrow G_{\{k\}}$  generates a family of maps  $m_k^{ij}[X]: G_{\{i,j\} \sqcup X} \rightarrow G_{\{k\} \sqcup X}$ , each of which must satisfy the relations of the group object such as equation (2.11).

This way of packaging the data leads us to the following generalization:

**Definition 2.7.** A meta-group in  $\mathcal{C}$  is the following data:

- A family of objects  $G_X \in \mathcal{C}$ , indexed over finite sets  $X$ .
- A family of morphisms  $m_k^{ij}[X]: G_{\{i,j\} \sqcup X} \rightarrow G_{\{k\} \sqcup X}$  called “multiplication”.
- A family of “unit” morphisms  $\eta_i[X]: G_X \rightarrow G_{\{i\} \sqcup X}$ .
- A family of “inversion” morphisms  $S_j^i[X]: G_{\{i\} \sqcup X} \rightarrow G_{\{j\} \sqcup X}$ .
- A collection of relations between the morphisms, written as equalities of morphisms between the  $G_X$ ’s. For example, associativity may be written:

$$\begin{array}{ccc}
 G_{\{1,2,3\} \sqcup X} & \xrightarrow{m_1^{1,2}[X \sqcup \{3\}]} & G_{\{1,3\} \sqcup X} \\
 m_2^{2,3}[X \sqcup \{1\}] \downarrow & & \downarrow m_1^{1,3}[X] \\
 G_{\{1,2\} \sqcup X} & \xrightarrow{m_1^{1,2}[X]} & G_{\{1\} \sqcup X}
 \end{array} \quad (2.12)$$

### 2.3 ALGEBRAIC DEFINITIONS

We now introduce the algebraic structures which will be used to define the tangle invariant. These definitions follow those given by Majid in [Maj], although the ones presented below are given in a way that their corresponding meta-structure is readily visible.

**Definition 2.8** (meta-algebra). A meta-algebra (or meta-monoid) is a collection of objects  $\{A_X\}_X$  in  $\mathcal{C}$  together with an associative multiplication  $m_k^{i,j}: A_{\{i,j\}} \rightarrow A_{\{k\}}$  (satisfying equation (2.13)), and a unit  $\eta_i: A_\emptyset \rightarrow A_{\{i\}}$  satisfying equation (2.14).

*Remark 2.9.* When  $\mathcal{C} = \mathbf{Vect}$  and  $A_X = V^{\otimes X}$  for some vector space  $V$ , definition 2.8 becomes the more familiar definition of an algebra. When  $A_\emptyset$

is a field, it is more common think of the unit as an element  $\mathbf{1} \in V$ . The unit map is then defined by linearly extending the assignment  $\eta_i(1) = \mathbf{1}_i$ .

$$\begin{array}{ccc}
 A_{\{1,2,3\}} & \xrightarrow{m_1^{1,2}} & A_{\{1,3\}} \\
 m_2^{2,3} \downarrow & & \downarrow m_1^{1,3} \\
 A_{\{1,2\}} & \xrightarrow{m_1^{1,2}} & A_{\{1\}}
 \end{array} \quad (2.13)$$

$$\begin{array}{ccc}
 A_{\{1\}} & \xrightarrow{\eta_2} & A_{\{1,2\}} \\
 \text{id} \searrow & m_1^{2,1} \downarrow & \downarrow m_1^{1,2} \\
 & A_{\{1\}} &
 \end{array} \quad (2.14)$$

*Remark 2.10.* From now on, we will denote repeated multiplication as in equation (2.13) by using extra indices. For instance:  $m_\ell^{i,j,k} := m_r^{i,j} \parallel m_\ell^{r,k} = m_s^{j,k} \parallel m_\ell^{i,s}$ .

There is also the dual notion of a coalgebra, which arises by reversing the arrows in equations (2.13) and (2.14):

**Definition 2.11** (meta-coalgebra). A meta-colagebra (or meta-comonoid) is a collection  $\{C_X\}_X$  together with a comultiplication  $\Delta_{jk}^i: C_{\{i\}} \rightarrow C_{\{j,k\}}$  which is coassociative (equation (2.15)) and a counit, which is a map  $\epsilon^i: A_i \rightarrow A_\emptyset$  satisfying equation (2.16).

$$\begin{array}{ccc}
 C_{\{1,2,3\}} & \xleftarrow{\Delta_{1,2}^1} & C_{\{2,3\}} \\
 \Delta_{2,3}^2 \uparrow & & \uparrow \Delta_{1,3}^1 \\
 C_{\{1,2\}} & \xleftarrow{\Delta_{1,2}^1} & C_{\{1\}}
 \end{array} \quad (2.15)$$

$$\begin{array}{ccc}
 C_{\{1\}} & \xleftarrow{\epsilon^2} & C_{\{1,2\}} \\
 \text{id} \swarrow & \Delta_{1,2}^1 \uparrow & \uparrow \Delta_{2,1}^1 \\
 & C_{\{1\}} &
 \end{array} \quad (2.16)$$

*Remark 2.12.* From now on, we will denote repeated comultiplication as in equation (2.15) by using extra indices. For instance:  $\Delta_{j,k,\ell}^i := \Delta_{j,r}^i \parallel \Delta_{k,\ell}^r = \Delta_{s,\ell}^i \parallel \Delta_{j,j}^s$ .

If a meta-object  $\{B_X\}_x$  satisfies both definitions of an algebra and a coalgebra, we introduce a definition for when the structures are compatible with each other in the following way:

**Definition 2.13** (meta-bialgebra). A meta-bialgebra (or meta-bimonoid) is a meta-algebra  $(B, m, \eta)$  and a meta-coalgebra  $(B, \Delta, \epsilon)$ , such that  $\Delta$  and  $\epsilon$  are meta-algebra morphisms. <sup>†3</sup>

<sup>†3</sup>  $B_X$  inherits a (co)algebra structure from  $B$ , given by  $(B_X)_Y := B_{X^Y}$  and component-wise operations. The bialgebra structure on  $B_\emptyset$  is given by  $m = \eta = \Delta = \epsilon = \text{id}$ .

$$\begin{array}{ccc}
B_{\{1,2\}} & \xrightarrow{m_1^{1,2}} & B_{\{1\}} \\
\downarrow \Delta_{1,1,2}^1 // \Delta_{2,1,2}^2 & & \downarrow \Delta_{1,2}^1 \\
B_{\{1_1,1_2,2_1,2_2\}} & \xrightarrow{m_1^{1,1,2} // m_2^{2,1,2}} & B_{\{1,2\}}
\end{array} \quad (2.17)$$

$$\begin{array}{ccc}
& & B_{\{1\}} \\
& \nearrow \eta_1 & \downarrow \Delta_{1,2}^1 \\
B_\emptyset & & \\
& \searrow \eta_1 // \eta_2 & \\
& & B_{\{1,2\}}
\end{array} \quad (2.18)$$

$$\begin{array}{ccc}
B_{\{1,2\}} & \xrightarrow{m_1^{1,2}} & B_{\{1\}} \\
& \searrow \epsilon^1 // \epsilon^2 & \swarrow \epsilon^1 \\
& & B_\emptyset
\end{array} \quad (2.19)$$

$$\begin{array}{ccc}
B_\emptyset & \xrightarrow{\eta_1} & B_{\{1\}} \\
& \searrow \text{id} & \downarrow \epsilon^1 \\
& & B_\emptyset
\end{array} \quad (2.20)$$

*Remark 2.14.* The conditions for  $\Delta$  being an algebra morphism are presented in equations (2.17) and (2.18), while those for  $\epsilon$  are in equations (2.19) and (2.20).<sup>†4</sup> Observing invariance under arrow reversal, it may not come as a surprise that equations (2.17) and (2.19) also are the conditions for  $m$  being a coalgebra morphism, and equations (2.18) and (2.20) tell us that  $\eta$  is as well.

Next, we introduce a notion of invertibility which extends a bialgebra to a Hopf algebra.

**Definition 2.15** (meta-Hopf algebra). A meta-Hopf algebra (or meta-Hopf monoid) is a bialgebra  $H$  together with a map  $S: H \rightarrow H$  called the antipode, which satisfies  $\Delta_{1,2}^1 // S_1^1 // m_1^{1,2} = \epsilon^1 // \eta_1 = \Delta_{1,2}^1 // S_2^2 // m_1^{1,2}$ . As a commutative diagram, this looks like equation (2.21)

$$\begin{array}{ccccc}
H_{\{1\}} & \xrightarrow{\epsilon^1} & H_\emptyset & \xrightarrow{\eta_1} & H_{\{1\}} \\
& \searrow \Delta_{1,2}^1 & & \nearrow m_1^{1,2} & \\
& & H_{\{1,2\}} & \xrightleftharpoons[S_1^1]{S_2^2} & H_{\{1,2\}}
\end{array} \quad (2.21)$$

In order to do knot theory, we need an algebraic way to represent a crossing of two strands. This is accomplished by the so-called  $\mathcal{R}$ -matrix:

**Definition 2.16** (quasitriangular meta-Hopf algebra). A quasitriangular meta-Hopf algebra (or quasitriangular meta-Hopf monoid) is a Hopf algebra

<sup>†4</sup> While notation explicitly naming each tensor factor appears cumbersome in these diagrams, it will prove invaluable later when used on tangle diagrams, so we leave it as is for the sake of consistency.

$H$ , together with an invertible element  $\mathcal{R}_{i,j} \in H_{i,j}$ , called the  $\mathcal{R}$ -matrix, which satisfies the following properties: (we will denote the inverse by  $\overline{\mathcal{R}}$ )

$$\mathcal{R}_{12} \parallel \Delta_{23}^2 = \mathcal{R}_{a3} \mathcal{R}_{b2} \parallel m_1^{ab} \quad (2.22)$$

$$\mathcal{R}_{13} \parallel \Delta_{12}^1 = \mathcal{R}_{1a} \mathcal{R}_{2b} \parallel m_3^{ab} \quad (2.23)$$

$$\Delta_{21}^1 = \Delta_{12}^1 \mathcal{R}_{1_i,2_i} \overline{\mathcal{R}}_{1_f,2_f} \parallel m_1^{1_i,1,1_f} \parallel m_2^{2_i,2,2_f} \quad (2.24)$$

**Definition 2.17** (Drinfeld element). In a quasitriangular meta-Hopf algebra  $H$ , the Drinfeld element,  $u \in H$  is given by:

$$u := \mathcal{R}_{21} \parallel S_1^1 \parallel m^{12} \quad (2.25)$$

**Definition 2.18** (monodromy). Each quasitriangular meta-Hopf algebra has a monodromy  $Q_{12} := \mathcal{R}_{12} \mathcal{R}_{34} \parallel m_1^{14} \parallel m_2^{23}$ . Its inverse will be denoted  $\overline{Q}_{12} = \overline{\mathcal{R}}_{12} \overline{\mathcal{R}}_{34} \parallel m_1^{14} \parallel m_2^{23}$ .

*Proof.* See Majid's work in [Maj] or Etingof and Schiffmann in [ES] for more details on this standard result. Note that the proof does not rely on the additive structure of the Hopf algebra, which allows us to extend this result to the realm of meta-Hopf algebras.  $\square$

**Definition 2.19** (ribbon meta-Hopf algebra). A quasitriangular meta-Hopf algebra  $H$  is called ribbon if it has an element  $\nu \in Z(H)$  such that:

$$\nu_1 \nu_2 \parallel m^{12} = u_1 u_2 \parallel S_2^2 \parallel m^{12} \quad (2.26)$$

$$\nu_1 \parallel \Delta_{12}^1 = \nu_1 \nu_2 \parallel \overline{Q}_{34} \parallel m_1^{13} \parallel m_2^{24} \quad (2.27)$$

$$\nu \parallel S = \nu \quad (2.28)$$

$$\nu \parallel \epsilon = \eta \parallel \epsilon = 1 \quad (2.29)$$

**Definition 2.20** (distinguished grouplike element (spinner)). A distinguished grouplike element (or spinner) in a quasitriangular meta-Hopf algebra  $H$  is an invertible element  $C \in H$  (with inverse  $\bar{C}$ ) such that for all  $x \in H$ :

$$C_1 \nu_2 C_3 \parallel S_2^2 \parallel m^{123} = \nu \quad (2.30)$$

$$C_1 \parallel \Delta_{12}^1 = C_1 C_2 \quad (2.31)$$

$$C \parallel S = \bar{C} \quad (2.32)$$

$$C_1 x_2 \bar{C}_3 \parallel m^{1,2,3} = x \parallel S \parallel S \quad (2.33)$$

$$C \parallel \epsilon = \eta \parallel \epsilon = 1 \quad (2.34)$$

**Lemma 2.21** (spinners and ribbon Hopf algebras). *If a Hopf algebra has either a ribbon element  $\nu$  or a spinner  $C$ , then it must have the other as well, given by the formula:  $C_1 \nu_2 \parallel m^{12} = \mathbf{u}$ .*

## 2.4 THE META-ALGEBRA OF TANGLE DIAGRAMS

The particular structures introduced were chosen for their ability to represent the topological properties of knotted objects. We will now introduce the notion of a tangle and demonstrate its meta-algebraic structure.

### *Upright tangles*

For our purposes, a tangle will be visualised as follows: take a stiff circular metal frame and attach a collection of strings to the wire, ensuring that the strings always remain inside the circle, and that each string is tied to the metal frame in two unique locations (that is, no two strings share an endpoint).

**Definition 2.22** (pure tangle). A pure tangle is an embedding of line segments (called components or strands) into the thickened unit disk  $D \times [-1, 1]$  (or a disjoint union of such disks) such that the endpoints of the line segments are fixed along  $\partial D \times \{0\}$ . Two pure tangles are considered equivalent if there exists an isotopy of the embedding which fixes the endpoints of the strands. The term “pure” refers to the absence of closed loops.

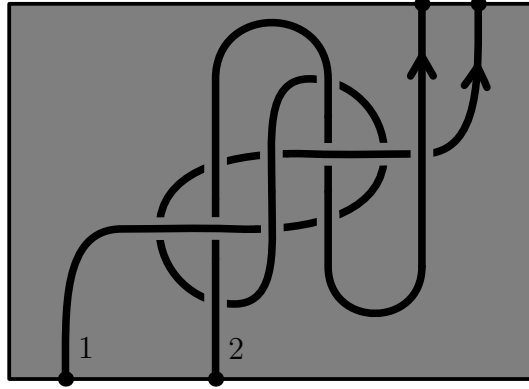


Figure 2.1: Example of a tangle with strands labelled 1 and 2.

**Definition 2.23** (framed tangle). The object which is more natural for us to deal with is the framed tangle, which is a pure tangle together with a choice of section of the normal bundle for each component. One may equivalently think of replacing the strings in a pure tangle with thin ribbons. Unless otherwise mentioned, it will be assumed that all tangles are framed.

In order to best capture the combinatorial properties of a tangle, observe that a generic projection of a tangle to its central core  $D \times \{0\}$  will result in the strands forming a graph, with each crossing of two strands in the tangle producing a vertex in the graph. By assigning to each vertex the sign of the corresponding crossing (either “positive” or “negative”), we end up with a combinatorial object which is equivalent to the original concept of a tangle.

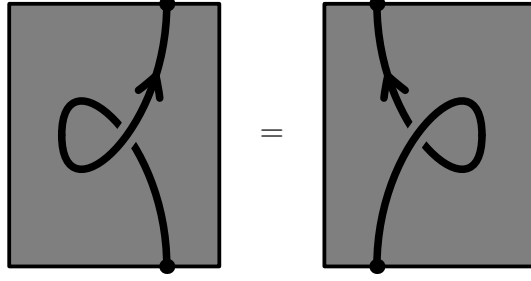
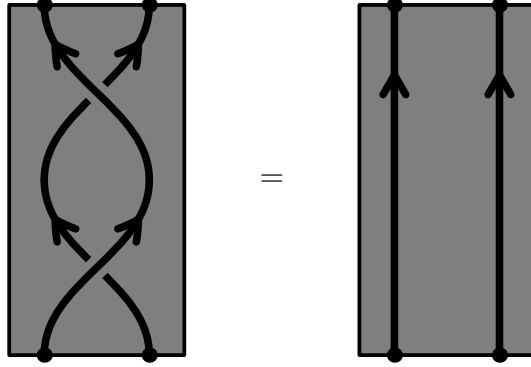
**Definition 2.24** (pure tangle diagram). A pure tangle diagram is a projection of a tangle onto its central core such that all the line segments are immersions which intersect both the boundary disk and the other strands transversally, together with an assignment of a sign to each strand intersection. Small open neighbourhoods of these intersections are called crossings, while the complement of the crossings is a collection of embedded line segments called arcs.

Two pure tangle diagrams are considered equivalent if they differ by a finite sequence of Reidemeister moves, as outlined in ??–2.4

The rotation numbers of arcs will play a role in this thesis, so we will capture these data in the following way (as described in ??):

**Definition 2.25** (upright pure tangle diagram). We will put a further requirement on our tangles: each arc must begin and end pointing upwards,



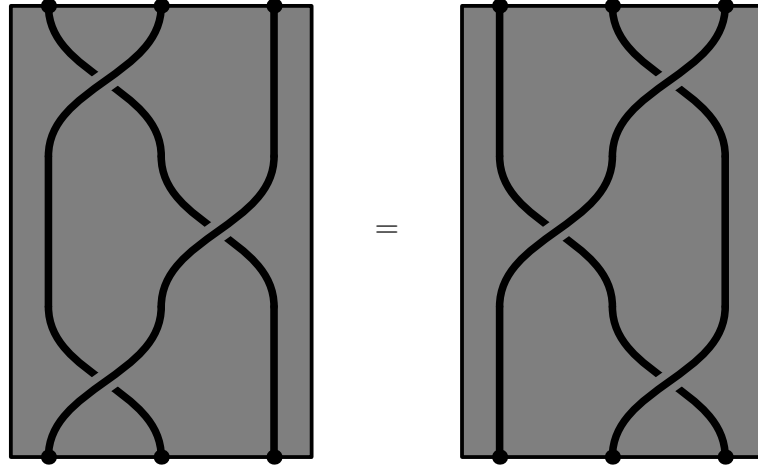
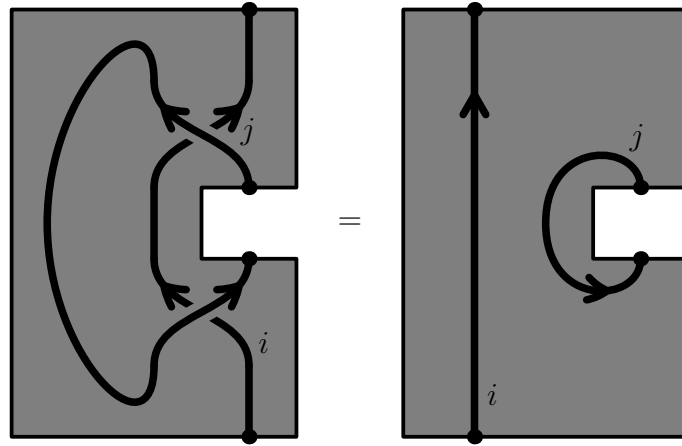
Figure 2.2: Reidemeister move  $R1'$ Figure 2.3: Reidemeister move  $R2$ 

and each crossing must involve only upward-pointing curves. This way, each arc will have a well-defined integer rotation number. Two tangles are considered equivalent if they agree under the “rotational Reidemeister moves”. Such tangle diagrams will be called upright tangle diagrams.

*Remark 2.26.* This concept was first introduced by Louis Kauffman in [Kau] under the name rotational virtual knot theory. In this formulation, we insist that all strands end pointing upwards instead of merely requiring that endpoint vectors are vertical, so we will use the term “upright” to remind the reader of this difference.

Fortunately, ambient isotopy allows us to rotate any classical tangle into an upward-pointing form. Additionally, there is only one way to do this. We reproduce the proof of this fact by Bar-Natan and van der Veen in [BNvdV] below:

**Lemma 2.27** (tangles inject into upright tangles). *To each pure tangle diagram  $D$  there exists an upright pure tangle diagram  $D'$  obtained from  $D$  by a planar isotopy. Further, if  $D''$  is another such upright pure tangle diagram*

Figure 2.4: Reidemeister move  $R3$ Figure 2.5: The (cyclic) rotational Reidemeister move  $R2_{\text{rot}}$ 

obtained from  $D$ , then  $D'$  and  $D''$  differ by a finite sequence of rotational Reidemeister moves and a change of rotation number at the endpoints.

*Proof.* Each arc and crossing in the diagram  $D$  may be rotated so that its endpoints are pointing upwards, giving rise to a diagram  $D'$ . Two (nonupright) tangle diagrams are equivalent when they differ by a finite sequence of Reidemeister moves. Each of these Reidemeister may also be rotated to an equivalence of upright tangles, each of which is given as a rotational Reidemeister move ??-??. The last possibility is the rotation of an entire crossing, which is covered by figure 2.6.  $\square$

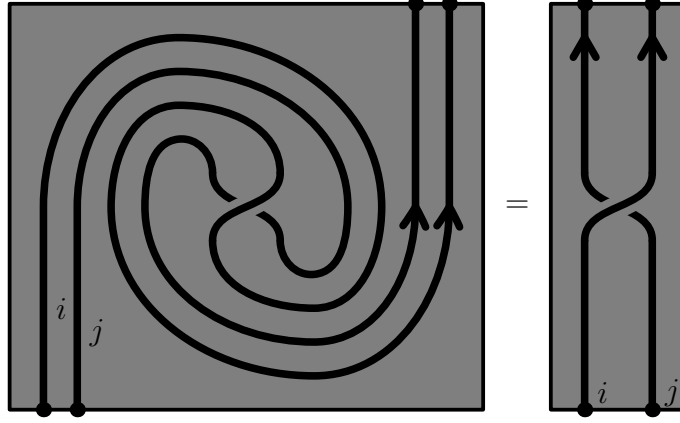


Figure 2.6: The whirling move

*The meta-algebra structure of upright tangle diagrams*

Let us now formally connect tangle diagrams with meta-algebras.

**Theorem 2.28** (tangles form a ribbon meta-Hopf algebra). *Define  $\mathcal{T}_X$  to be the set of upright tangles with  $|X|$  strands, each labelled by an element  $i \in X$ . Then the collection  $\{\mathcal{T}_X\}_X$  forms a quasitriangular ribbon meta-Hopf algebra with the following operations:*

- *multiplication  $m_k^{ij}[X]$  takes a tangle with strands  $X \sqcup \{i, j\}$  and glues the end of strand  $i$  to strand  $j$ , labelling the resulting strand  $k$ .*
- *the unit  $\eta_i[X]$  takes a tangle diagram with strands  $X$  and introduces a new strand  $i$  which does not touch any of the other strands.*
- *the comultiplication  $\Delta_{jk}^i[X]$  takes a tangle with strands  $X \sqcup \{i\}$  and doubles strand  $i$ , separating the two strands along the framing of strand  $i$ , calling the right strand  $j$  and the left one  $k$ .<sup>†5</sup>*
- *the counit  $\epsilon^i[X]$  takes a tangle with strands indexed by  $X \sqcup \{i\}$  and returns the tangle with strand labelled by  $i$  deleted.*
- *The antipode  $S_j^i[X]$  takes a tangle with strands labelled by  $X \sqcup \{i\}$  and reverses the direction of strand  $i$ , then adds a counter-clockwise cap to the new beginning, and a clockwise cup to the end. This new strand is*

<sup>†5</sup> While this convention appears unfortunate, we follow the notation laid out in [BNvdV] so that the antipode and spinner have a more memorable representation, namely looking like the letters they are represented by (see ?? for more details).

called  $j$ . When applied to a single vertical strand, the resulting tangle looks like the letter “S”.

- the  $\mathcal{R}$ -matrix  $\mathcal{R}_{ij}$  is given by the two-strand tangle with a single positive crossing of strand  $i$  over strand  $j$ . The inverse  $\mathcal{R}$ -matrix  $\overline{\mathcal{R}}_{ij}$  is the two-strand tangle with a negative crossing of strand  $i$  over strand  $j$ .
- The spinner  $C_i[X]$  takes a tangle in  $\mathcal{T}_X$  and adds a new strand with rotation number 1 which has no interactions with any other strands. This new strand looks like the letter “C”.

*Remark 2.29.* One may object that strand-stitching  $m_k^{ij}$  is not defined when the endpoint of strand  $i$  is not adjacent to the starting point of strand  $j$ . This issue is resolved in multiple ways:

1. Extend the collection of tangles we work with to include virtual tangles. This generalization of tangles deals exactly with the issue that multiplication need not produce a planar tangle diagram. In fact, virtual tangles can be thought of as merely non-planar tangle diagrams.
2. Commit to only apply multiplication when doing so would result in a valid (classical) tangle. This is the approach we will take when performing computations on tangles.

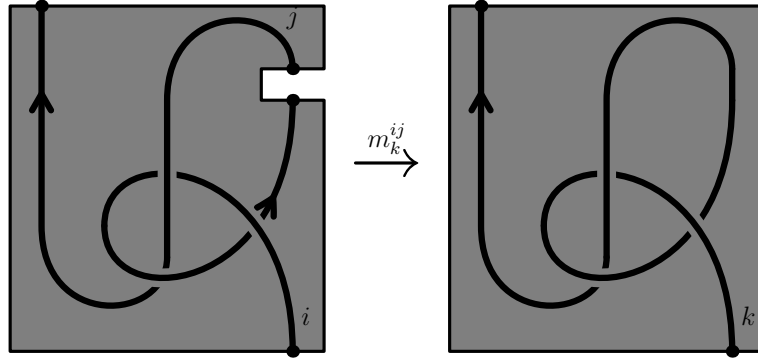
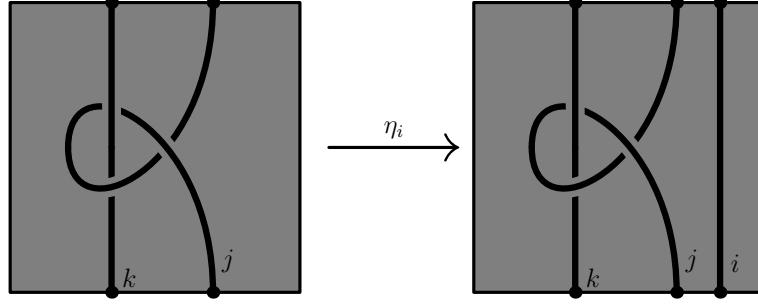
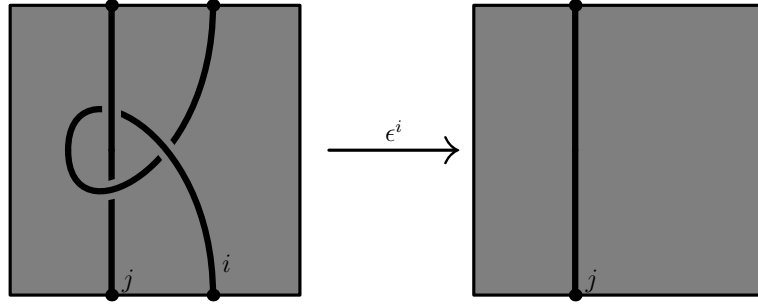


Figure 2.7: Multiplication  $m_k^{ij}$  stitches two strands in a tangle together.

*Proof.* Associativity of multiplication (equation (2.13)) follows from the fact that stitching strands together amounts to concatenating the order

Figure 2.8: The unit  $\eta_i$  introduces a new strand in a tangle.Figure 2.9: The counit  $\epsilon^i$  deletes a strand in a tangle.

of the crossings each strand interacts with. Since list concatenation is an associative operation, associativity follows in this case as well.

Adding a non-interacting strand to a diagram, then stitching it to an existing strand (equation (2.14)) does not change any of the combinatorial data in the diagram, and results in identical diagrams.

Establishing coassociativity (equation (2.15)) amount to the same argument that cutting a piece of paper into three strips does not depend on the order of cutting.

The counit identity (equation (2.16)) states deleting a strand is the same operation as first doubling it, then deleting both resulting strands.

The meta-bialgebra axioms we verify next:

Equation (2.17) states that if two strands are stitched together, then the resulting strand is doubled, this could have equivalently been achieved by doubling each of the original strands, then performing a stitching on both resulting pairs of strands.

Equation (2.19) simply states that stitching two strands together, then removing the resulting strand could have equally been achieved by removing both of the original strands without stitching them first.

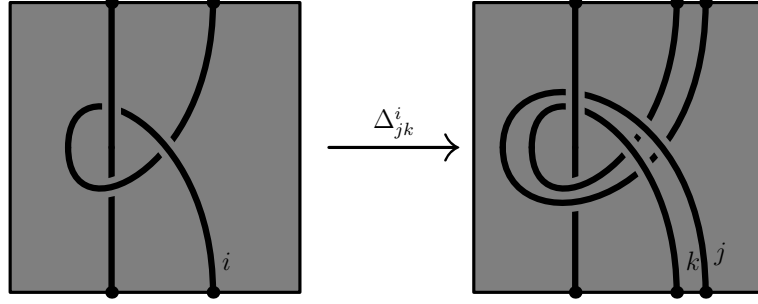


Figure 2.10: The comultiplication  $\Delta_{jk}^i$  doubles a strand in a tangle along its framing. Notice the right-to-left strand labels.

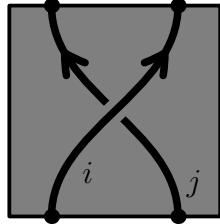


Figure 2.11: The  $\mathcal{R}$ -matrix  $\mathcal{R}_{ij}$  represents a tangle with a single positive crossing.

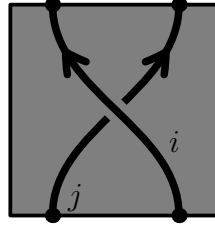


Figure 2.12: The inverse  $\mathcal{R}$ -matrix  $\mathcal{R}_{ij}^{-1}$  represents a tangle with a single negative crossing.

Equation (2.20) states that introducing a strand, then immediately removing it is the identity operation.

Equation (2.18) says that doubling a newly-introduced (and therefore free of crossings) strand is the same operation as introducing two strands separately. (For those worried that this equation depends on the location of the separately introduced strands, this is one place that the use of virtual tangles will be used, which does not heed the relative locations of disjoint strands.)

Equation (2.21) states that when a strand is doubled, then one of the two strands is reversed, multiplying the two strands together results in a strand which can be rearranged to not interact with any of the other strands. This can be readily seen, as this newly-created strand looks like a snake weaving through the tangle diagram. One can remove the snake by applying a series of Reidemeister 2 moves, resulting in a strand disjoint from the rest of the diagram. This is the same as deleting the original strand, then introducing a new disjoint one.

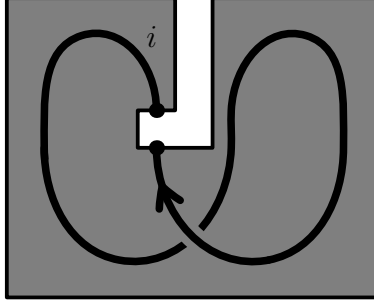


Figure 2.13: The Drinfeld element in the meta-Hopf algebra of tangles.

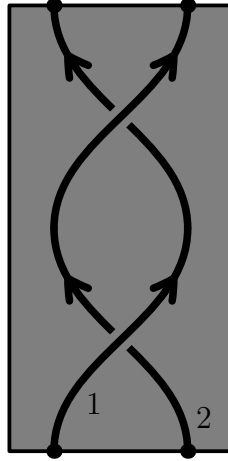


Figure 2.14: The monodromy in the meta-Hopf algebra of tangles.

The quasitriangular axioms are equalities of pairs of three-strand tangles:

- Equations (2.22) and (2.23) tell us that doubling a strand involved in a single crossing can also be built by adjoining two crossings together.
- Equation (2.24) tells us that we can swap the order of a doubled strand by adding crossings to either end (reminiscent of a Reidemeister 2 move)

Finally, we observe that the quotient we introduce to tangle diagrams by the Reidemeister moves does not introduce any new relations. Reidemeister 2 follows from the invertibility of the  $\mathcal{R}$ -matrix. Next, it is readily seen that the quasitriangular relations governing the  $\mathcal{R}$ -matrix force it to solve the Yang-Baxter equation, which is one equivalent to the Reidemeister 3 in this case.

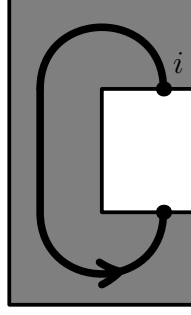


Figure 2.15: The spinner  $C_i$  represents a strand with rotation number 1.

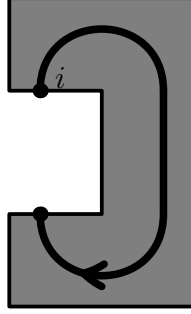


Figure 2.16: The inverse spinner  $\overline{C}_i$  represents a strand with rotation number  $-1$ .

Using lemma 2.21, it is enough to verify the spinner axioms (equations (2.30) to (2.34)). All these axioms have corresponding pictures one can draw, keeping in mind the orientations in the definitions of the relevant operations.  $\square$

## 2.5 THE $yba x$ META-ALGEBRA

Here we define the ribbon Hopf algebra  $U$ .

Define the Lie algebra

$$\mathfrak{g} := \text{span} \left\{ \mathbf{y}, \mathbf{b}, \mathbf{a}, \mathbf{x} \mid [\mathbf{a}, \mathbf{x}] = \mathbf{x}, [\mathbf{a}, \mathbf{y}] = -\mathbf{y}, [\mathbf{x}, \mathbf{y}] = \mathbf{b}, [\mathbf{b}, \cdot] = 0 \right\} \quad (2.35)$$

Then the algebra  $U$  is defined to be the universal enveloping algebra  $\mathfrak{U}(\mathfrak{g})$ . The bialgebra structure of  $U$  is: for any  $\mathbf{z} \in \{\mathbf{y}, \mathbf{b}, \mathbf{a}, \mathbf{x}\}$ , we have

$$\Delta_{i,j}(\mathbf{z}) = \mathbf{z}_i + \mathbf{z}_j \quad (2.36)$$

$$\epsilon(\mathbf{z}) = 0 \quad (2.37)$$



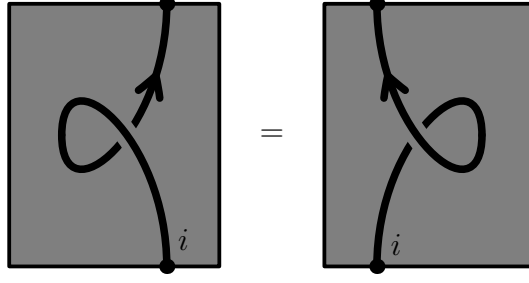


Figure 2.17: A ribbon element  $\nu_i$  in the meta-Hopf algebra of tangles. One can use lemma 2.21 to verify this is compatible with the spinner.

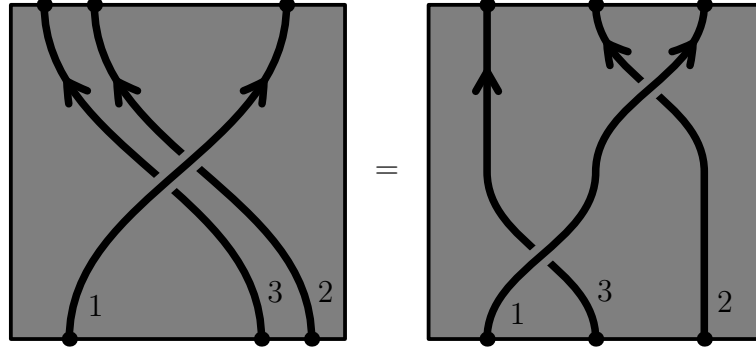


Figure 2.18: Example of a tangle satisfying equation (2.22)

Next, we define the Hopf algebra structure by defining the antipode

$$S(\mathbf{z}) = -\mathbf{z} \quad (2.38)$$

The ribbon structure of  $U$  requires we introduce both the  $\mathcal{R}$ -matrix and the spinner  $C$ :

$$\mathcal{R}_{i,j} := \exp\left(\mathbf{a}_j \mathbf{b}_i + \frac{1 - \mathbf{B}_i}{\mathbf{b}_i} \mathbf{y}_i \mathbf{x}_j\right) \quad (2.39)$$

$$C := \sqrt{\mathbf{B}} \quad (2.40)$$

## 2.6 MORPHISMS BETWEEN META-OBJECTS

When equipped with meta-structures on both tangles and an algebraic object, we can define a tangle invariant by considering a morphism between the meta-objects.

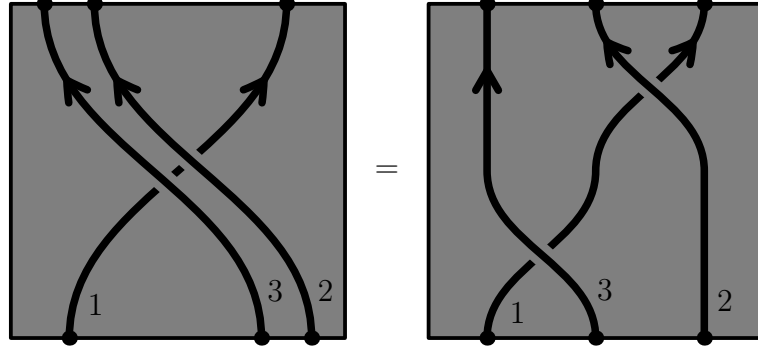


Figure 2.19: Example of a tangle satisfying equation (2.23)

**Definition 2.30** (morphism of meta-objects). Let  $\{A_X\}_X$  and  $\{B_X\}_X$  be compatible meta-objects (i.e. ones with the same operations and relations between the operations). A morphism  $\phi$  between these meta-objects is map  $\phi: \{A_X\}_X \rightarrow \{B_X\}_X$  sending  $A_X \mapsto B_X$  such that for each map  $f_Y^X$  in  $A$ ,  $\phi(f_Y^X) = f_Y^X$  in  $B$ .

*Upright tangle invariants from a ribbon meta-Hopf algebra*

We define a  $U$ -valued tangle invariant in the following way:

1. Given a pure tangle, disconnect each crossing from its neighbours, as well as each arc with a nonzero rotation number.
2. Replace each crossing with an  $\mathcal{R}$ -matrix  $\mathcal{R}_{ij} \in U_{\{i,j\}}$ , and each rotation of an arc with a spinner  $C_i \in U_{\{i\}}$ .
3. For each disconnection, there is a corresponding stitching operation required to bring the tangle back to its original state. Replace each stitching operation with a multiplication operation in  $U$ .

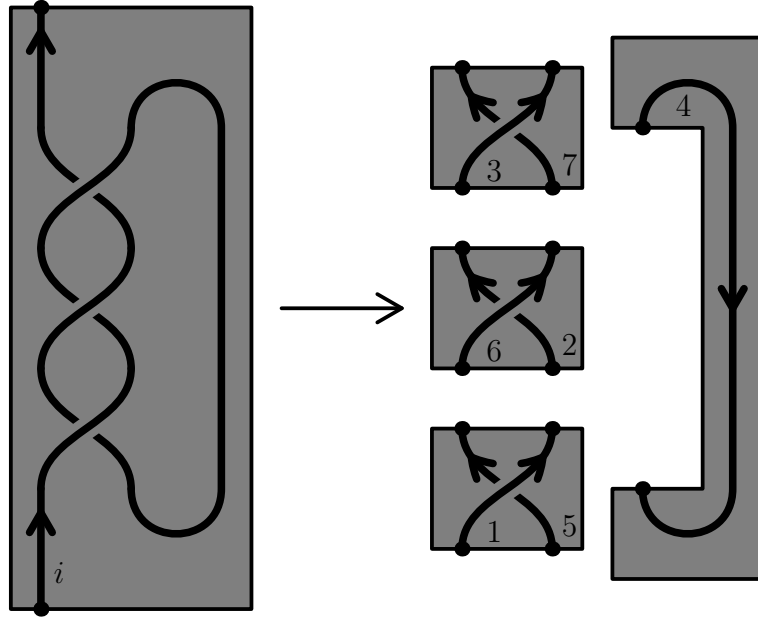


Figure 2.20: Breaking up a tangle into its constituent components. The left-hand tangle is obtainable by the right-hand one by applying the map  $m_i^{1,2,\dots,7}$ .

# PERTURBED GAUSSIANS

---

We now summarize the work of Bar-Natan and van der Veen in [BNvdV], which develops a universal knot invariant using perturbed Gaussians.

## 3.1 EXPRESSING MORPHISMS AS GENERATING FUNCTIONS

When defining a morphism-valued tangle invariant, one needs a compact way of encoding the morphism. In [BNvdV] this is achieved through the use of generating functions, whose definition we reproduce below:

For  $A$  and  $B$  finite sets, consider the set  $\text{hom}(\mathbb{Q}[z_A], \mathbb{Q}[z_B])$  of linear maps between multivariate polynomial rings. Such a map is determined by its values on the monomials  $z_A^{\mathbf{n}}$  for each multi-index  $\mathbf{n} \in \mathbb{N}^A$ .

**Definition 3.1** (Exponential generating function). The exponential generating function of a map  $\Phi: \mathbb{Q}[z_A] \rightarrow \mathbb{Q}[z_B]$  between polynomial spaces is

$$\mathcal{G}(\Phi) := \sum_{\mathbf{n} \in \mathbb{N}^A} \frac{\Phi(z_A^{\mathbf{n}})}{\mathbf{n}!} \zeta_A^{\mathbf{n}} \in \mathbb{Q}[z_B][[\zeta_A]] \quad (3.1)$$

*Remark 3.2.* Extending the definition of  $\Phi$  to  $\mathbb{Q}[z_B][[\zeta_A]]$  by the extending scalars to  $\mathbb{Q}[[\zeta_A]]$  gives us an equivalent formulation:

$$\mathcal{G}(\Phi) = \Phi \left( \sum_{\mathbf{n} \in \mathbb{N}^A} \frac{(z_A \zeta_A)^{\mathbf{n}}}{\mathbf{n}!} \right) = \Phi(\mathcal{G}(\text{id}_{\mathbb{Q}[z_A]})) \quad (3.2)$$

By the PBW theorem, we know that  $U$  is isomorphic as a vector space to the polynomial ring  $\mathbb{Q}[y, b, a, x]$  by choosing an ordering of the generators (following [BNvdV], we use  $(\mathbf{y}, \mathbf{b}, \mathbf{a}, \mathbf{x})$ ):

$$\begin{aligned} \mathbb{O}: \mathbb{Q}[y, b, a, x] &\xrightarrow{\sim} U \\ y^{n_1} b^{n_2} a^{n_3} x^{n_4} &\mapsto \mathbf{y}^{n_1} \mathbf{b}^{n_2} \mathbf{a}^{n_3} \mathbf{x}^{n_4} \end{aligned} \quad (3.3)$$

To efficiently describe  $\mathbb{k}$ -linear maps between tensor powers of the algebra  $U$ , we define categories  $\mathcal{U}$ ,  $\mathcal{H}$ , and  $\mathcal{C}$  with objects finite sets and morphisms:

$$\mathrm{Hom}_{\mathcal{U}}(J, K) := \mathrm{Hom}_{\mathbb{k}}(U^{\otimes J}, U^{\otimes K}) \quad (3.4)$$

$$\mathrm{Hom}_{\mathcal{H}}(J, K) := \mathrm{Hom}_{\mathbb{k}}(\mathbb{Q}[z_J], \mathbb{Q}[z_K]) \quad (3.5)$$

$$\mathrm{Hom}_{\mathcal{C}}(J, K) := \mathbb{Q}[z_K][[\zeta_J]] \quad (3.6)$$

There exist monoidal isomorphisms between these categories, namely  $\mathbb{O}: \mathcal{H} \xrightarrow{\sim} \mathcal{U}$  and  $\mathcal{G}: \mathcal{H} \xrightarrow{\sim} \mathcal{C}$  as introduced in equations (3.1) and (3.3).

We use this formulation because of the existence of a computationally amenable subcategory of  $\mathcal{C}$  which contains the image of this invariant.

#### *Formulating composition in other categories*

Composing operations in  $\mathcal{U}$  or  $\mathcal{H}$  is straightforward to define, but lacks a closed form. However, on  $\mathcal{C}$ , the corresponding definition of composition takes the following form (quoted from [BNvdV, Lemma 3]):

**Lemma 3.3** (Composition of generating functions). *Suppose  $J, K, L$  are finite sets and  $\phi \in \mathrm{Hom}(\mathbb{Q}[z_J], \mathbb{Q}[z_K])$  and  $\psi \in \mathrm{Hom}(\mathbb{Q}[z_K], \mathbb{Q}[z_L])$ . We have*

$$\mathcal{G}(\phi // \psi) = \left( \mathcal{G}(\phi) \Big|_{z_K \rightarrow \partial \zeta_K} \mathcal{G}(\psi) \right) \Big|_{\zeta_K=0} \quad (3.7)$$

Since the above notation will occur several times, we will use the notion of contraction used by Bar-Natan and van der Veen (taken from [BNvdV, Definition 4]):

**Definition 3.4** (Contraction). Let  $f \in \mathbb{k}[[r, s]]$  be a powerseries. The contraction of  $f = \sum_{k,l} c_{k,l} r^k s^l$  along the pair  $(r, s)$  is:

$$\langle f \rangle_{(r,s)} := \sum_k c_{k,k} k! = \sum_{k,l} c_{k,l} \partial_s^k s^l \Big|_{s=0} \quad (3.8)$$

Further, this notation is to be extended to allow for multiple consecutive contractions for  $f \in \mathbb{k}[[r_i, s_i]]_{i \leq n}$ :

$$\langle f \rangle_{((r_i)_i, (s_i)_i)} := \left\langle \left\langle \langle f \rangle_{(r_1, s_1)} \right\rangle_{(r_2, s_2)} \cdots \right\rangle_{(r_n, s_n)} \quad (3.9)$$

It is important to note that contraction does not always define a convergent expression. We will focus our attention on cases when convergence is well-defined, and especially those where the computation is accessible.

The theorem we will rely heavily on in this thesis is the following, taken from [BNvdV, Theorem 6]:

**Theorem 3.5** (Contraction theorem). *For any  $n \in \mathbb{N}$ , consider the ring  $R_n = \mathbb{Q}[r_j, g_j][[s_j, W_{ij}, f_j \mid 1 \leq i, j \leq n]]$ . Then*

$$\langle \mathbf{e}^{gs+rf+rWs} \rangle_{r,s} = \det(\tilde{W}) \mathbf{e}^{g\tilde{W}f} \quad (3.10)$$

where  $\tilde{W} = (1 - W)^{-1}$ .

The main takeaway of this theorem is this: morphisms whose generating functions are Gaußians have a clean formula for composition. Furthermore, this formula is computationally reasonable, growing only polynomially in complexity with  $n$ . This is contrasted with the conventional approach of choosing a representation  $V$  of  $U$ . When one considers morphisms between large tensor powers  $V^{\otimes n}$ , the computational complexity is exponential in  $n$ .

#### *Expressing Hopf algebra operations as perturbed Gaußians*

Using this vector space isomorphism, [BNvdV] expresses all Hopf algebra operations as power series in a closed form, namely as perturbed Gaußians.

**Theorem 3.6** (The meta-Hopf structure of  $U$  is Gaußian). *Each of the meta-Hopf algebra operations for  $U$  as defined in section 2.5 all have the form of a perturbed Gaußian. That is:*

$$\mathcal{G}(m_k^{ij}) = \exp\left((\alpha_i + \alpha_j)a_k + (\beta_i + \beta_j + \xi_i\eta_j)b_k + \left(\frac{\xi_i}{\mathcal{A}_j} + \xi_j\right)x_k + \left(\frac{\eta_j}{\mathcal{A}_i} + \eta_i\right)y_k\right) \quad (3.11)$$

$$\mathcal{G}(\eta_i) = 1 \quad (3.12)$$

$$\mathcal{G}(\Delta_{jk}^i) = \exp(\beta_i(b_j + b_k) + \alpha_i(a_j + a_k) + \eta_i(y_j + y_k) + \xi_i(x_j + x_k)) \quad (3.13)$$

$$\mathcal{G}(\epsilon^i) = 1 \quad (3.14)$$

$$\mathcal{G}(S_i^i) = \exp(-a_i\alpha_i - b_i\beta_i - \eta_i\mathcal{A}_iy_i - \mathcal{A}_i\xi_ix_i + \eta_i\mathcal{A}_i\xi_ib_i) \quad (3.15)$$

$$\mathcal{G}(\mathcal{R}_{ij}) = \exp\left(a_jb_i + \frac{B_i - 1}{-b_i}y_ix_j\right) \quad (3.16)$$

$$\mathcal{G}(C_i) = \sqrt{B_i} \quad (3.17)$$

$$\mathcal{G}(\nu_i) = \sqrt{B_i} \exp\left(a_ib_i + \frac{1 - B_i}{b_i}x_iy_i\right) \quad (3.18)$$

*Proof.* To prove equation (3.11), we first note the Weyl canonical commutation relation:

$$\mathbf{e}^{\xi\mathbf{x}}\mathbf{e}^{\eta\mathbf{y}} = \mathbf{e}^{-\xi\eta\mathbf{b}}\mathbf{e}^{\eta\mathbf{y}}\mathbf{e}^{\xi\mathbf{x}} \quad (3.19)$$

Secondly, using equation (4.6) and  $\mathcal{A} := \mathbf{e}^\alpha$ , we compute

$$\mathbf{e}^{\alpha\mathbf{a}}\mathbf{e}^{\eta\mathbf{y}} = \mathbf{e}^{\alpha\mathbf{a}} \sum_n \frac{(\eta\mathbf{y})^n}{n!} = \sum_n \frac{(\eta\mathbf{y})^n}{n!} \mathbf{e}^{\alpha(\mathbf{a}-n)} = \sum_n \frac{(\eta\mathbf{y})^n}{n!} \mathbf{e}^{\alpha\mathbf{a}} \mathcal{A}^{-n} = \mathbf{e}^{\frac{\eta}{\mathcal{A}}\mathbf{y}} \mathbf{e}^{\alpha\mathbf{a}} \quad (3.20)$$

similarly,

$$\mathbf{e}^{\xi\mathbf{x}}\mathbf{e}^{\alpha\mathbf{a}} = \sum_n \frac{(\xi\mathbf{x})^n}{n!} \mathbf{e}^{\alpha\mathbf{a}} = \sum_n \mathbf{e}^{\alpha(\mathbf{a}-n)} \frac{(\xi\mathbf{x})^n}{n!} = \sum_n \mathbf{e}^{\alpha\mathbf{a}} \frac{\left(\frac{\xi\mathbf{x}}{\mathcal{A}}\right)^n}{n!} = \mathbf{e}^{\alpha\mathbf{a}} \mathbf{e}^{\frac{\xi}{\mathcal{A}}\mathbf{x}} \quad (3.21)$$

Using this relation allows us to commute exponentials past each other to bring expressions into *yba*-order. Below we omit the index  $k$  for readability:

$$\begin{aligned}
\mathcal{G}(m^{ij}) &= (\mathbb{O}^{-1} \circ m^{ij} \circ \mathbb{O})(e^{\eta_i y_i + \beta_i b_i + \alpha_i a_i + \xi_i x_i} e^{\eta_j y_j + \beta_j b_j + \alpha_j a_j + \xi_j x_j}) \\
&= \mathbb{O}^{-1}(e^{\eta_i y} e^{\beta_i b} e^{\alpha_i a} e^{\xi_i x} e^{\eta_j y} e^{\beta_j b} e^{\alpha_j a} e^{\xi_j x}) \\
&= \mathbb{O}^{-1}(e^{\eta_i y} e^{\beta_i b} e^{\alpha_i a} (e^{-\xi_i \eta_j b} e^{\eta_j y} e^{\xi_i x}) e^{\beta_j b} e^{\alpha_j a} e^{\xi_j x}) \\
&= \mathbb{O}^{-1}\left(e^{(\beta_j + \beta_i - \xi_i \eta_j) b} e^{\eta_i y} e^{\frac{\eta_j}{\mathcal{A}_i} y} e^{\alpha_i a} e^{\xi_i x} e^{\alpha_j a} e^{\xi_j x}\right) \\
&= \mathbb{O}^{-1}\left(e^{(\beta_j + \beta_i - \xi_i \eta_j) b} e^{(\eta_i + \frac{\eta_j}{\mathcal{A}_i}) y} e^{\alpha_i a} e^{\alpha_j a} e^{\frac{\xi_i}{\mathcal{A}_j} x} e^{\xi_j x}\right) \\
&= \mathbb{O}^{-1}\left(e^{(\eta_i + \frac{\eta_j}{\mathcal{A}_i}) y} e^{(\beta_j + \beta_i - \xi_i \eta_j) b} e^{(\alpha_i + \alpha_j) a} e^{(\frac{\xi_i}{\mathcal{A}_j} + \xi_j) x}\right) \\
&= e^{(\eta_i + \frac{\eta_j}{\mathcal{A}_i}) y} e^{(\beta_j + \beta_i - \xi_i \eta_j) b} e^{(\alpha_i + \alpha_j) a} e^{(\frac{\xi_i}{\mathcal{A}_j} + \xi_j) x}
\end{aligned} \tag{3.22}$$

Since this expression is now written in the **yba**-order, we conclude that the corresponding generating function is this same expression, but written with commuting variables.

The other computation we must verify is the antipode, which follows similarly:

$$\begin{aligned}
\mathcal{G}(S) &= (\mathbb{O}^{-1} \circ S \circ \mathbb{O})(e^{\eta y + \beta b + \alpha a + \xi x}) \\
&= \mathbb{O}^{-1}(e^{-\xi x} e^{-\alpha a} e^{-\beta b} e^{-\eta y}) \\
&= \mathbb{O}^{-1}(e^{-\xi x} e^{-\mathcal{A} \eta y} e^{-\alpha a} e^{-\beta b}) \\
&= \mathbb{O}^{-1}(e^{-\xi \mathcal{A} \eta b} e^{-\mathcal{A} \eta y} e^{-\xi x} e^{-\alpha a} e^{-\beta b}) \\
&= \mathbb{O}^{-1}(e^{-\mathcal{A} \eta y} e^{-\xi \mathcal{A} \eta b} e^{-\beta b} e^{-\xi x} e^{-\alpha a}) \\
&= \mathbb{O}^{-1}(e^{-\mathcal{A} \eta y} e^{-(\xi \mathcal{A} \eta + \beta) b} e^{-\alpha a} e^{-\xi \mathcal{A} x})
\end{aligned} \tag{3.23}$$

Finally, equations (3.12), (3.14) and (3.16) to (3.18) follow immediately.  $\square$

### *Notational conventions*

The generating function of a tangle is not the entirety of this definition, for the additional data is the domain and codomain of the corresponding morphism.

We will thereby write a morphism with domain  $D$ , codomain  $C$ , and generating function  $f(\zeta_D, z_C)$  as  $f(\zeta_D, z_C)_C^D$ .



When applied to knots, this invariant computes the Alexander polynomial  $\Delta$ :

$$Z(K_{3,1}) = \left( \frac{1}{B_1^{-1} + 1 + B_1^1} \right)_{\{1\}}^{\emptyset} = \Delta(K_{3,1})^{-1} \quad (3.24)$$

Here is a second example, where we define  $D_i^n := B_i^n + B_i^{-n}$  to emphasize the palindromic nature of the Alexander polynomial:

$$Z(K_{11a10}) = \left( \frac{1}{2D_1^3 - 11D_1^2 + 25D_1^1 - 31} \right)_{\{1\}}^{\emptyset} = \Delta(K_{11a10})^{-1} \quad (3.25)$$

Since each tangle is expressed as an object, the domains in these examples are empty.

# CONSTRUCTING THE TRACE

## 4.1 EXTENDING A PURE TANGLE INVARIANT TO LINKS AND GENERAL TANGLES

Thus far, the algebraic setting we have defined allows us to describe invariants of tangles with no closed components. We now extend the notion of a meta-Hopf algebra to include closed components.

**Definition 4.1** (traced meta-algebra). A traced meta-algebra is a family of meta-algebras: for each finite set  $L$ , we assign one meta-algebra  $\{A_{L,S}\}_S$ .<sup>†1</sup> The multiplication maps  $m_k^{i,j}[L]$  then take the form:

$$m_k^{i,j}[L][S]: A_{\{i,j\} \sqcup S, L} \rightarrow A_{\{k\} \sqcup S, L} \quad (4.1)$$

for  $i, j, k$  disjoint from both  $S$  and  $L$ .

There is an additional structure, called a trace. The compatibility of the families of meta-algebras is governed this trace in the following way:  $\text{tr}^i: A_{\{i\} \sqcup S, L} \rightarrow A_{S, \{i\} \sqcup L}$  which is universal with respect to the cyclic property:

$$m_k^{i,j} \parallel \text{tr}^k = m_k^{j,i} \parallel \text{tr}^k \quad (4.2)$$

Furthermore,  $\text{tr}^i$  is a morphism of meta-coalgebras. That is:

$$\Delta_{jk}^i \parallel \text{tr}^j \parallel \text{tr}^k = \text{tr}^i \parallel \Delta_{jk}^i \quad (4.3)$$

$$\text{tr}^i \parallel \epsilon^i = \epsilon^i \quad (4.4)$$

The first example we give is that of impure tangles.

**Definition 4.2** (Impure Upright Tangles). Let  $\mathcal{T}_{L,S}^{\text{rv}}$  be the set of upright tangles with open strands indexed by  $S$  and closed strands indexed by  $L$ .

<sup>†1</sup> These sets index the “strands”  $S$  and the “loops”  $L$ .

The operations  $\phi[L][S]$  are defined analogously to the  $\phi[S]$  given in ?? (Here  $\phi$  varies over  $m, \eta, \Delta, \epsilon, S, \mathcal{R}$ , and  $C$ .)

**Lemma 4.3** (tangles as a traced algebra). *The collection of all  $\mathcal{T}_{L,S}^{rv}$  is a traced ribbon meta-Hopf algebra, with trace map given by closing a strand into a loop.*

*Proof.* When  $L = \emptyset$ , the situation is exactly the case of theorem 2.28, so  $\mathcal{T}_{\emptyset,S}^{rv} = \mathcal{T}_S$  is a meta-Hopf algebra. Furthermore, since the Reidemeister moves are local operations, the presence of closed components does not affect our ability to verify the identities on the Hopf-algebra operations.

The last point to verify is that closing a strand into a loop is a cyclic operation. Given two strands, we must verify that stitching one end together, then tracing the other yields the same diagram as stitching the other ends together, then taking the trace. However, by definition of trace, these two actions yield identical diagrams, the two strands replaced by the same closed loop.  $\square$

**Lemma 4.4** (coinvariants as a trace map). *Let  $A$  be an algebra, and denote by  $A_A := A/[A, A]$  its set of coinvariants. Then define  $A_{S,L} := A^{\otimes S} \otimes A_A^{\otimes L}$ . Then  $A$  defines a traced meta-algebra with trace map given by  $\text{tr}_j^i: A_i \rightarrow (A_A)_j$ .*

*Proof.* Observe that for any choice of  $L$ , extending morphisms by the identity yield an isomorphism of traced meta-Hopf algebras:

$$\begin{aligned} \phi_L: \{A^{\otimes S}\}_S &\xrightarrow{\sim} \{A^{\otimes S} \otimes A_A^{\otimes L}\}_S \\ A^{\otimes S} &\mapsto A^{\otimes S} \otimes A_A^{\otimes L} \\ f &\mapsto f \otimes \text{id}_{A_A}^{\otimes L} \end{aligned} \tag{4.5}$$

Next, we must show that  $m_k^{ij} \parallel \text{tr}^k = m_k^{ji} \parallel \text{tr}^k$ . This amounts to showing that, given  $u, v \in A$ , that  $\overline{uv} = \overline{vu} \in A_A$ . However, by the construction of the coinvariants,  $\overline{uv} - \overline{vu} = \overline{uv - vu} = \overline{0} \in A$ , and we are done.  $\square$

## 4.2 THE COINVARIANTS OF $U$

We start with a result which simplifies working with coinvariants:

**Lemma 4.5** (Coinvariant simplification). *Let  $\mathfrak{h}$  be a Lie algebra. Then  $\mathfrak{U}(\mathfrak{h})_{\mathfrak{U}(\mathfrak{h})} = \mathfrak{U}(\mathfrak{h})_{\mathfrak{h}}$ .*

*Proof.* First, observe that for any  $u, v, f \in \mathfrak{U}(\mathfrak{h})$ ,  $\text{ad}_{uv}(f) = \text{ad}_u(vf) + \text{ad}_v(fu)$ . Proceeding inductively, for any monomial  $\mu \in \mathfrak{U}(\mathfrak{h})$ ,  $\text{ad}_\mu(u)$  is a linear combination of elements of  $[\mathfrak{h}, \mathfrak{U}(\mathfrak{h})]$ . By linearity of  $\text{ad}$ , we conclude  $[\mathfrak{U}(\mathfrak{h}), \mathfrak{U}(\mathfrak{h})] = [\mathfrak{h}, \mathfrak{U}(\mathfrak{h})]$ .  $\square$

**Theorem 4.6.** *The coinvariants of  $U$ ,  $U_U$ , has basis  $\{y^n a^k x^n\}_{n,k \geq 0}$ .*

*Proof.* Using lemma 4.5, we need only compute  $[\mathfrak{g}, U]$  to determine  $U_U$ . Given a polynomial  $f$ , we have the following relations in  $U$ :

$$f(a)y^r = y^r f(a-r) \quad x^r f(a) = f(a-r)x^r \quad (4.6)$$

Next we compute the adjoint actions of  $y$ ,  $a$ , and  $x$ . (Recall  $b$  is central.)

$$\text{ad}_a f(x) = x f'(x) \quad \text{ad}_a f(y) = -y f'(y) \quad (4.7)$$

$$\text{ad}_x f(y) = b f'(y) \quad \text{ad}_x f(a) = -\nabla[f](a)x \quad (4.8)$$

$$\text{ad}_y f(x) = -b f'(x) \quad \text{ad}_y f(a) = y \nabla[f](a) \quad (4.9)$$

(Here  $\nabla$  is the backwards finite difference operator  $\nabla[f](x) := f(x) - f(x-1)$ .)

Observe for any  $n, m, k$ , and polynomials  $f$  and  $g$ :

$$\text{ad}_a(y^m g(b, a)x^n) = (n-m)y^m g(b, a)x^n \quad (4.10)$$

$$\text{ad}_x(y^{n+1}b^{m-1}f(a)x^k) = (n+1)y^n b^m f(a)x^k - y^{n+1}b^{m-1}\nabla[f](a)x^{k+1} \quad (4.11)$$

$$\text{ad}_y(y^n b^{m-1}f(a)x^{k+1}) = -(k+1)y^n b^m f(a)x^k + y^{n+1}b^{m-1}\nabla[f](a)x^{k+1} \quad (4.12)$$

By equation (4.10), any monomial whose powers of  $y$  and  $x$  differ vanish in  $U_{\mathfrak{g}}$ . As a consequence, in equations (4.11) and (4.12), the only nontrivial case is when  $n = k$ , resulting in the same relation. By induction on  $n$ , we conclude that:

$$y^n b^m f(a)x^k \sim \delta_{nk} \frac{n!}{(n+m)!} y^{n+m} \nabla^m[f](a)x^{n+m} \quad (4.13)$$

where  $\sim$  refers to equivalence in the set of coinvariants. Observing when  $f$  is a monomial in equation (4.13), we see  $U_{\mathfrak{g}}$  is spanned by  $\{y^n a^k x^n\}_{n,k \geq 0}$ .

Finally, all that remains to show is this set is linearly independent. This is equivalent to no two  $y^n a^k x^n$ 's with distinct choices of exponents being related under a sequence of relations. Since lemma 4.5 allows us to only consider sequences of relations of the form  $\text{ad}_z$  for  $z$  a one-letter word in  $\{y, b, a, x\}$ , inspection of the above comprehensive summary of all one-letter relations (in particular, equation (4.13) with  $m = 0$ ) allows us to conclude that this set is indeed linearly independent.  $\square$

### *A generating function for the coinvariants*

In order to define a generating function, we need to choose an appropriate basis for the space of coinvariants. We define an isomorphism from the space of coinvariants to a polynomial space, tweaking the earlier-defined basis by scalar multiples. Since it plays the role of the ordering map, we also name it  $\mathbb{O}$ .

$$\begin{aligned} \mathbb{O}: \mathbb{Q}[a, z] &\xrightarrow{\sim} U_U \\ a^n z^k &\mapsto \frac{1}{k!} y^k a^n x^k \\ k! \nabla^m[f](a) z^{k+m} &\mapsto y^k b^m f(a) x^k \end{aligned} \quad (4.14)$$

This defines a commutative square upon whose bottom edge  $\tau = \mathbb{O} \parallel \text{tr} \parallel \mathbb{O}^{-1}$  we compute the generating function:

$$\begin{array}{ccc} U & \xrightarrow{\text{tr}} & U_U \\ \mathbb{O} \uparrow & & \uparrow \mathbb{O} \\ \mathbb{Q}[y, b, a, x] & \xrightarrow{\tau} & \mathbb{Q}[a, z] \end{array} \quad (4.15)$$

We begin with a result on finite differences:

**Lemma 4.7** (finite differences of exponentials). *The finite difference operator acts in the following way on exponentials:*

$$\nabla^n[\mathbf{e}^{\alpha a}](a) = (1 - \mathbf{e}^{-\alpha})^n \mathbf{e}^{\alpha a} \quad (4.16)$$

*Proof.* Using the fact that  $\nabla^n[f](x) = \sum_{k=0}^n \binom{n}{k} (-1)^k f(x - k)$ , we see that  $\nabla^n[\mathbf{e}^{\alpha a}](a) = \sum_{k=0}^n \binom{n}{k} (-1)^k \mathbf{e}^{\alpha a - \alpha k} = (1 - \mathbf{e}^{-\alpha})^n \mathbf{e}^{\alpha a}$ .  $\square$

We now are ready to compute the generating function for the trace:

**Theorem 4.8** (Generating function for the trace of  $U$ ).

$$\mathcal{G}(\text{tr}) = \exp\left(\alpha a + (\eta\xi + \beta(1 - e^{-\alpha}))z\right) \quad (4.17)$$

*Proof.* Using lemma 4.7 and the extension of scalars of  $\text{tr}$  to  $\mathbb{Q}[\eta, \beta, \alpha, \xi]$ , we see

$$\begin{aligned} \mathcal{G}(\mathbb{O} \parallel \text{tr} \parallel \mathbb{O}^{-1}) &= (e^{\eta y} e^{\beta b} e^{\alpha a} e^{\xi x}) \parallel \text{tr} \parallel \mathbb{O}^{-1} \\ &= \mathbb{O}^{-1} \sum_{i,j,k} \text{tr} \left( \frac{(\eta y)^i}{i!} \frac{(\beta b)^j}{j!} e^{\alpha a} \frac{(\xi x)^k}{k!} \right) \\ &= \sum_{i,j} \frac{\eta^i \beta^j \xi^i}{i! j!} (1 - e^{-\alpha})^j e^{\alpha a} z^{i+j} = e^{\alpha a + (\eta\xi + \beta(1 - e^{-\alpha}))z} \quad \square \end{aligned} \quad (4.18)$$

*Evaluation of the trace on a generic element*

Here we will outline a computation involving the trace by using Bar-Natan and van der Veen's Contraction Theorem.

A typical value for a tangle invariant that arises is of the form:

$$P e^{c + \alpha a_i + \beta b_i + \xi(b_i)x_i + \eta(b_i)y_i + \lambda(b_i)x_i y_i} \quad (4.19)$$

Here,  $c$ ,  $\alpha$ , and  $\beta$  denote constants with respect to the variables  $y_i$ ,  $b_i$ ,  $a_i$ , and  $x_i$  (collectively referred to as " $v_i$ "s), while  $\xi$ ,  $\eta$ , and  $\lambda$  are potentially  $b_i$ -dependent, and  $P$  is a (rational) function in (the square root of)  $B_i$ .

**Theorem 4.9** (The trace of a Gaussian). *With symbols as defined above, let  $f(y_i, b_i, a_i, x_i) = P(B_i) e^{c + \alpha a_i + \beta b_i + \xi(b_i)x_i + \eta(b_i)y_i + \lambda(b_i)x_i y_i}$ . Then*

$$\langle f(y_i, b_i, a_i, x_i) \text{tr}^i \rangle_{v_i} = \frac{P(e^{-\mu})}{1 - \lambda(\mu)\bar{z}_i} e^{c + \alpha \bar{a}_i + \beta \mu + \frac{\eta(\mu)\xi(\mu)\bar{z}_i}{1 - \lambda(\mu)\bar{z}_i}} \quad (4.20)$$

where  $\mu := (1 - e^{-\alpha})\bar{z}_i$ .

*Proof.* Let us compute the trace of equation (4.19). For clarity, we will put bars over the coinvariants variables  $a_i$  and  $z_i$ , as they do not play a role in the contraction.

$$\begin{aligned} & \langle P(B_i) \mathbf{e}^{c+\alpha a_i+\beta b_i+\xi(b_i)x_i+\eta(b_i)y_i+\lambda(b_i)x_i y_i} \text{tr}^i \rangle_{v_i} \\ &= \langle P(B_i) \mathbf{e}^{c+\beta b_i+\xi(b_i)x_i+\eta(b_i)y_i+\lambda(b_i)x_i y_i+\eta_i \xi_i \bar{z}_i+\beta_i(1-\mathbf{e}^{-\alpha})\bar{z}_i} \mathbf{e}^{\alpha a_i+\alpha_i \bar{a}_i} \rangle_{v_i} \\ &= \mathbf{e}^{\alpha \bar{a}_i} \langle P(B_i) \mathbf{e}^{c+\xi(b_i)x_i+\eta(b_i)y_i+\lambda(b_i)x_i y_i+\eta_i \xi_i \bar{z}_i} \mathbf{e}^{\beta b_i+\beta_i(1-\mathbf{e}^{-\alpha})\bar{z}_i} \rangle_{b_i, x_i, y_i} \end{aligned}$$

$$\begin{aligned} & \text{In what follows, we let } \mu := (1 - \mathbf{e}^{-\alpha})\bar{z}_i: \\ &= \mathbf{e}^{c+\alpha \bar{a}_i+\beta \mu} P(\mathbf{e}^{-\mu}) \langle \mathbf{e}^{\eta(\mu)y_i} \mathbf{e}^{(\xi(\mu)+\lambda(\mu)y_i)x_i+\xi_i \eta_i \bar{z}_i} \rangle_{x_i, y_i} \\ &= \mathbf{e}^{c+\alpha \bar{a}_i+\beta \mu} P(\mathbf{e}^{-\mu}) \langle \mathbf{e}^{\eta(\mu)y_i+\xi(\mu)\bar{z}_i \eta_i+\lambda(\mu)\bar{z}_i \eta_i y_i} \rangle_{y_i} \\ &= \frac{P(\mathbf{e}^{-\mu})}{1 - \lambda(\mu)\bar{z}_i} \mathbf{e}^{c+\alpha \bar{a}_i+\beta \mu+\frac{\eta(\mu)\xi(\mu)\bar{z}_i}{1-\lambda(\mu)\bar{z}_i}} \end{aligned} \tag{4.21}$$

□

We point out that the outcome of this computation is not guaranteed to be a Gaussian. This puts a limitation on the applicability of this formula to links with more than two components, explored in chapter 5.

### Computational examples

Using the formula given in equation (4.20), let us do some preliminary examples:

$$\text{tr}^i(R_{ij}) = 1 \tag{4.22}$$

$$\text{tr}^j(R_{ij}) = \mathbf{e}^{b_i \bar{a}_j} \tag{4.23}$$

$$\begin{aligned} & \text{tr}^2 \left( \sqrt{B_2} \mathbf{e}^{-a_2 b_1 - a_1 b_2 + \frac{(B_1-1)x_2 y_1}{b_1 B_1} + \frac{(B_2-1)x_1 y_2}{b_2 B_2}} \right) = \\ & \mathbf{e}^{\frac{a_1(\bar{z}_2 - B_1 \bar{z}_2)}{B_1} - b_1 \bar{a}_2 + \frac{e^{B_1 \bar{z}_2} (x_1 y_1 e^{B_1^{-1} \bar{z}_2 - x_1 y_1 e^{\bar{z}_2}})}{b_1} + \frac{1}{2} B_1^{-1} \bar{z}_2 - \frac{\bar{z}_2^2}{2}} \end{aligned} \tag{4.24}$$

Equations (4.22) and (4.23) are the values one obtains for the two (virtual) one-crossing, two-component link, while equation (4.24) is the value of the invariant on the Hopf link.

When computing this on a link, however, it is important to keep track of which strands are open, and which are closed. We will extend the notation

from the previous section to differentiate between open and closed indices. We write a morphism with domain  $D = D_o \sqcup D_c$ , codomain  $C = C_o \sqcup C_c$  (here  $D_o$  denotes domain indices which are open, while  $D_c$  those which are closed, with the same convention for  $C$ ) and generating function  $f(\zeta_D, z_C)$  as  $f(\zeta_D, z_C)_{(C_o, C_c)}^{(D_o, D_c)}$ .



## CONCLUSIONS

---

### *Limitations of this definition*

For some inputs to the trace, expressions involving the Lambert  $W$ -function appear, which complicates attempts to keep the invariant valued in the space of perturbed Gaussians.

### 5.1 COMPARISON WITH THE MULTIVARIABLE ALEXANDER POLYNOMIAL

Given that the long knot (i.e. one-component) case of this invariant encodes the Alexander Polynomial, it was suspected that the invariant on long links (i.e. multiple components, one of which is long) formed by adding the trace would encode the [MVA](#). However, there are links which the [MVA](#) separates which this invariant does not.

On all two-component links with at most 11 crossings (a collection of size 914), the trace map attains 878 distinct values, while the MVA attains only 778. However, the two invariants are incomparable in terms of their strength.

The links  $L_{5a1}$  and  $L_{10a43}$  are not distinguished by their partial traces, with both returning a value of:

$$\left( \left( \frac{B_1}{B_1^2 t_2 - 2B_1 t_2 + B_1 + t_2} \right)_{(\{1\}, \{2\})}, \left( \frac{B_2^{3/2}}{B_2^2 t_1 - 2B_2 t_1 + B_2 + t_1} \right)_{(\{2\}, \{1\})} \right) \quad (5.1)$$

The values of these links under the [MVA](#) are, however

$$\frac{(B_1 - 1)(B_2 - 1)}{\sqrt{B_1}\sqrt{B_2}} \text{ and } -\frac{(B_1 - 1)(B_2 - 1)(B_1 + B_2 - 1)(B_2 B_1 - B_1 - B_2)}{B_1^{3/2} B_2^{3/2}} \quad (5.2)$$

respectively.

In the other direction, there are also pairs of links in the same fibre of the [MVA](#) which this traced invariant can distinguish. In particular  $L_{5a1}$  and  $L_{7n2}$  both have the same value under the [MVA](#):

$$\frac{(B_1 - 1)(B_2 - 1)}{\sqrt{B_1}\sqrt{B_2}} \quad (5.3)$$

The trace yields the following values (respectively):

$$\left( \left( \frac{B_1}{B_1^2 t_2 - 2B_1 t_2 + B_1 + t_2} \right)_{(\{1\}, \{2\})}, \left( \frac{B_2^{3/2}}{B_2^2 t_1 - 2B_2 t_1 + B_2 + t_1} \right)_{(\{2\}, \{1\})} \right) \quad (5.4)$$

$$\left( \left( \frac{B_1}{B_1^2 t_2 - 2B_1 t_2 + B_1 + t_2} \right)_{(\{1\}, \{2\})}, \left( \frac{B_2^{5/2}}{B_2^2 t_1 - 2B_2 t_1 + B_2^2 - B_2 + t_1 + 1} \right)_{(\{2\}, \{1\})} \right) \quad (5.5)$$

This example also serves to highlight that the information provided by leaving one strand open is not enough to recover the value of a different strand being left open.

## 5.2 FURTHER WORK

While all other Hopf algebra operations in  $U$  are expressed by [\[BNvdV\]](#) as perturbed Gaussians, the form in equation [\(4.17\)](#) does not conform to the same structure. Further work is needed to either implement this operation into the established framework, or to suitably extend the framework (perhaps with the use of Lambert  $W$ -functions).

## CODE

---

### A.1 IMPLEMENTATION OF THE INVARIANT $Z$

This is a Mathematica™ implementation by Bar-Natan and van der Veen in [BNvdV], modified by the author. The full source code is available at <https://github.com/phro/GD0>. We begin by setting some variables, as well as a method for modifying associations.

```

1  γ = 1; ħ = 1; $k = 0;
2  setValue[value_,obj_,coord_] := Module[
3      {b=Association@@obj},
4      b[coord] = value; Head[obj]@@Normal@b
5  ]

```

We introduce notation  $\text{PG}[L, Q, P]$  to be interpreted as the Perturbed Gaussian  $Pe^{L+Q}$ . The function `fromE` serves as a compatibility layer between a former version of the code.

```

6  toPG[L_, Q_, P_] := PG["L" -> L, "Q" -> Q, "P" -> P]
7  fromE[e_\[DoubleStruckCapitalE]] := toPG@@e/.
8      Subscript[(v:y|b|t|a|x|B|T|η|β|τ|α|ξ|A), i_] -> v[i]

```

We define the Kronecker- $\delta$  function next.

```

9  δ[i_, j_] := If[SameQ[i, j], 1, 0]

```

Next we introduce helper functions for the reading and manipulating of PG-objects:

```

10 getL[pg_PG] := Lookup[Association@@pg, "L", 0]
11 getQ[pg_PG] := Lookup[Association@@pg, "Q", 0]
12 getP[pg_PG] := Lookup[Association@@pg, "P", 1]
13
14 setL[L_][pg_PG] := setValue[L, pg, "L"];

```

```

15 setQ[Q_][pg_PG] := setValue[Q, pg, "Q"];
16 setP[P_][pg_PG] := setValue[P, pg, "P"];
17
18 applyToL[f_][pg_PG] := pg//setL[pg//getL//f]
19 applyToQ[f_][pg_PG] := pg//setQ[pg//getQ//f]
20 applyToP[f_][pg_PG] := pg//setP[pg//getP//f]

```

Next is a function CF, which bring objects into canonical form allows us to compare for equality effectively. This is defined by Bar-Natan and van der Veen.

```

21 CCF[e_] := ExpandDenominator@ExpandNumerator@Together[
22     Expand[e] //. E^x_ E^y_ :> E^(x + y) /. E^x_ :>
23     ↪ E^CCF[x]
24 ];
25 CF[sd_SeriesData] := MapAt[CF, sd, 3];
26 CF[e_] := Module[
27     {vs = Union[
28         Cases[e, (y|b|t|a|x|η|β|τ|α|ξ)[_], ∞],
29         {y, b, t, a, x, η, β, τ, α, ξ}
30     ]},
31     Total[CoefficientRules[Expand[e], vs] /.
32         (ps_ -> c_) :> CCF[c] (Times @@ (vs^ps))
33     ];
34 CF[e_PG] := e//applyToL[CF]//applyToQ[CF]//applyToP[CF]

```

We must also define the notion of equality for PG-objects, as well as what it means to multiply them.

```

35 Congruent[x_, y_, z_] := And[Congruent[x, y], Congruent[y, z]]
36 PG /: Congruent[pg1_PG, pg2_PG] := And[
37     CF[getL@pg1 == getL@pg2],
38     CF[getQ@pg1 == getQ@pg2],
39     CF[Normal[getP@pg1 - getP@pg2] == 0]
40 ];
41
42 PG /: pg1_PG * pg2_PG := toPG[
43     getL@pg1 + getL@pg2,

```

```

44      getQ@pg1 + getQ@pg2,
45      getP@pg1 * getP@pg2
46  ]
47
48  setEpsilonDegree[k_Integer][pg_PG] :=
    ↪ setP[Series[Normal@getP@pg,{ϵ, 0, k}]] [pg]

```

The variables  $y, b, t, a$ , and  $x$  are paired with their dual variables  $\eta, \beta, \tau, \alpha$ , and  $\xi$ . This applies as well when they have subscripts.

```

49  dds12vars = {y, b, t, a, x, z};
50  dds12varsDual = {η, β, τ, α, ξ, ζ};
51
52  Evaluate[Dual/@dds12vars] = dds12varsDual;
53  Evaluate[Dual/@dds12varsDual] = dds12vars;
54  Dual@z = ζ;
55  Dual@ζ = z;
56  Dual[u_[i_]] := Dual[u][i]

```

Since various exponentials of the lowercase variables frequently appear, we introduce capital variable names to handle various exponentiated forms.

```

57  U2l = {
58      B[i_]^p_. := E^(-p ħ γ b[i]), B^p_. := E^(-p ħ γ b),
59      W[i_]^p_. := E^(w[i])      , W^p_. := E^(p w),
60      T[i_]^p_. := E^(-p ħ t[i]) , T^p_. := E^(-p ħ t),
61      A[i_]^p_. := E^(p γ α[i])   , A^p_. := E^(p γ α)
62  };
63  l2U = {
64      E^(c_. b[i_] + d_.) := B[i]^(-c/(ħ γ))E^d,
65      E^(c_. b + d_.)    := B^(-c/(ħ γ))E^d,
66      E^(c_. t[i_] + d_.) := T[i]^(-c/ħ)E^d,
67      E^(c_. t + d_.)     := T^(-c/ħ)E^d,
68      E^(c_. α[i_] + d_.) := A[i]^(c/γ)E^d,
69      E^(c_. α + d_.)     := A^(c/γ)E^d,
70      E^(c_. w[i_] + d_.) := W[i]^(c)E^d,
71      E^(c_. w + d_.)     := W^(c)E^d,
72      E^expr_             := E^Expand@expr
73  };

```

Below the notion of differentiation is defined for expressions which involve both upper- and lower-case variables.

```

74 DD[f_, b_] := D[f, b] - h γ B D[f, B];
75 DD[f_, b[i_]] := D[f, b[i]] - h γ B[i] D[f, B[i]];
76
77 DD[f_, t_] := D[f, t] - h T D[f, T];
78 DD[f_, t[i_]] := D[f, t[i]] - h T[i] D[f, T[i]];
79
80 DD[f_, α_] := D[f, α] + γ A D[f, A];
81 DD[f_, α[i_]] := D[f, α[i]] + γ A[i] D[f, A[i]];
82
83 DD[f_, v_] := D[f, v];
84 DD[f_, {v_, 0}] := f;
85 DD[f_, {}] := f;
86 DD[f_, {v_, n_Integer}] := DD[DD[f, v], {v, n-1}];
87 DD[f_, {l_List, ls___}] := DD[DD[f, l], {ls}];

```

What follows now is the implementation of contraction as introduced in definition 3.4. We begin with the introduction of contractions of (finite) polynomials.

```

88 collect[sd_SeriesData, ζ_] := MapAt[collect[#, ζ] &, sd, 3];
89 collect[expr_, ζ_] := Collect[expr, ζ];
90
91 Zip[{}][P_] := P;
92 Zip[ζs_List][Ps_List] := Zip[ζs]/@Ps;
93 Zip[{ζ_, ζs___}][P_] := (collect[P // Zip[{ζs}], ζ] /.
94     f_. ζ^d_. :> DD[f, {Dual[ζ], d}]) /.
95     Dual[ζ] -> 0 /.
96     ((Dual[ζ] /. {b->B, t->T, α -> A}) -> 1)

```

We define contraction along the variables  $x$  and  $y$  (here packaged into the matrix  $Q$ ).

```

97 QZip[ζs_List][pg_PG] := Module[{Q, P, ζ, z, zs, c, ys, ηs, qt,
    ↪ zrule, ζrule},
98     zs = Dual/@ζs;
99     Q = pg//getQ;

```

```

100     P = pg//getP;
101     c = CF[Q/.Alternatives@@Union[ $\zeta$ s, zs]->0];
102     ys = CF/@Table[D[Q, $\zeta$ ]/.Alternatives@@zs->0,{ $\zeta$ , $\zeta$ s}];
103      $\eta$ s = CF/@Table[D[Q,z]/.Alternatives@@ $\zeta$ s->0,{z,zs}];
104     qt = CF/@#&/@Inverse@Table[
105          $\delta$ [z, Dual[ $\zeta$ ]] - D[Q,z, $\zeta$ ],
106         { $\zeta$ , $\zeta$ s},{z,zs}
107     ];
108     zrule = Thread[zs -> CF/@(qt . (zs + ys))];
109      $\zeta$ rule = Thread[ $\zeta$ s ->  $\zeta$ s +  $\eta$ s . qt];
110     CF@setQ[c +  $\eta$ s.qt.ys]@setP[Det[qt] Zip[ $\zeta$ s][P /.
        ↪ Union[zrule,  $\zeta$ rule]]]@pg
111 ]

```

We define contraction along the variables  $a$  and  $b$  (here packaged into the matrix  $L$ ).

```

112 LZip[ $\zeta$ s_List][pg_PG] := Module[
113     {
114         L, Q, P,  $\zeta$ , z, zs, Zs, c, ys,  $\eta$ s, lt,
115         zrule, Zrule,  $\zeta$ rule, Q1, EEQ, EQ, U
116     },
117     zs = Dual/@ $\zeta$ s;
118     {L, Q, P} = Through[{getL, getQ, getP}@pg];
119     Zs = zs /. {b -> B, t -> T,  $\alpha$  -> A};
120     c = CF[L/.Alternatives@@Union[ $\zeta$ s,
        ↪ zs]->0/.Alternatives@@Zs -> 1];
121     ys = CF/@Table[D[L, $\zeta$ ]/.Alternatives@@zs->0,{ $\zeta$ , $\zeta$ s}];
122      $\eta$ s = CF/@Table[D[L,z]/.Alternatives@@ $\zeta$ s->0,{z,zs}];
123     lt = CF/@#&/@Inverse@Table[
124          $\delta$ [z, Dual[ $\zeta$ ]] - D[L,z, $\zeta$ ],
125         { $\zeta$ , $\zeta$ s},{z,zs}
126     ];
127     zrule = Thread[zs -> CF/@(lt . (zs + ys))];
128     Zrule = Join[zrule, zrule /.
129         r_Rule := ( (U = r[[1]] /. {b -> B, t -> T,  $\alpha$ 
        ↪ -> A}) ->
130         (U /. U2l /. r //. l2U))

```

```

131 ];
132 \[Zeta]rule = Thread[\[Zeta]s -> \[Zeta]s + \[Eta]s .
      ↪ lt];
133 Q1 = Q /. Union[Zrule, ζrule];
134 EEQ[ps___] :=
135     EEQ[ps] = (
136         CF[E^-Q1 DD[E^Q1, Thread[{zs, {ps}}]]] /.
137         {Alternatives@@zs -> 0,
          ↪ Alternatives @@Zs -> 1}]
138     );
139 CF@toPG[
140     c + ηs.lt.ys,
141     Q1 /. {Alternatives@@zs -> 0, Alternatives@@Zs
          ↪ -> 1},
142     Det[lt] (Zip[ζs][[EQ@@zs] (P /.
          ↪ Union[Zrule, ζrule])]] /.
143         Derivative[ps___][EQ][___] := EEQ[ps] /.
          ↪ _EQ -> 1
144     )
145 ]
146
147 ]

```

The function `Pair` combines the above zipping functions into the final contraction map.

```

148 Pair[{}][L_PG, R_PG] := L R;
149 Pair[is_List][L_PG, R_PG] := Module[{n},
150     Times[
151         L /. ((v: b|B|t|T|a|x|y)[#] -> v[n@#]&/@is),
152         R /. ((v: β|τ|α|A|ξ|η)[#] -> v[n@#]&/@is)
153     ] // LZip[Join@@Table[Through[{β, τ, a}[n@i]], {i, is}]]
          ↪ //
154     QZip[Join@@Table[Through[{ξ, y}[n@i]], {i, is}]]
155 ]

```

Our next task is to provide domain and codomain information for the PG-objects. These will be packaged inside a GDO, (Gaußian Differential Operator).



The four lists' names refer to whether it is a domain or a codomain, and whether the index corresponds to an open strand or a closed one.

```

156 toGDO[do_List,dc_List,co_List,cc_List,L_,Q_,P_] := GDO[
157     "do" -> do,
158     "dc" -> dc,
159     "co" -> co,
160     "cc" -> cc,
161     "PG" -> toPG[L, Q, P]
162 ]
163
164 toGDO[do_List,dc_List,co_List,cc_List,pg_PG] := GDO[
165     "do" -> do,
166     "dc" -> dc,
167     "co" -> co,
168     "cc" -> cc,
169     "PG" -> pg
170 ]

```

Next are defined functions for accessing and modifying sub-parts of GDO-objects. The last argument of Lookup is the default value if nothing is specified. This means that a morphism with empty domain or codomain may be specified as such by omitting that portion of the definition.

```

171 getDO[gdo_GDO] := Lookup[Association@@gdo, "do", {}]
172 getDC[gdo_GDO] := Lookup[Association@@gdo, "dc", {}]
173 getCO[gdo_GDO] := Lookup[Association@@gdo, "co", {}]
174 getCC[gdo_GDO] := Lookup[Association@@gdo, "cc", {}]
175
176 getPG[gdo_GDO] := Lookup[Association@@gdo, "PG", PG[]]
177
178 getL[gdo_GDO] := gdo//getPG//getL
179 getQ[gdo_GDO] := gdo//getPG//getQ
180 getP[gdo_GDO] := gdo//getPG//getP
181
182 setPG[pg_PG][gdo_GDO] := setValue[pg, gdo, "PG"]
183
184 setL[L_][gdo_GDO] := setValue[setL[L][gdo//getPG], gdo, "PG"]

```

```

185 setQ[Q_][gdo_GD0] := setValue[setQ[Q][gdo//getPG], gdo, "PG"]
186 setP[P_][gdo_GD0] := setValue[setP[P][gdo//getPG], gdo, "PG"]
187
188 setD0[do_][gdo_GD0] := setValue[do, gdo, "do"]
189 setDC[dc_][gdo_GD0] := setValue[dc, gdo, "dc"]
190 setC0[co_][gdo_GD0] := setValue[co, gdo, "co"]
191 setCC[cc_][gdo_GD0] := setValue[cc, gdo, "cc"]
192
193 applyToD0[f_][gdo_GD0] := gdo//setD0[gdo//getD0//f]
194 applyToDC[f_][gdo_GD0] := gdo//setDC[gdo//getDC//f]
195 applyToC0[f_][gdo_GD0] := gdo//setC0[gdo//getC0//f]
196 applyToCC[f_][gdo_GD0] := gdo//setCC[gdo//getCC//f]
197
198 applyToPG[f_][gdo_GD0] := gdo//setPG[gdo//getPG//f]
199
200 applyToL[f_][gdo_GD0] := gdo//setL[gdo//getL//f]
201 applyToQ[f_][gdo_GD0] := gdo//setQ[gdo//getQ//f]
202 applyToP[f_][gdo_GD0] := gdo//setP[gdo//getP//f]

```

The canonical form function (CF) and the contraction mapping (Pair) we extend to include GD0-objects. Furthermore, on the level of GD0-objects we can compose morphisms and keep track of the corresponding domains and codomains, using the left-to-right composition operator “//”.

```

203 CF[e_GD0] := e//
204     applyToD0[Union]//
205     applyToDC[Union]//
206     applyToC0[Union]//
207     applyToCC[Union]//
208     applyToPG[CF]
209
210 Pair[is_List][gdo1_GD0, gdo2_GD0] := GD0[
211     "do" -> Union[gdo1//getD0, Complement[gdo2//getD0,
212         ↪ is]],
213     "dc" -> Union[gdo1//getDC, gdo2//getDC],
214     "co" -> Union[gdo2//getC0, Complement[gdo1//getC0,
215         ↪ is]],
216     "cc" -> Union[gdo1//getCC, gdo2//getCC],

```

```

215         "PG" -> Pair[is][gdo1//getPG, gdo2//getPG]
216     ]
217
218     gdo1_GD0 // gdo2_GD0 :=
        ⇨ Pair[Intersection[gdo1//getC0,gdo2//getD0]][gdo1,gdo2];

```

We also define notions of equality and multiplication (by concatenation) for GD0's.

```

219 GD0 /: Congruent[gdo1_GD0, gdo2_GD0] := And[
220     Sort@*getD0/@Equal[gdo1, gdo2],
221     Sort@*getDC/@Equal[gdo1, gdo2],
222     Sort@*getC0/@Equal[gdo1, gdo2],
223     Sort@*getCC/@Equal[gdo1, gdo2],
224     Congruent[gdo1//getPG, gdo2//getPG]
225 ]
226
227 GD0 /: gdo1_GD0 gdo2_GD0 := GD0[
228     "do" -> Union[gdo1//getD0, gdo2//getD0],
229     "dc" -> Union[gdo1//getDC, gdo2//getDC],
230     "co" -> Union[gdo1//getC0, gdo2//getC0],
231     "cc" -> Union[gdo1//getCC, gdo2//getCC],
232     "PG" -> (gdo1//getPG)*(gdo2//getPG)
233 ]

```

For the sake of compatibility with Bar-Natan and van der Veen's program, we introduce several conversion functions between the two notations.

```

234 setEpsilonDegree[k_Integer][gdo_GD0] :=
235     setP[Series[Normal@getP@gdo,{ϵ,0,k}]] [gdo]
236
237 fromE[Subscript[\[DoubleStruckCapitalE],{do_List,
        ⇨ dc_List}->{co_List, cc_List}][
238     L_, Q_, P_
239 ]] := toGD0[do, dc, co, cc, fromE[\[DoubleStruckCapitalE][L, Q,
        ⇨ P]]]
240
241 fromE[Subscript[\[DoubleStruckCapitalE], dom_List->cod_List][
242     L_, Q_, P_

```

```

243 ]] := GD0["do" -> dom, "co" -> cod,
244         "PG" -> fromE\[DoubleStruckCapitalE][L, Q, P]]
245 ]

```

It is at this point that we implement the morphisms of the algebra  $U$ . Each operation is prepended with a “c” to emphasize that this is a classical algebra, not a quantum deformation.

```

246 fromLog[l_] := CF@Module[
247     {L, l0 = Limit[l, ε->0]},
248     L = l0 /. (η|y|ξ|x)[_]->0;
249     PG[
250         "L" -> L,
251         "Q" -> l0 - L
252     ]/.l2U
253 ]
254
255 cΛ = (η[i] + E^(-γ α[i] - ε β[i]) η[j]/(1+γ ε η[j]ξ[i])) y[k] +
256       (β[i] + β[j] + Log[1 + γ ε η[j]ξ[i]]/ε) b[k] +
257       (α[i] + α[j] + Log[1 + γ ε η[j]ξ[i]]/γ) a[k] +
258       (ξ[j] + E^(-γ α[j] - ε β[j]) ξ[i]/(1+γ ε η[j]ξ[i])) x[k];
259
260 cm[i_, j_, k_] = GD0["do" -> {i,j}, "co" -> {k}, "PG" ->
    ↪ fromLog[cΛ]];
261
262 cη[i_] = GD0["co" -> {i}];
263 cσ[i_, j_] = GD0["do"->{i}, "co"->{j},
264     "PG"->fromLog[β[i] b[j] + α[i] a[j] + η[i] y[j] + ξ[i]
    ↪ x[j]]
265 ];
266 cε[i_] = GD0["do" -> {i}];
267 cΔ[i_, j_, k_] = GD0["do"->{i}, "co"->{j, k},
268     "PG" -> fromLog[
269         β[i](b[j] + b[k]) +
270         α[i](a[j] + a[k]) +
271         η[i](y[j] + y[k]) +
272         ξ[i](x[j] + x[k])
273     ]

```

```

274 ];
275
276 sY[i_, j_, k_, l_, m_] = GD0["do" -> {i}, "co" -> {j, k, l, m},
277   "PG" -> fromLog[ $\beta[i]b[k] + \alpha[i]a[l] + \eta[i]y[j] +$ 
278      $\hookrightarrow \xi[i]x[m]$ ]
279 ];
280 sS[i_] = GD0["do" -> {i}, "co" -> {i},
281   "PG" -> fromLog[ $-(\beta[i] b[i] + \alpha[i] a[i] + \eta[i] y[i] +$ 
282      $\hookrightarrow \xi[i] x[i])$ ]
283 ];
284 cS[i_] = sS[i] // sY[i, 1, 2, 3, 4] // cm[4, 3, i] // cm[i, 2,
285    $\hookrightarrow i$ ] // cm[i, 1, i];
286 cR[i_, j_] = GD0[
287   "co" -> {i, j},
288   "PG" -> toPG[ $\hbar a[j] b[i], (B[i]-1)/(-b[i]) x[j] y[i],$ 
289      $\hookrightarrow 1$ ]
290 ];
291 cRi[i_, j_] = GD0[
292   "co" -> {i, j},
293   "PG" -> toPG[ $\hbar a[j] b[i], (B[i]-1)/(B[i] b[i]) x[j]$ 
294      $\hookrightarrow y[i], 1$ ]
295 ];
296 CC[i_] := GD0["co" -> {i}, "PG" -> PG["P" ->  $B[i]^{(1/2)}$ ]]
297 CCi[i_] := GD0["co" -> {i}, "PG" -> PG["P" ->  $B[i]^{(-1/2)}$ ]]
298
299 cKink[i_] = Module[{k}, cR[i, k] CCi[k] // cm[i, k, i]]
300 cKinki[i_] = Module[{k}, cRi[i, k] CC[k] // cm[i, k, i]]
301
302 cKinkn[0][i_] = c $\eta$ [i]
303 cKinkn[1][i_] = cKink[i]
304 cKinkn[-1][i_] = cKinki[i]

```

```

305 cKinkn[n_Integer][i_] :=
    ↳ Module[{j}, cKinkn[n-1][i] cKink[j] // cm[i, j, i] //; n > 1
306 cKinkn[n_Integer][i_] :=
    ↳ Module[{j}, cKinkn[n+1][i] cKinki[j] // cm[i, j, i] //; n < -1
307
308 uR[i_, j_] = Module[{k}, cR[i, j] cKinki[k] // cm[i, k, i]]
309 uRi[i_, j_] = Module[{k}, cRi[i, j] cKink[k] // cm[i, k, i]]

```

## A.2 IMPLEMENTATION OF THE TRACE

Now we implement the trace. We introduce several functions which extract the various coefficients of a GDO, so that we may apply equation (4.20). Coefficients are extracted based on whether they belong to the matrix L or the matrix Q.

```

310 getConstLCoef::usage = "getConstLCoef[i][gdo] returns the terms
    ↳ in the L-portion of a GDO expression which are not a
    ↳ function of y[i], b[i], a[i], nor x[i]."
311 getConstLCoef[i_][gdo_] :=
312     (SeriesCoefficient[#, {b[i], 0, 0}] &) @*
313     (Coefficient[#, y[i], 0] &) @*
314     (Coefficient[#, a[i], 0] &) @*
315     (Coefficient[#, x[i], 0] &) @*
316     ReplaceAll[U2l] @*
317     getL@
318     gdo
319
320 getConstQCoef::usage = "getConstQCoef[i][gdo] returns the terms
    ↳ in the Q-portion of a GDO expression which are not a
    ↳ function of y[i], b[i], a[i], nor x[i]."
321 getConstQCoef[i_][gdo_][bb_] :=
322     ReplaceAll[{b[i] -> bb}] @*
323     (Coefficient[#, y[i], 0] &) @*
324     (Coefficient[#, a[i], 0] &) @*
325     (Coefficient[#, x[i], 0] &) @*
326     ReplaceAll[U2l] @*
327     getQ@

```

```

328         gdo
329
330     getyCoef::usage = "getyCoef[i][gdo][b[i]] returns the linear
        ↪ coefficient of y[i] as a function of b[i]."
331     getyCoef[i_][gdo_][bb_] :=
332         ReplaceAll[{b[i]->bb}] @*
333         ReplaceAll[U2l] @*
334         (Coefficient[#, x[i],0]&) @*
335         (Coefficient[#, y[i],1]&) @*
336         getQ@
337         gdo
338
339     getbCoef::usage = "getbCoef[i][gdo] returns the linear
        ↪ coefficient of b[i]."
340     getbCoef[i_][gdo_] :=
341         (SeriesCoefficient[#, {b[i],0,1}]&) @*
342         (Coefficient[#, a[i],0]&) @*
343         (Coefficient[#, x[i],0]&) @*
344         (Coefficient[#, y[i],0]&) @*
345         ReplaceAll[U2l] @*
346         getL@
347         gdo
348
349     getPCoef::usage = "getPCoef[i][gdo] returns the perturbation P
        ↪ of a GDO as a function of b[i]."
350     getPCoef[i_][gdo_][bb_] :=
351         ReplaceAll[{b[i]->bb}] @*
352         (Coefficient[#, a[i],0]&) @*
353         (Coefficient[#, x[i],0]&) @*
354         (Coefficient[#, y[i],0]&) @*
355         ReplaceAll[U2l] @*
356         getP@
357         gdo
358
359     getaCoef::usage = "getaCoef[i][gdo] returns the linear
        ↪ coefficient of a[i]."

```

```

360 getaCoef[i_][gdo_] :=
361     (SeriesCoefficient[#, {b[i],0,0}]&) @*
362     (Coefficient[#, a[i],1]&) @*
363     ReplaceAll[U2l] @*
364     getL@
365     gdo
366
367 getxCof::usage = "getxCof[i][gdo][b[i]] returns the linear
    ↪ coefficient of x[i] as a function of b[i]."
368 getxCof[i_][gdo_][bb_] :=
369     ReplaceAll[{b[i]->bb}] @*
370     ReplaceAll[U2l] @*
371     (Coefficient[#, y[i],0]&) @*
372     (Coefficient[#, x[i],1]&) @*
373     getQ@
374     gdo
375
376 getabCof::usage = "getabCof[i][gdo] returns the linear
    ↪ coefficient of a[i]b[i]."
377 getabCof[i_][gdo_] :=
378     (SeriesCoefficient[#, {b[i],0,1}]&) @*
379     (Coefficient[#, a[i],1]&) @*
380     ReplaceAll[U2l] @*
381     getL@
382     gdo
383
384 getxyCof::usage = "getxyCof[i][gdo][b[i]] returns the linear
    ↪ coefficient of x[i]y[i] as a function of b[i]."
385 getxyCof[i_][gdo_][bb_] :=
386     ReplaceAll[{b[i]->bb}] @*
387     ReplaceAll[U2l] @*
388     (Coefficient[#, x[i],1]&) @*
389     (Coefficient[#, y[i],1]&) @*
390     getQ@
391     gdo

```



In order to run more efficiently, limits are first computed by direct evaluation, unless such an operation is ill-defined. In such a case, the corresponding series is computed and evaluated at the limit point.

```

392 safeEval[f_][x_] := Module[{fx, x0},
393     If[(fx=Quiet[f[x]]) === Indeterminate,
394         Series[f[x0],{x0,x,0}]/Normal,
395         fx
396     ]
397 ]
398
399 closeComponent[i_][gdo_GD0]:=gdo//
400     setC0[Complement[gdo//getC0,{i}]]//
401     setCC[Union[gdo//getCC,{i}]]

```

Now we come to the implementation of the trace map. The current implementation requires that the coefficient of  $a_i b_i$  be zero. (See chapter 5 for how this restriction limits computability.)

```

402 tr::usage = "tr[i] computes the trace of a GDO element on
    ↪ component i. Current implementation assumes the Subscript[a,
    ↪ i] Subscript[b, i] term vanishes and $k=0."
403 tr::nonzeroSigma = "tr[`1`]: Component `1` has writhe: `2`,
    ↪ expected: 0."
404 tr[i_][gdo_GD0] := Module[
405     {
406         cL = getConstLCoef[i][gdo],
407         cQ = getConstQCoef[i][gdo],
408         βP = getPCoef[i][gdo],
409         ηη = getyCoef[i][gdo],
410         ββ = getbCoef[i][gdo],
411         αα = getaCoef[i][gdo],
412         ξξ = getxCoef[i][gdo],
413         λ = getxyCoef[i][gdo],
414         ta
415     },
416     ta = (1-Exp[-αα]) z[i];
417     expL = cL + αα w[i] + ββ ta;

```

```

418     expQ = safeEval[cQ[#] + z[i]ηη[#]ξξ[#]/(1-z[i]
      ↪ λ[#])&][ta];
419     expP = safeEval[βP[#]/(1-z[i] λ[#])&][ta];
420     CF[(gdo//closeComponent[i]//
      ↪ setL[expL]//setQ[expQ]//setP[expP])//.l2U]
421 ] /; Module[
422     {σ = getabCoef[i][gdo]},
423     If[σ == 0,
424         True,
425         Message[tr::nonzeroSigma, i, ToString[σ]];
      ↪ False
426     ]
427 ]

```

Here we introduce some formatting to display the output more aesthetically.

```

428 Format[gdo_GD0] := Subsuperscript\[DoubleStruckCapitalE],
429     Row[{gdo//getC0, ",", gdo//getCC}],
430     Row[{gdo//getD0, ",", gdo//getDC}]
431 ][gdo//getL, gdo//getQ, gdo//getP];
432 Format[pg_PG] := \[DoubleStruckCapitalE][pg//getL, pg//getQ,
      ↪ pg//getP];
433
434 SubscriptFormat[v_] := (Format[v[i_]] := Subscript[v, i]);
435
436 SubscriptFormat/@{y,b,t,a,x,z,w,η,β,α,ξ,A,B,T,W};

```

### *Implementing the full invariant*

Now we are in a position to implement the  $Z$  invariant to tangles with a closed component. We begin by defining an object representing an isolated strand with arbitrary integer rotation number, CCn:

```

437 CCn[i_][n_Integer] := Module[{j},
438     If[n == 0,
439         GD0["co" -> {i}],
440         If[n > 0,
441             If[n == 1,

```

```

442         CC[i],
443         CC[j]//CCn[i][n-1]//cm[i,j,i]
444     ],
445     If[n== -1,
446         CCi[i],
447         CCi[j]//CCn[i][n+1]//cm[i,j,i]
448     ]
449 ]
450 ]
451 ]

```

Since multiplication is associative, we may implement a generalized multiplication which can take any number of arguments. It is also named `cm`, with a first argument given as an ordered list of indices to be concatenated.

```

452 cm[{}, j_] := cη[j]
453 cm[{i_}, j_] := cσ[i,j]
454 cm[{i_, j_}, k_] := cm[i,j,k]
455 cm[ii_List, k_] := Module[
456     {
457         i = First[ii],
458         is = Rest[ii],
459         j ,
460         js ,
461         l
462     },
463     j = First[is];
464     js = Rest[is];
465     cm[i,j,l] // cm[Prepend[js, l], k]
466 ]

```

The function `toGDO` serves as the invariant for the generators of the tangles. We define its value on crossings and on concatenations of elements.

```

467 toGDO[Xp[i_,j_]] := cR[i,j]
468 toGDO[Xm[i_,j_]] := cRi[i,j]
469 toGDO[xs_Strand] := cm[List@@xs, First[xs]]
470 toGDO[xs_Loop]   := Module[{x = First[xs]}, cm[List@@xs,
    ↪ x]//tr[x]]

```

```

471
472 getIndices[RVT[cs_List, _List, _List]] :=
    ↪ Sort@Catenate@(List@@@cs)
473
474 TerminalQ[cs_List][i_] := MemberQ[Last/@cs,i];
475 next[cs_List][i_] := If[TerminalQ[cs][i],
476     Nothing,
477     ↪ Extract[cs,((#/.{c_,j_}->{c,j+1}& )@FirstPosition[i]@cs)]
478 ]
479
480 InitialQ[cs_List][i_] := MemberQ[First/@cs,i];
481 prev[cs_List][i_] := If[InitialQ[cs][i],
482     Nothing,
483     ↪ Extract[cs,((#/.{c_,j_}->{c,j-1}& )@FirstPosition[i]@cs)]
484 ]

```

To minimize the size of computations, whenever adjacent indices are present in the partial computation, they are to be concatenated before more crossings are introduced.

```

485 MultiplyAdjacentIndices[{cs_List, calc_GD0}] := Module[
486     { is=getC0[calc]
487     , i
488     , i2
489     },
490     i = SelectFirst[is, MemberQ[is, next[cs][#]]&];
491     If[Head[i] === Missing,
492         {cs, calc},
493         i2 = next[cs][i];
494         {DeleteCases[cs, i2, 2], calc//cm[i, i2, i]}
495     ]
496 ]
497
498 MultiplyAllAdjacentIndices[{cs_List, calc_GD0}] :=
499     FixedPoint[MultiplyAdjacentIndices, {cs, calc}]
500

```

```

501 generateGDOFromXing[x:_Xp|_Xm,rs_Association]:=Module[
502     {p, i,j, in, jn},
503     {i,j} = List@@x;
504     {in,jn} = Lookup[rs,{i,j},0];
505     toGDO[x]*CCn[p[i]][in]*CCn[p[j]][jn]
506     ↪ //cm[p[i],i,i]//cm[p[j],j,j]
507 ]
508 addRotsToXingFreeStrands[rvt_RVT] := GDO[] * Times @@ (
509     CCn[#][Lookup[rvt[[3]], #, 0]] & /@
510     First /@ Select[rvt[[1]], Length@# == 1 &]
511 )

```

Next we implement the framed link invariant ZFramed.

```

512 ZFramedStep[{_List,{},_Association,calc_GDO]}:={{},{},<|>,calc};
513 ZFramedStep[{cs_List,xs_List,rs_Association,calc_GDO]}:=Module[
514     { x=First[xs], xss=Rest[xs]
515     , csOut, calcOut
516     , new
517     },
518     new=calc*generateGDOFromXing[x,rs];
519     {csOut,calcOut} = MultiplyAllAdjacentIndices[{cs,new}];
520     {csOut,xss,rs,calcOut}
521 ]
522
523 ZFramed[rvt_RVT] := Last@FixedPoint[ZFramedStep, {Sequence @@
524     ↪ rvt,
525     addRotsToXingFreeStrands[rvt]}}]
526 ZFramed[L_] := ZFramed[toRVT@L]

```

Finally, when we wish to consider the unframed invariant, we apply the function Unwrithe, defined below.

```

526 Z[rvt_RVT] := Unwrithe@Last@FixedPoint[ZFramedStep, {Sequence
527     ↪ @@ rvt, GDO[]}]
528 Z[L_] := Z[toRVT@L]

```

```

529 combineBySecond[l_List] := mergeWith[Total,#]& /@ GatherBy[l,
    ↪ First];
530 combineBySecond[lis___] := combineBySecond[Join[lis]]
531
532 mergeWith[f_, l_] := {l[[1, 1]], f@(#[[2]] & /@ l)}
533
534 Reindex[RVT[cs_, xs_, rs_]] := Module[
535   {
536     sf,
537     cs2, xs2, rs2,
538     repl, repl2
539   },
540   sf = Flatten[List@@#&/@cs];
541   repl = (Thread[sf -> Range[Length[sf]]]);
542   repl2 = repl /. {(a_ -> b_) -> ({a, i_} -> {b, i})};
543   cs2 = cs /. repl;
544   xs2 = xs /. repl;
545   rs2 = rs /. repl2;
546   RVT[cs2, xs2, rs2]
547 ]
548
549 UnwritheComp[i_][gdo_GD0] := Module[
550   {n = gdo//getL//SeriesCoefficient[#, {a[i]b[i], 0, 1}]&,
551     ↪ j},
552   gdo//(cKinkn[-n][j])//cm[i, j, i]
553 ]
554 Unwrithe[gdo_GD0] := (Composition@@(UnwritheComp/@(gdo//getC0)))@gdo
555
556 toRVT[L_RVT] := L

```

The partial trace is what we use to close a subset of the strands in a tangle. It takes the trace of all but one component, then returns the collection of all such ways of leaving one component open. (As described in ??).

```

557 ptr[L_RVT] := Module[
558   {
559     ZL = Z[L],

```

```

560             cod
561         },
562         cod = getC0@ZL;
563         Table[(Composition@@Table[tr[j],
564             ↪ {j, Complement[cod, {i}]}])[ZL], {i, cod}]
565     ]
566 ptr[L_] := ptr[toRVT[L]]

```

In order to be able to compare GDO's properly, we require a way to canonically represent them. This is achieved by reindexing the strands of the link and selecting one who's resulting invariant comes first in an (arbitrarily-selected) order, in this case the built-in ordering of expressions as defined by Mathematica<sup>TM</sup>.

```

566 getGDOIndices[gdo_GDO] := Sort@Catenate@Through[{getD0, getDC,
567     ↪ getC0, getCC}@gdo]
568 isolateVarIndices[i_ -> j_] :=
569     ↪ (v:y|b|t|a|x|η|β|α|ξ|A|B|T|w|z|W)[i] -> v[j];
570 ReindexBy[f_][gdo_GDO] := Module[
571     {
572         replacementRules,
573         varIndexFunc,
574         repFunc,
575         indices = getGDOIndices[gdo]
576     },
577     replacementRules = Thread[indices -> (f/@indices)];
578     repFunc = ReplaceAll[replacementRules];
579     varIndexFunc =
580     ↪ ReplaceAll[Thread[isolateVarIndices[replacementRules]]];
581     gdo//applyToPG[varIndexFunc]//
582         applyToC0[repFunc]//
583         applyToD0[repFunc]//
584         applyToDC[repFunc]//
585         applyToCC[repFunc]
586 ]

```

```

587 fromAssoc[ass_] := Association[ass][#] &
588
589 ReindexToInteger[gdos_List] := Module[
590     {is = getGD0Indices@gdos[[1]], f},
591     f = fromAssoc@Thread[is -> Range[Length[is]]];
592     ReindexBy[f]/@gdos
593 ]
594
595 getReindications[gdos_List] := Module[
596     {
597         gdosInt = ReindexToInteger@gdos,
598         is,
599         fs,
600         ls
601     },
602     is = getGD0Indices@gdosInt[[1]];
603     fs = (fromAssoc@*Association@*Thread)/@{is -> # & /@
        ↪ Permutations[is]};
604     ls = CF@ReindexBy[#]/@gdosInt&/@fs;
605     Sort[Sort/@ls]
606 ]
607
608 getCanonicalIndex[gdo_] := First@getReindications@gdo
609
610 deleteIndex[i_][expr_] := SeriesCoefficient[expr/.U2l, Sequence
        ↪ @@ ({#[i], 0, 0} & /@ {
611     y, b, t, a, x, z, w
612 })]/.l2U

```

Here we introduce functions to further verify the co-algebra structure of a traced ribbon meta-Hopf algebra. In particular, the counit is responsible for deleting a strand. This has further applications in determining whether the invariants of individual components are contained in those of more complex links.

```

613 deleteIndexPG[i_][pg_PG] := pg//
614     applyToL[deleteIndex[i]]//
615     applyToQ[deleteIndex[i]]//

```



```

616         applyToP[deleteIndex[i]]
617
618 deleteLoop[i_][gdo_] := gdo//
619         applyToCC[Complement[#, {i}]]&]//
620         applyToPG[deleteIndexPG[i]]

```

### A.3 IMPLEMENTATION OF ROTATION NUMBER ALGORITHM

#### *Description of algorithm for knots*

Bar-Natan and van der Veen develop an algorithm to convert a classical long knot into an upright tangle. It involves passing a line segment, called the *front*, over the knot, requiring that everything behind the front is in upright form. For example, consider the link: By pulling the crossings along

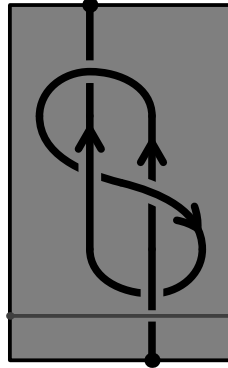


Figure A.1: A knot which is not in upright form. The front is written in grey.

the arc which touches the front, we can bring the knot to upright form.

#### *Extending the algorithm to multiple components*

An algorithm to convert a classical knot diagram into an upright knot diagram was implemented by Bar-Natan and van der Veen. Here we generalize the algorithm to convert a classical tangle with one open component to an upright tangle diagram. This generalization allows us to consider tangle diagrams with multiple components.

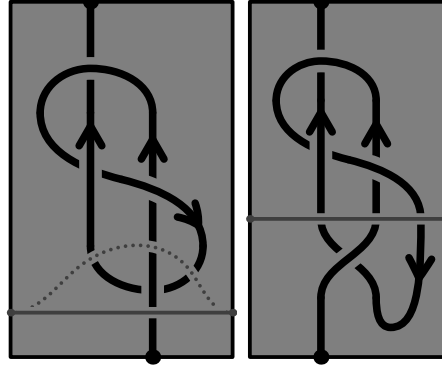


Figure A.2: By advancing the front over a crossing, we bring a crossing into upright form. A dashed front indicates where the front is advancing to.

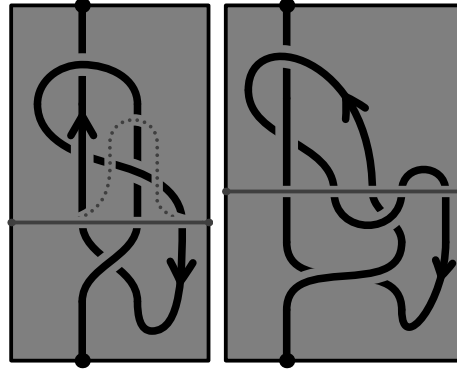


Figure A.3: By advancing the front over a crossing, we bring a crossing into upright form. A dashed front indicates where the front is advancing to.

**Lemma A.1** ([BNvdV], Lemma 43). *For each classical tangle with one open component, there exists a unique upright tangle whose unbounded arcs have rotation numbers 0.*

This is a Haskell implementation<sup>†1</sup> of the algorithm `toRVT`,<sup>†2</sup> which takes a classical tangle and produces an upright tangle by computing a compatible choice of rotation numbers for each arc. This follows largely the same logic as above, except the leftmost strand is always prioritized for absorption, regardless of which component it belongs to.

<sup>†1</sup> The full source code is available at <https://github.com/phro/KnotTheory>.

<sup>†2</sup> Here, the acronym RVT stands for “Rotational Virtual Tangle”, which is another term for “Upright Tangle”.

*Use case*

For example, to query the **SX**-form of a link (i.e. its skeleton-crossing form), one writes:

```
1 >>> link 4 True 1
2 SX [Loop [1,2,3,4], Loop [5,6,7,8]]
3     [Xm 1 6, Xm 3 8, Xm 5 2, Xm 7 4]
```

To convert from the **SX** form to an upright tangle form (here written **RVT**), we must first replace one of the closed **Loops** with an open **Strand** (accomplished by `openFirstStrand`):

```
4 >>> toRVT . openFirstStrand $ link 4 True 1
5 RVT [Strand [1,2,3,4], Loop [5,6,7,8]]
6     [Xm 1 6, Xm 3 8, Xm 5 2, Xm 7 4]
7     [(5, -1), (6, 1), (8, 1)]
```

Reading off the final line, we see that arc 5 has rotation number  $-1$ , arcs 6 and 8 have rotation number 1, and the rest of the arcs have rotation number 0.

*Implementation*

We begin with a series of imports of common functions, relating to list manipulations and type-wrangling. The exact details are not too important.

```
8 {-# LANGUAGE DeriveFunctor #-}
9 module KnotTheory.PD where
10 import Data.Maybe (listToMaybe, catMaybes, mapMaybe, fromMaybe,
11                  ↪ fromJust)
12 import Data.List (find, groupBy, sortOn, partition, intersect,
13                  ↪ union, (\\))
14 import Data.Tuple (swap)
15 import Data.Function (on)
16 import Control.Monad ((>=>))
17 import Control.Arrow ((>>>))
```

Next, we introduce the crossing type, which can be either positive **Xp** or negative **Xm** (using the mnemonic “plus” and “minus”):

```

16 type Index = Int
17 data Xing i = Xp i i | Xm i i -- | Xv i i
18   deriving (Eq, Show, Functor)

```

We define several functions which extract basic data from a crossing.

```

19 sign :: (Integral b) => Xing Index -> b
20 sign (Xp _ _) = 1
21 sign (Xm _ _) = -1
22
23 isPositive :: Xing i -> Bool
24 isPositive (Xp _ _) = True
25 isPositive (Xm _ _) = False
26
27 isNegative :: Xing i -> Bool
28 isNegative (Xp _ _) = False
29 isNegative (Xm _ _) = True
30
31 overStrand :: Xing i -> i
32 overStrand (Xp i _) = i
33 overStrand (Xm i _) = i
34
35 underStrand :: Xing i -> i
36 underStrand (Xp _ i) = i
37 underStrand (Xm _ i) = i

```

Next, we introduce the notion of a planar diagram, whose data is comprised of a collection of **Strands** and **Loops** (indexed by some type *i*, typically an integer). The **Skeleton** of a planar diagram is defined to be the collection of **Components**, each of which is either an open **Strand** or a closed **Loop**.

```

38 type Strand i = [i]
39 type Loop i = [i]
40 data Component i = Strand (Strand i) | Loop (Loop i)
41   deriving (Eq, Show, Functor)
42
43 type Skeleton i = [Component i]

```

Next, we introduce the notion of a **KnotObject**, which has its components labelled by the same type *i*. We further define a function **toRVT** which

converts a generic **KnotObject** into an upright tangle (in this codebase, the term Rotational Virtual Tangle (**RVT**) is frequently used for the notion of an upright tangle). We call an object a planar diagram (or **PD**) if it has a notion of **Skeleton** and a collection of crossings.

```

44 class KnotObject k where
45   toSX  :: (Ord i) => k i -> SX i
46   toRVT :: (Ord i) => k i -> RVT i
47   toRVT = toRVT . toSX
48
49 class PD k where
50   skeleton :: k i -> Skeleton i
51   xings    :: k i -> [Xing i]
52

```

The **SX** form of a diagram just contains the **Skeleton** and the **Xings** (crossings), while the **RVT** form also assigns each arc an integral rotation number.

```

53 data SX i = SX (Skeleton i) [Xing i] deriving (Show, Eq,
54   ↪ Functor)
54 data RVT i = RVT (Skeleton i) [Xing i] [(i,Int)] deriving (Show,
55   ↪ Eq, Functor)

```

Given any labelling of the arcs in a diagram, we can re-label the arcs using consecutive whole numbers. This is accomplished with **reindex**:

```

55 reindex :: (PD k, Functor k, Eq i) => k i -> k Int
56 reindex k = fmap (fromJust . flip lookup table) k
57   where
58     table = zip (skeletonIndices s) [1..]
59     s = skeleton k

```

Most importantly, we now declare that a diagram expressed in **SX** form (that is, without any rotation data) may be assigned rotation numbers to each of its arcs in a meaningful way. The bulk of the work is done by **getRotNums**, which is defined farther below. We handle the case where the entire tangle is a single crossingless strand separately.

```

60 instance KnotObject SX where
61   toSX = id

```

```

62   toRVT k@(SX cs xs) = RVT cs xs rs where
63     rs = filter ((/=0) . snd) . mergeBy sum $ getRotNums k fl
64     il = head . toList $ s
65     Just s = find isStrand cs
66     fl = case next il (toList s) of
67         Just _  -> [(Out,il)]
68         Nothing -> []
69
70   instance KnotObject RVT where
71     toRVT = id
72     toSX (RVT s xs _) = SX s xs
73
74   instance PD SX where
75     skeleton (SX s _) = s
76     xings (SX _ xs)   = xs
77
78   instance PD RVT where
79     skeleton (RVT s _ _) = s
80     xings (RVT _ xs _)   = xs

```

Next, we include a series of functions which answer basic questions about planar diagrams. Note in `rotnum`, if a rotation number is not present in the table of values, it is assumed to be 0.

```

81   rotnums :: RVT i -> [(i,Int)]
82   rotnums (RVT _ _ rs) = rs
83
84   rotnum :: (Eq i) => RVT i -> i -> Int
85   rotnum k i = fromMaybe 0 . lookup i . rotnums $ k
86
87   isStrand :: Component i -> Bool
88   isStrand (Strand _) = True
89   isStrand _          = False
90
91   isLoop :: Component i -> Bool
92   isLoop (Loop _) = True
93   isLoop _        = False
94

```

```

95  toList :: Component i -> [i]
96  toList (Strand is) = is
97  toList (Loop is)   = is
98
99  skeletonIndices :: Skeleton i -> [i]
100 skeletonIndices = concatMap toList
101
102 involves :: (Eq i) => Xing i -> i -> Bool
103 x `involves` k = k `elem` [underStrand x, overStrand x]
104
105 otherArc :: (Eq i) => Xing i -> i -> Maybe i
106 otherArc x i
107   | i == o      = Just u
108   | i == u      = Just o
109   | otherwise   = Nothing
110   where o = overStrand x
111         u = underStrand x
112
113 next :: (Eq i) => i -> Strand i -> Maybe i
114 next e = listToMaybe . drop 1 . dropWhile (/= e)
115
116 prev :: (Eq i) => i -> Strand i -> Maybe i
117 prev e = next e . reverse
118
119 nextCyc :: (Eq i) => i -> Loop i -> Maybe i
120 nextCyc e xs = next e . take (length xs + 1). cycle $ xs
121
122 prevCyc :: (Eq i) => i -> Loop i -> Maybe i
123 prevCyc e xs = prev e . take (length xs + 1). cycle $ xs
124
125 isHeadOf :: (Eq i) => i -> [i] -> Bool
126 x `isHeadOf` ys = x == head ys
127
128 isLastOf :: (Eq i) => i -> [i] -> Bool
129 x `isLastOf` ys = x == last ys
130

```

```

131 nextComponentIndex :: (Eq i) => i -> Component i -> Maybe i
132 nextComponentIndex i (Strand is) = next i is
133 nextComponentIndex i (Loop is) = nextCyc i is
134
135 prevComponentIndex :: (Eq i) => i -> Component i -> Maybe i
136 prevComponentIndex i (Strand is) = prev i is
137 prevComponentIndex i (Loop is) = prevCyc i is
138
139 isHeadOfComponent :: (Eq i) => i -> Component i -> Bool
140 isHeadOfComponent _ (Loop _ ) = False
141 isHeadOfComponent i (Strand is) = i `isHeadOf` is
142
143 isLastOfComponent :: (Eq i) => i -> Component i -> Bool
144 isLastOfComponent _ (Loop _ ) = False
145 isLastOfComponent i (Strand is) = i `isLastOf` is
146
147 isTerminalOfComponent :: (Eq i) => Component i -> i -> Bool
148 isTerminalOfComponent c i = i `isHeadOfComponent` c || i
    ↪ `isLastOfComponent` c
149
150 isTerminalIndex :: (Eq i) => Skeleton i -> i -> Bool
151 isTerminalIndex cs i = any (`isTerminalOfComponent` i) cs
152
153 nextSkeletonIndex :: (Eq i) => Skeleton i -> i -> Maybe i
154 nextSkeletonIndex s i = listToMaybe . mapMaybe
    ↪ (nextComponentIndex i) $ s
155
156 prevSkeletonIndex :: (Eq i) => Skeleton i -> i -> Maybe i
157 prevSkeletonIndex s i = listToMaybe . mapMaybe
    ↪ (prevComponentIndex i) $ s

```

In order to obtain all the crossing indices, we must take every combination of the under- and over-strands and their following indices:

```

158 getXingIndices :: (Eq i) => Skeleton i -> Xing i -> [i]
159 getXingIndices s x = catMaybes
160     [ f a | f <- [id, (>= nextSkeletonIndex s)], a <- [0,
    ↪ u] ]

```



```

161         where o = return (overStrand x)
162               u = return (underStrand x)
163
164      $\delta :: (\text{Eq } a) \Rightarrow a \rightarrow a \rightarrow \text{Int}$ 
165      $\delta \ x \ y$ 
166     |  $x == y$       = 1
167     | otherwise    = 0
168
169     mergeBy :: (Ord i) => ([a] -> b) -> [(i,a)] -> [(i,b)]
170     mergeBy f = map (wrapIndex f) . groupBy ((==) `on` fst) .
        ↪ sortOn fst
171     where
172         wrapIndex :: ([a] -> b) -> [(i,a)] -> (i,b)
173         wrapIndex g xs@(x:_) = (fst x, g . map snd $ xs)

```

Here we come to the main function, `getRotNums`, for which we have the following requirements (not expressed in the code):

1. The diagram `k` is a  $(1, n)$ -tangle (a tangle with only one open component)
2. The underlying graph of `k` is a planar.
3. The diagram `k` is a connected.

Only in this case will the function `toRVT` will then output a planar  $(1, n)$ -upright tangle which corresponds to a classical (i.e. planar) diagram.

This function involves taking a simple open curve (a Jordan curve passing through infinity) called the **Front**, and passing it over arcs in the diagram. This curve is characterized by the arcs it passes through, together with their orientations. Each intersection of the **Front** with the diagram provides a different **View**, either **In** or **Out** of the **Front** when following the orientation of the intersecting arc.

```

174     type Front i = [View i]
175     type View   i = (Dir, i)

```

We obtain the rotation numbers by successively passing the front across new crossings (achieved by `advanceFront`), keeping track of the rotation

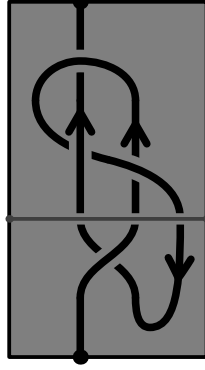


Figure A.4: A tangle with a front passing over it. The portion of the tangle below the front is in upright form.

numbers of arcs which have already passed by the front. Once the front has passed across every crossing, all the rotation numbers have been computed.

Next, we define `converge`, which iterates a function until a fixed point is achieved.

```

176 converge :: (Eq a) => (a -> a) -> a -> a
177 converge f x
178     | x == x'    = x
179     | otherwise = converge f x'
180 where x' = f x

```

The function `convergeT` wraps `converge` in monadic transformations. In our context, the monad will be used to keep track of rotation numbers of the arcs.

```

181 convergeT :: (Monad m, Eq (m a)) => (a -> m a) -> a -> m a
182 convergeT f = return >>> converge (>=> f)

```

The implementation of `getRotNums` takes a front and advances it along a diagram until no more changes occur.

```

183 getRotNums :: (Eq i) => SX i -> Front i -> [(i, Int)]
184 getRotNums k = convergeT (advanceFront k) >>> fst

```

When advancing the `Front`, we start by absorbing arcs that intersect with the front twice until the leftmost `View` no longer connects directly back to the `Front`. At this point, we can absorb a crossing into the front.

```

185 advanceFront :: (Eq i) => SX i -> Front i -> [(i,Int)], Front
    ↪ i)
186 advanceFront k = convergeT (absorbArc k) ==> absorbXing k

```

We next check for the case where the leftmost arc connects back to the **Front**. If it is pointing **Out** (and therefore connects back **In** further to the right), we adjust the rotation number of the arc by  $-1$ . Otherwise, we leave both the **Front** and the rotation numbers unchanged.

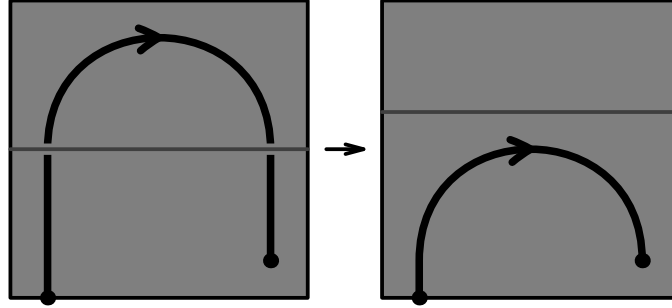


Figure A.5: Example of absorbing an arc which intersects the front multiple times. If the horizontal tangent vector points to the right, as in this picture, then the rotation number of the arc is decreased by 1. Otherwise, no change in the rotation number is recorded.

```

187 absorbArc :: (Eq i) => SX i -> Front i -> [(i,Int)], Front i)
188 absorbArc k [] = return []
189 absorbArc k f@(f1:fs) = case fs1 of
190     ( In,i):_ -> (return (i,-1), fss)
191     (Out,i):_ -> return fss           -- No new rotation
    ↪ numbers
192     [] -> return f
193 where (fs1,fss) = partition (((==) `on` snd) f1) fs

```

Our goal is to repeat this operation until we get a fixed point, which is encoded in **absorbArcs**:

```

194 absorbArcs :: (Eq i) => SX i -> Front i -> [(i,Int)], Front i)
195 absorbArcs k = convergeT (absorbArc k)

```

Absorb a crossing involves expanding one's view at an arc from looking at a crossing to all the views one gets when looking in every direction at the crossing (namely, to the left, along the arc, and to the right). The function

`absorbXing` performs this task on the leftmost **View** on the **Front**. The transverse strand receives a positive rotation number if it moves from left to right. The arc receiving the rotation depends on how the crossing is oriented.

```

196 absorbXing :: (Eq i) => SX i -> Front i -> [(i,Int)],Front i)
197 absorbXing _ [] = return []
198 absorbXing k (f:fs) = (rs,newFront++fs) where
199     newFront = catMaybes [l, a, r]
200     l = lookLeft k f
201     a = lookAlong k f
202     r = lookRight k f
203     rs = case (l,f,r) of
204         (Just (In,i), (Out,_),_ ) -> [(i,1)]
205         (_, (In ,_),Just (Out, j)) -> [(j,1)]
206         _ -> [ ]
207
208 data Dir = In | Out
209     deriving (Eq, Show)

```

The following functions take a **View**, returning the **View** one has when looking in the corresponding direction. Since it is possible for the resulting gaze to be merely the boundary, it is possible for these functions to return **Nothing**.

```

210 lookAlong :: (Eq i, PD k) => k i -> View i -> Maybe (View i)
211 lookAlong k (d, i) = case d of
212     Out -> sequence (Out, nextSkeletonIndex s i)
213     In  -> sequence (In , prevSkeletonIndex s i)
214     where s = skeleton k
215
216 lookSide :: (Eq i, PD k) => Bool -> k i -> View i -> Maybe
217     ⇨ (View i)
218 lookSide isLeft k di@(Out,i) = do
219     x <- findNextXing k di
220     j <- otherArc x i
221     if isLeft == ((underStrand x == i) == isPositive x)
222     then return (In, j)

```

```

222         else sequence (Out, nextSkeletonIndex (skeleton k) j)
223 lookSide isLeft k (In,i) =
224     sequence (Out, prevSkeletonIndex (skeleton k) i) >=>
225     lookSide (not isLeft) k
226
227 lookLeft  :: (Eq i, PD k) => k i -> View i -> Maybe (View i)
228 lookLeft = lookSide True
229
230 lookRight :: (Eq i, PD k) => k i -> View i -> Maybe (View i)
231 lookRight = lookSide False
232
233 findNextXing :: (Eq i, PD k) => k i -> View i -> Maybe (Xing i)
234 findNextXing k (Out,i) = find (`involves` i) $ xings k
235 findNextXing k (In ,i) = do
236     i' <- prevSkeletonIndex (skeleton k) i
237     find (`involves` i') $ xings k

```

# BIBLIOGRAPHY

---

- [BNS] Dror Bar-Natan and Sam Selmani, Meta-monoids, meta-bicrossed products, and the alexander polynomial, no. 10, 1350058, Publisher: World Scientific Publishing Co.
- [BNvdV] Dror Bar-Natan and Roland van der Veen, Perturbed gaussian generating functions for universal knot invariants, no. arXiv:2109.02057.
- [ES] Pavel I. Etingof and Olivier Schiffmann, Lectures on quantum groups, International Press.
- [Kau] Louis H. Kauffman, Rotational virtual knots and quantum link invariants.
- [Maj] Shahn Majid, A quantum groups primer, London Mathematical Society Lecture Note Series, Cambridge University Press.

## COLOPHON

This thesis was typeset using the typographical look-and-feel classicthesis developed by André Miede and Ivo Pletikosić.

The style was inspired by Robert Bringhurst's seminal book on typography The Elements of Typographic Style.