Table des matières

Chapitre 1

Prérequis

L'objectif de ce chapitre est de fournir les différents outils nécessaires à la compréhension de la suite de ce manuscrit. On commence ainsi par une brève introduction à la théorie des noeuds dans $\mathbb{R}^3$, avant de traiter rapidement des invariants de type fini, des groupes de tresses et des diagrammes de Gauss. On transpose ensuite ces explications dans le cadre d'un travail dans le tore solide.

1.1 Introduction à la théorie des noeuds

Cette section a pour objectif d'introduire le lecteur à la théorie des noeuds. Les résultats et les définitions peuvent être approfondis dans [22] ou [29] par exemple.

Définition 1.1.1. Un **noeud orienté** est un plongement $K : S^1 \to \mathbb{R}^3$ pour lequel on a choisi une orientation.

Définition 1.1.2. Un **entrelacs** est un plongement de plusieurs copies de S^1 dans $\mathbb{R}^3$. Le nombre de composantes d'un entrelacs correspond alors au nombre de copies de S^1.

Ainsi, un entrelacs à 1 composante est un noeud. Dans ce manuscrit, nous ne traiterons que des noeuds. Un exemple de noeud est illustré dans la figure 1.1.

Définition 1.1.3. Soit K un noeud orienté donné. On considère la projection canonique $p : \mathbb{R}^3 \to \mathbb{R}^2$ sur le plan (x, y). Alors l'image de K par p où l'on précise à chaque point double quel brin passe au dessus de l'autre, est appelé **diagramme de noeud**. Les points doubles sont appelés **croisements** du diagramme.

Définition 1.1.4. Deux noeuds $K_1 : S^1 \to \mathbb{R}^3$ et $K_2 : S^1 \to \mathbb{R}^3$ sont dits **équivalents** si les deux plongements sont isotopes de manière ambiante : c'est-à-dire qu'il existe une application continue $F : \mathbb{R}^3 \times [0, 1] \to \mathbb{R}^3$ telle que F_0 est l'identité de $\mathbb{R}^3$ dans $\mathbb{R}^3$ et chaque F_t est un homéomorphisme de $\mathbb{R}^3$ dans lui-même et $F_1 \circ K_1 = K_2$.

Remarque. La notion d'isotopie ambiante est une notion plus fine que l'isotopie classique (non introduite ici). En effet, deux noeuds sont toujours isotopes et ainsi, l'étude des noeuds à isotopie près est triviale puisque tout noeud est isotope au noeud trivial. La notion d'isotopie ambiante prend en compte le complémentaire du noeud $\mathbb{R}^3 \setminus \mathrm{Im}(K)$.

Un des principes de la théorie des noeuds est d'essayer de regrouper les noeuds en famille de noeuds équivalents. Un outil récurrent de la théorie des noeuds qui permet de classifier les noeuds est l'invariant de noeuds.

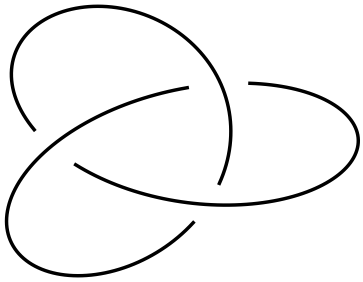


FIGURE 1.1 – Un des noeuds les plus célèbres : le noeud de trèfle

Définition 1.1.5. Un **invariant de noeuds** est une quantité définie pour chaque noeud qui est égale pour tous les noeuds équivalents.

Remarque. On parle d'invariant total si l'on a : $I(K_1) = I(K_2)$ si et seulement si K_1 et K_2 sont équivalents. En général, par définition, un invariant ne va satisfaire qu'un côté de l'équivalence : si K_1 et K_2 sont équivalents, alors $I(K_1) = I(K_2)$.

De nombreux invariants existent et peuvent prendre de multiples formes : un nombre, un polynôme (comme le polynôme d'Alexander [3], le polynôme de Jones [19] ou encore le polynôme HOMFLY [13]), le groupe fondamental du complément d'un noeud ([16]), les invariants de type fini ([32]) ou encore l'intégrale de Kontsevich qui est un invariant complet([23]).

Si la définition de noeuds équivalents peut sembler abstraite au lecteur, le théorème suivant, que l'on doit à Reidemeister ([28]) et Alexander Briggs ([2], en donne une interprétation du point de vue des diagrammes de noeuds.

Théorème 1.1.1. *Deux noeuds sont équivalents si l'on peut passer d'un diagramme d'un noeud à l'autre par une suite de mouvements de Reidemeister décrits dans la figure 1.2*

Remarque. Ici, dans les mouvements de Reidemeister, on suppose que les deux diagrammes de noeuds considérés de part et d'autre du mouvement sont identiques et ne diffèrent que dans un petit voisinage que l'on a illustré.

Une preuve de ce théorème peut être trouvée dans [21]. Ainsi, avec ce théorème, on peut se permettre de travailler directement avec des diagrammes de noeuds. On désignera par la suite, sans ambiguïté, un noeud ou son diagramme de noeud par le même nom.

Une première classification des diagrammes de noeuds a été faite dans [29] pour les noeuds à au plus 10 croisements.

Ainsi, en utilisant le théorème sur les mouvements de Reidemeister, une autre définition d'un invariant de noeud peut être : une quantité **invariante par mouvements de Reidemeister**.

On introduit à présent un outil souvent utilisé dans les invariants de noeuds orientés : le signe d'un croisement.

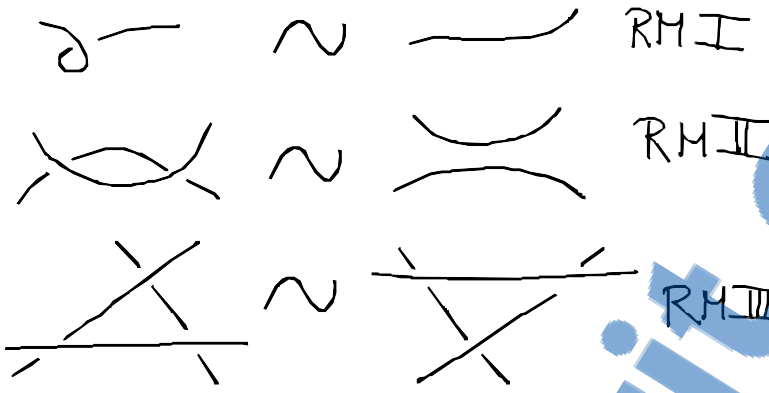


FIGURE 1.2 – Les trois types de mouvements de Reidemeister

Définition 1.1.6. Le **signe d'un croisement** d'un noeud orienté est défini comme tel : un croisement sera positif s'il est de la forme de A et négatif s'il est de la forme de B, où A et B sont représentés dans la figure 1.3.

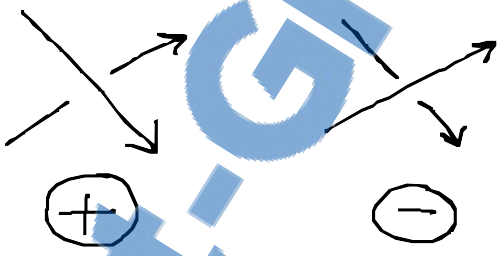


FIGURE 1.3 – Signe d'un croisement, de gauche à droite : forme A et forme B

Un moyen de déterminer le signe est de regarder l'angle orienté entre le brin orienté au dessus et le brin orienté d'en dessous. En effet, pour un croisement positif, aller du brin au dessus vers le brin en dessous donnera un angle orienté positivement (dans le sens trigonométrique) tandis que pour un croisement négatif, cela donnera l'inverse.

1.2 Invariants de type fini

Dans cette section, nous introduisons, sans rentrer dans le détail, les notions d'invariants de type fini et de noeud singulier. Les invariants de type fini ou invariants de Vassilliev ont été découverts par Vassiliev ([32]) et Goussarov ([18]). Pour avoir une lecture plus approfondie, on pourra se référer à [6] qui propose une approche détaillée des invariants de type finis.

L'étude des noeuds orientés peut se prolonger à l'étude de noeuds singuliers. Un **noeud singulier** est un plongement $S^1 \to \mathbb{R}^3$ qui contient un nombre fini de points doubles que l'on appelle des croisements singuliers. Pour généraliser la notion de diagramme de noeuds aux noeuds

singuliers, les points doubles du noeud singulier seront représentés avec des ronds noirs. Ainsi, on peut prolonger un invariant de noeuds V aux noeuds singuliers par la définition suivante :

Définition 1.2.1. V(CroisementSingulier) = V(CroisementPositif) - V(CroisementNegatif) Les croisements sont détaillés dans la figure 1.4.

Remarque. Ici, tout comme pour les noeuds classiques, on suppose que les trois diagrammes de noeuds singuliers considérés sont identiques et ne diffèrent que dans un petit voisinage que l'on a illustré.

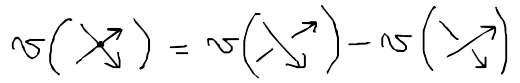


FIGURE 1.4 – Prolongement d'un invariant à un noeud singulier

On étudie généralement les invariants de type fini dans le cadre des noeuds de $\mathbb{R}^3$. Dans ce manuscrit, on s'intéresse à une famille particulière de noeuds : les 4-tresses nodales fermées dans le tore solide (que l'on définit plus loin). Aussi, nous adaptons les définitions classiques d'invariants de type fini à ce cadre particulier.

Définition 1.2.2. Soit m un entier positif. Un invariant V de 4-tresses nodales fermées dans le tore solide est un **invariant de type** m si V s'annule sur les 4-tresses fermées singulières qui ont strictement plus de m croisements singuliers.

Un invariant V est dit **de Vassiliev ou de type fini pour les 4-tresses nodales fermées le tore solide** s'il existe m tel que V est un invariant de type m. L'espace $\mathcal{V}$ de tous les invariants de Vassiliev pour les 4-tresses nodales fermées dans le tore solide est naturellement filtré avec $\mathcal{V}_m = \{\text{invariants de type } m\}$.

Dans le chapitre 3, on reliera la famille principale d'invariants de ce manuscrit à $\mathcal{V}_2$.

1.3 Groupe de tresse

Les tresses sont une famille d'objets noués introduite en 1925 par Artin dans [5]. Elle sert de modèle-jouet (*toy model* en anglais) en théorie des noeuds. Il existe de nombreuses manières de définir des tresses. Pour ce manuscrit, nous avons choisi de les définir de la même façon que dans [14] qui propose une introduction claire et concise des tresses. Nous nous inspirerons donc essentiellement de la structure de [14] dans cette section mais nous avons choisi de prendre la définition de [27] car elle est plus pertinente pour ce travail.

Soit $n \geq 1$ un entier et soit $P_1, ..., P_n$ n points distincts du plan $\mathbb{R}^2$ (on pourra supposer $P_k = (k, 0)$, $\forall 1 \leq k \leq n$).

Définition 1.3.1. Une **tresse à n brins ou n-tresse** est un n-tuple $\beta = (b_1, ..., b_n)$ de chemins, $b_k : [0, 1] \to \mathbb{R}^2$ tels que :

- $b_k(0) = P_k$ pour tout $1 \leq k \leq n$;
- il existe une permutation $\chi = \theta(\beta) \in \mathrm{Sym}_n$ telle que $b_k(1) = P_{\chi(k)}$ pour tout $1 \leq k \leq n$
- $b_k(t) \neq b_l(t), \forall k \neq l$, $\forall t \in [0, 1]$

Définition 1.3.2. On dit alors que deux tresses α et β sont **homotopes** s'il existe une famille continue $\{\gamma_s\}_{s\in[0,1]}$ de tresses telles que $\gamma_0 = \alpha$ et $\gamma_1 = \beta$.

On remarque que $\theta(\alpha) = \theta(\beta)$ (permutations) si α et β sont homotopes.
Dans la suite, ce qui va nous intéresser, ce sont plutôt les classes d'homotopie de tresses.

Définition 1.3.3. Soit I_k une copie de l'intervalle $[0,1]$. Pour une tresse $\beta = (b_1, ..., b_n)$, on définit la **tresse géométrique** :

$$\beta^g : I_1 \sqcup .... \sqcup I_n \to \mathbb{R} \times [0,1]$$

par $\beta^g(t) = (b_k(t), t) \forall t \in I_k, \forall 1 \leq k \leq n$

On considère alors la projection $proj : \mathbb{R}^2 \times [0,1] \to \mathbb{R} \times [0,1]$ définie par $proj(x,y,t) = (x,t)$. Alors, à homotopie près, on peut supposer que $proj \circ \beta^g$ est une immersion lisse avec un nombre fini de points doubles transversaux : ce sont les croisements de la tresse (on précisera quel brin passe au dessus de l'autre). On obtient ainsi un diagramme de tresse de β. On reprend un exemple de [27] dans la figure 1.5.

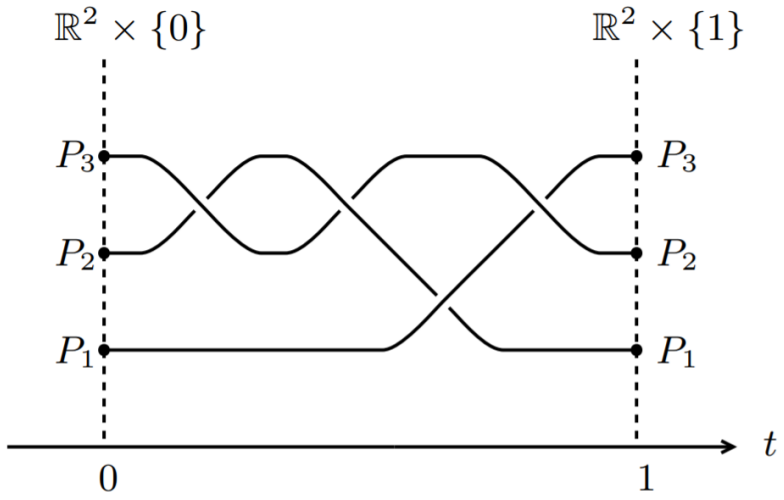


FIGURE 1.5 – Un exemple de tresse

Définition 1.3.4. On définit le produit de deux tresses $\alpha = (a_1, ..., a_n)$ et $\beta = (b_1, ..., b_n)$ par la tresse :

$$\alpha.\beta = (a_1 b_{\chi(1)}, ..., a_n b_{\chi(n)})$$

avec $\chi = \theta(\alpha)$ la permutation associée à α.

Encore une fois, on utilise l'exemple pertinent de [27] dans la figure 1.6.
On note alors B_n l'ensemble des classes d'homotopie des n-tresses.

Proposition 1.3.0.1. *B_n muni du produit de tresse forme un groupe que l'on nomme le groupe de tresses à n-brins.*

Dans ce groupe, l'élément neutre est représenté par la tresse constante $Id = (Id_1, ..., Id_n)$ où Id_k correspond au chemin constant de P_k. L'élément inverse d'une tresse β correspond à son image miroir : en terme de tresses géométriques, les croisements de β^{-1} sont ceux de β avec le signe opposé. Une illustration de [27] peut être trouvée dans la figure 1.7.

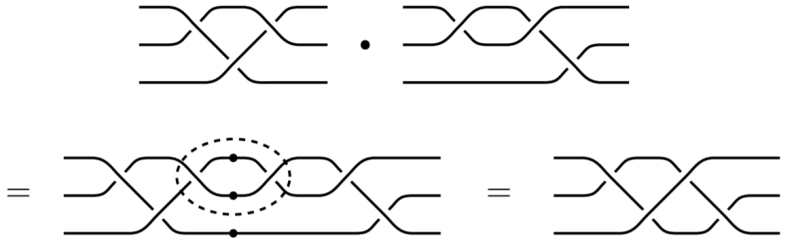


FIGURE 1.6 – Un exemple de produit de deux tresses

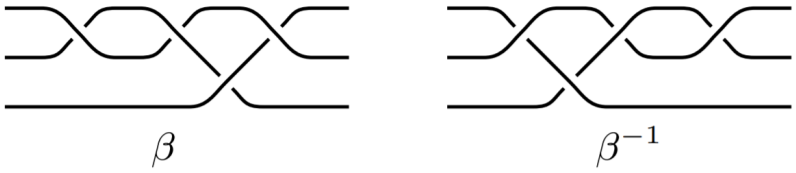


FIGURE 1.7 – L'inverse d'une tresse

Pour $1 \leq k \leq n-1$, on définit par σ_k la tresse qui correspond au croisement positif entre le k-ième brin et celui du dessus (donc le $k+1$-ième brin) et qui vaut l'identité sur les autres brins. On remarque facilement que les σ_k générent le groupe B_n. Un exemple de ces générateurs est illustré dans les figures 1.8 et 1.9.

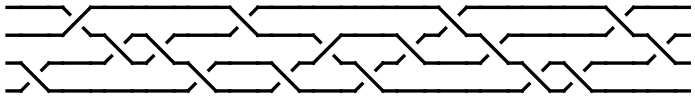


FIGURE 1.8 – Un exemple de tresse : $\bar{\sigma_3}\bar{\sigma_3}\bar{\sigma_1}\bar{\sigma_2}\sigma_1\sigma_1\sigma_1\bar{\sigma_2}\sigma_3\sigma_3\bar{\sigma_1}\bar{\sigma_3}\bar{\sigma_2}\sigma_3\bar{\sigma_2}\bar{\sigma_2}\bar{\sigma_1}$

Le théorème suivant, que l'on doit à Artin ([5],[4]) et Magnus ([24]) nous permet de mieux comprendre le groupe B_n.

Théorème 1.3.1. *Le groupe de tresses B_n engendré par les générateurs $\sigma_1, ..., \sigma_{n-1}$ et les relations :*

- $\sigma_i\sigma_{i+1}\sigma_i = \sigma_{i+1}\sigma_i\sigma_{i+1}$,
- $\sigma_i\sigma_j = \sigma_j\sigma_i$ *si* $|i-j| \geq 2$.

Remarque. La première relation introduite dans le théorème correspond au mouvement de Reidemeister III. Le mouvement de Reidemeister II correspond simplement à la relation naturel dans un groupe : $\sigma_i\sigma_i^{-1} = \sigma_i^{-1}\sigma_i = 1$, où 1 correspond à l'élément neutre de B_n.

Définition 1.3.5. On va définir pour B_n **l'élément de Garside** par :

$$\Delta_n = \prod_{j=1}^{n-1}\prod_{i=1}^{j}\sigma_i = (\sigma_1\sigma_2...\sigma_{n-1})(\sigma_1\sigma_2...\sigma_{n-2})...(\sigma_1\sigma_2)(\sigma_1)$$

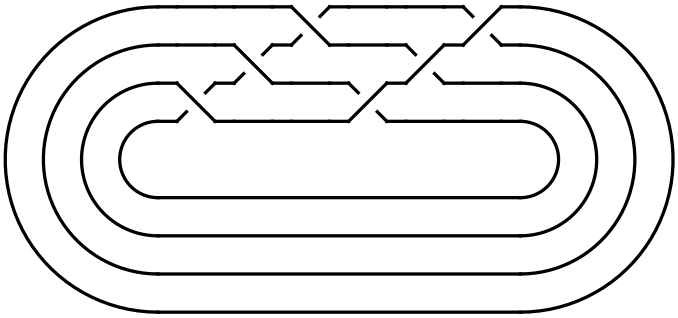


FIGURE 1.9 – La tresse $\sigma_1\sigma_2\sigma_3\bar{\sigma_1}\bar{\sigma_2}\bar{\sigma_3}$

Une illustration pour $n = 4$ est donnée dans la figure 1.10

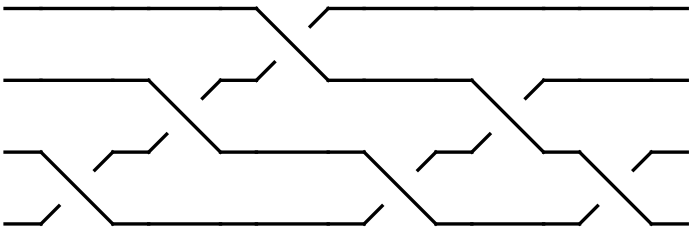


FIGURE 1.10 – L'élément de Garside $\Delta_4 = \sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1$

Définies comme tel, les tresses ne correspondent ni à des entrelacs ni à des noeuds. Pour obtenir des noeuds ou des entrelacs, il faut alors "refermer" la tresse en reliant les extrémités de chaque brin $b_i(0), b_i(1)$. Un célèbre théorème que l'on doit à J.W. Alexander ([1]) démontre que n'importe quel entrelacs peut être obtenu comme une fermeture de tresse.

Définition 1.3.6. Une **tresse nodale** est une tresse qui se referme en un noeud.

Remarque. Cette définition de tresse nodale n'est pas usuelle. Attention, en général, quand on parle de tresses nodales, ce sont des tresses qui ont des points singuliers. Ici, dans ce travail, les tresses nodales désigneront des tresses **classiques** mais qui se referment en un noeud (entrelacs à 1 composante).

Comme indiqué dans la section précédente, nous ne traiterons que de tresses nodales dans ce manuscrit. Plus précisément, nous allons étudier les 4-tresses nodales. Un exemple est donné dans la figure 1.11.

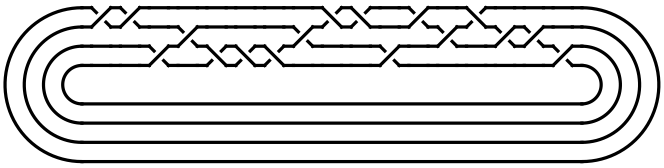


FIGURE 1.11 – La tresse de la figure 1.8 est une **tresse nodale**

1.4 Diagrammes de Gauss

Les diagrammes de Gauss sont un outil permettant de formaliser les diagrammes de noeuds. Ils sont particulièrement utiles lorsque l'on veut coder informatiquement un diagramme de noeuds. On peut également citer [8] qui utilise les diagrammes de Gauss afin, entre autre, de donner une autre manière de calculer le polynôme de HOMFLY-PT.

Définition 1.4.1. Un **diagramme de Gauss** est un cercle S^1 avec un ensemble de flèches signées dont les extrémités appartiennent au cercle.

Remarque. Habituellement, un diagramme de Gauss sera représenté avec les flèches "à l'intérieur du cercle".

On peut relier cette notion à la notion de diagramme de noeud : sachant qu'un noeud K est un plongement de S^1, le cercle correspond à S^1, les flèches correspondent aux croisements de K et les signes des flèches sont naturellement les signes des croisements qu'elles représentent. On choisit la convention suivante : le pied de la flèche correspond au sous-croisement et la tête de flèche au sur-croisement.

Remarque. Pour faciliter la traduction d'un diagramme de noeud en diagramme de Gauss, on numérote les croisements. Informatiquement, c'est ce qui est fait. Ainsi, pour générer un diagramme de Gauss, l'algorithme se fixe arbitrairement un point de départ (nommé le point de base et symbolisé par $*$) et parcourt le noeud dans le sens de son orientation. On commence par une liste vide [] à laquelle on va rajouter les croisements au fur et à mesure du parcours. Si l'algorithme passe par le croisement numéroté i en empruntant le brin du dessus, on rajoute $-i$ à la liste. Si c'est par le brin du dessous, on rajoute i. La liste finale obtenue permet de tracer le diagramme de Gauss. Il suffit de tracer un cercle, de placer les différents i et $-i$ dans l'ordre de la liste puis de les relier par un segment. $-i$ correspond alors à la tête de flèche et i à un pied de flèche. Un exemple pour le noeud de trèfle est donné dans la figure 1.12.

Remarque. Tout noeud (et plus généralement, tout entrelacs) se traduit en un diagramme de Gauss (pour un entrelacs, le diagramme de Gauss sera composé d'autant de cercles qu'il aura de composantes). Cependant, un diagramme de Gauss ne correspond pas forcément à un noeud (ou un entrelacs). La notion qui permet d'obtenir une équivalence parfaite entre diagrammes de Gauss et noeuds est la notion de noeuds virtuels que nous ne traitons pas dans ce manuscrit.

Puisqu'il y a une correspondance entre noeuds et diagrammes de Gauss, il est pertinent de traduire les différents mouvements de Reidemeister en terme de diagrames de Gauss. Cette traduction est illustrée dans la figure 1.13.

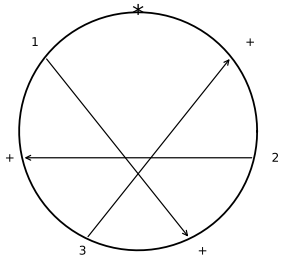


FIGURE 1.12 – Le diagramme de Gauss du noeud de trèfle dont la fermeture de tresse s'écrit $\sigma_1\sigma_1\sigma_1$

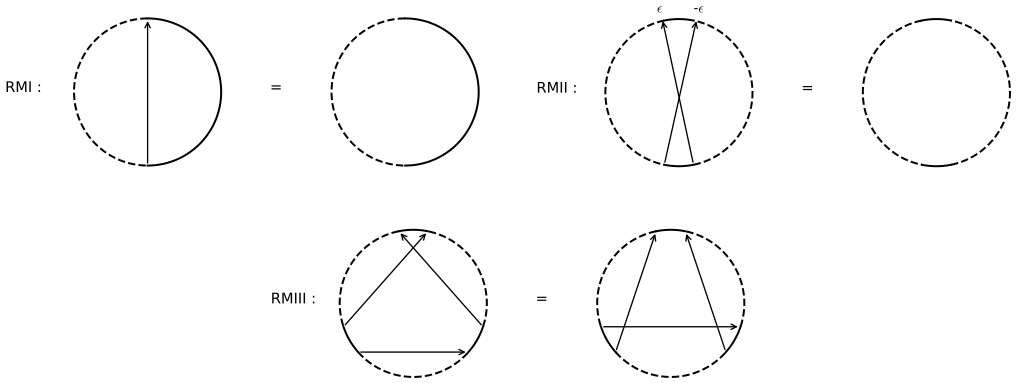


FIGURE 1.13 – Les mouvements de Reidemeister en terme de diagrammes de Gauss

1.5 Notions dans le cadre du tore solide

A présent, nous allons recentrer notre travail sur le tore solide. Tout d'abord, il faut citer un célèbre théorème que l'on doit à Markov ([25]) et dont une preuve peut être lue dans [20] ou [7]. Ce que nous dit ce théorème, c'est que deux tresses fermées dans $\mathbb{R}^3$ sont équivalentes si et seulement si elles sont conjuguées (b et b' sont **conjuguées** s'il existe $a \in B_n$ telle que $aba^{-1} = b'$). Or, b et b' sont conjuguées ssi leur fermetures dans le tore solide sont isotopes comme tresses fermées ([30]). De plus, deux tresses fermées dans le tore solide sont isotopes comme tresses fermées si et seulement si ils sont isotopes comme noeuds (entrelacs) dans le tore solide ([26]). Cela motive ainsi notre travail dans le tore solide.

Définition 1.5.1. Un **noeud dans le tore solide** est un plongement $K : S^1 \to D^2 \times S^1$

Concernant les diagrammes de noeuds, via l'inclusion du tore solide dans $\mathbb{R}^3$ (on considère alors le tore solide comme le solide engendré par la rotation de D^2 autour de l'axe z) permet de visualiser les diagrammes de noeuds dans le tore comme un diagramme de noeuds où l'on a rajouté l'information du centre du tore. Géométriquement, on le symbolisera par un point noir. On illustre cela dans la figure 1.14.

Les mouvements de Reidemeister pour un noeud dans le tore solide sont essentiellement les mêmes que dans $\mathbb{R}^3$ à condition que cela se passe "loin" du point noir.

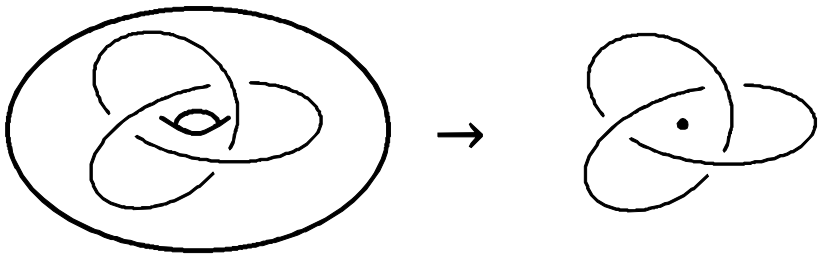


FIGURE 1.14 – Le noeud de trèfle en tant que noeud dans le tore solide

Le contexte du tore solide est très favorable au travail avec les tresses. En effet, la fermeture d'une tresse dans le tore solide est assez naturelle.

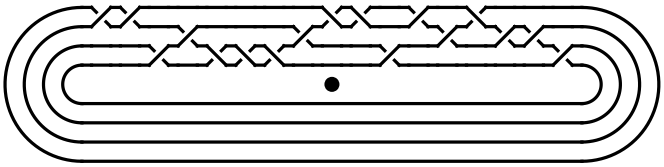


FIGURE 1.15 – La tresse $\bar{\sigma_3}\bar{\sigma_3}\bar{\sigma_1}\bar{\sigma_2}\sigma_1\sigma_1\sigma_1\bar{\sigma_2}\sigma_3\sigma_3\bar{\sigma_1}\bar{\sigma_3}\bar{\sigma_2}\sigma_3\bar{\sigma_2}\bar{\sigma_2}\bar{\sigma_1}$ en tant que tresse nodale dans le tore solide

Maintenant que nous avons parlé de noeuds et de tresses dans le tore solide, il est temps de faire le lien avec les diagrammes de Gauss. **Comment représenter un noeud dans le tore solide par un diagramme de Gauss à partir de son diagramme de noeuds dans le tore solide ?** On ne s'interessera qu'aux tresses nodales dans le tore solide à partir de maintenant.

Le diagramme d'un noeud dans le tore solide est le diagramme de ce noeud dans $\mathbb{R}^3$ enrichi de l'information du point noir. En termes de diagramme de Gauss, il est plus complexe de coder cette information.

Définition 1.5.2. On définit le **lacet standard associé à un croisement** dans un diagramme de tresse nodale dans le tore solide comme le lacet qui part du surcroisement dans le sens de l'orientation et revient au croisement.

La figure 1.16 présente un exemple de lacet standard associé à un croisement.

Puisque le groupe fondamentale du tore solide est $\mathbb{Z}$, on peut **associer un entier à chaque croisement** que l'on nommera **marquage**. De plus, la tresse nodale pouvant elle même être considérée comme un lacet du tore solide, on peut également considérer sa classe dans le groupe fondamental.

Proposition 1.5.0.1. *La classe d'une n-tresse nodale dans le groupe fondamental du tore solide est n.*

Démonstration. Une tresse nodale contient une seule composante. Ainsi, pour connaître sa classe dans le groupe fondamental du tore solide, il suffit de "compter" le nombre de "tours" autour du centre du tore. D'où le résultat. □

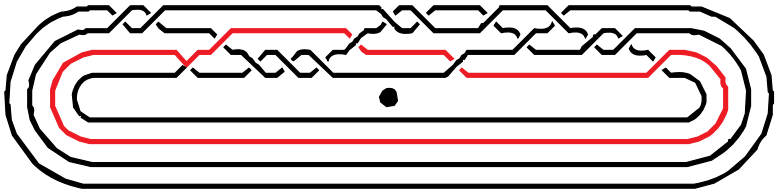


FIGURE 1.16 – Le lacet standard associé au troisième croisement de $\bar{\sigma_3}\bar{\sigma_3}\bar{\sigma_1}\bar{\sigma_2}\sigma_1\sigma_1\sigma_1\bar{\sigma_2}\sigma_3\sigma_3\bar{\sigma_1}\bar{\sigma_3}\bar{\sigma_2}\sigma_3\bar{\sigma_2}\bar{\sigma_2}\bar{\sigma_1}$

Ainsi, on sait que les entiers que l'on associe à chaque croisement sont dans $\{0, ..., n\}$. On peut même affiner le résultat puisque l'on traite de tresses :

Proposition 1.5.0.2. *Le marquage d'un croisement d'une n-tresse est un entier entre 1 et $n-1$.*

Démonstration. La structure d'une tresse nodale empêche tout lacet associé à un croisement d'avoir une classe d'homotopie de 0. Egalement, le lacet associé à un croisement ne peut avoir une classe d'homotopie de n sinon ce serait la tresse entière. □

A présent, nous avons toutes les informations pour représenter une tresse nodale dans le tore solide par un diagramme de Gauss : on commence par générer son diagrammme de Gauss si on considère la tresse nodale comme un noeud de $\mathbb{R}^3$ (via l'inclusion naturelle du tore solide dans $\mathbb{R}^3$) et on enrichit chaque flèche d'un marquage.

Puisque dans le manuscrit nous ne travaillerons qu'avec des 4-tresses, les marquages ne peuvent être que 1,2 ou 3. Ainsi, pour alléger les diagrammes de Gauss, un code couleur a été choisi pour représenter ces marquages. **Rouge pour 1, bleu pour 2 et vert pour 3**.

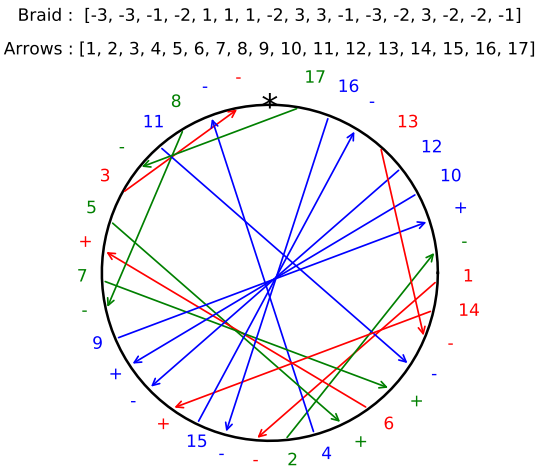


FIGURE 1.17 – Exemple de diagramme de Gauss pour la tresse $\bar{\sigma_3}\bar{\sigma_3}\bar{\sigma_1}\bar{\sigma_2}\sigma_1\sigma_1\sigma_1\bar{\sigma_2}\sigma_3\sigma_3\bar{\sigma_1}\bar{\sigma_3}\bar{\sigma_2}\sigma_3\bar{\sigma_2}\bar{\sigma_2}\bar{\sigma_1}$

Remarque. Dans ce manuscrit, nous avons choisi une convention pour le lacet associé à un croisement. Cependant, pour un croisement donné d'une tresse nodale, il y a exactement deux lacets naturels dont la concaténation donne la tresse entière. Ainsi, quelqu'un qui choisirait l'autre convention n'aurait qu'à transformer tous les marquages m en $n - m$.

Les détails du calcul des marquages des flèches sont disponibles dans le chapitre 4.

1.6 Théorie des noeuds à un paramètre

La plupart des invariants classiques se calculent à partir du diagramme d'un noeud ou de son diagramme de Gauss. L'approche de ce manuscrit est différente, dans la continuité directe de [11].

On considère M_n **l'espace de tous les diagrammes de n-tresses nodales dans le tore solide**. Cet espace est naturellement stratifié. Par exemple, la strate de codimension 1 Σ^1 correspond aux mouvements de Reidemeister. [12] détaille l'étude de cette stratification.

Pour calculer notre invariant de 4-tresses nodales, on va donc associer un **lacet dans** M_4 à une 4-tresse nodale. Chaque point de ce lacet correspondant à un diagramme de tresse, on peut voir ce lacet comme un "film" d'une tresse qui tourne en boucle.

1.6.1 Le lacet *rot*

Dans la suite, on détaille la définition du lacet *rot* introduit par Gramain ([17]) que l'on va utiliser pour calculer notre invariant.

On considère le tore solide $T = S^1 \times D^2 \hookrightarrow \mathbb{R}^3$ $z - axe$.

Définition 1.6.1. $rot(T)$ est la famille des difféomorphismes de T définis de la manière suivante : on effectue une rotation d'un angle $t \in [0, 2\pi]$ sur le tore solide autour de son centre de telle sorte que chaque disque (section du tore) est invariant et est tourné simultanément autour du centre.

Définition 1.6.2. Soit β une tresse nodale du tore solide. $rot(\beta)$ est le lacet induit par $rot(T)$.

Cette définition nous permet d'avoir une interprétation géométrique du lacet *rot*. Cependant, dans la suite, puisque l'on veut programmer un algorithme qui calculera notre invariant, il sera plus utile de décrire ce lacet algébriquement pour les tresses.

Définition 1.6.3. Soit β une tresse nodale du tore solide. Alors le lacet $rot(\beta)$ peut être défini par la séquence suivante :

$$\beta \to \beta\Delta\Delta^{-1} \to \Delta\beta'\Delta^{-1} = \beta'\Delta^{-1}\Delta = \beta'\Delta\Delta^{-1} \to \Delta\beta\Delta^{-1} = \beta\Delta^{-1}\Delta = \beta$$

où Δ est l'élément de Garside défini dans une section précédente.

Remarque.
- La séquence $\beta \to \beta\Delta\Delta^{-1}$ correspond à une suite de mouvements de Reidemeister II,
- La séquence $\beta\Delta\Delta^{-1} \to \Delta\beta'\Delta^{-1}$ correspond à une suite de mouvements de Reidemeister II et III.

On voit ainsi déjà les premiers éléments qui nous permettront d'implémenter la partie du programme qui calculera le lacet *rot* pour une 4-tresse nodale.

1.6.2 Mouvement de Reidemeister III et diagrammes de Gauss

Lorsque l'on va considérer les éléments de $rot(\gamma) \cap \Sigma^1$, on obtient des diagrammes de tresses qui contiennent 1 point singulier : ces diagrammes correspondent au moment précis où les croisements ne sont plus des points doubles lors d'un mouvement de Reidemeister. Ainsi, si le mouvement est un mouvement de Reidemeister II (on dira que le diagramme est un point de Σ^1_{tan}), alors le point singulier est un point d'autotangence (les deux croisements du mouvement de Reidemeister II sont sur le point de s'annuler). En termes de diagrammes de Gauss, un tel point singulier se traduira par deux flèches de signes opposés confondues l'une dans l'autre. En revanche, si le mouvement est un mouvement de Reidemeister III (on dira que le diagramme est un point de Σ^1_{tri}), alors le point singulier est un point triple (les trois brins sont superposés au dessus d'un même point). Transposé en diagrammes de Gauss, ce point triple correspond alors à 3 flèches qui forment un triangle (les trois croisements du mouvement de Reidemeister sont superposés). On nommera ces diagrammes particuliers des diagrammes à point triple. Le but de cette sous-section est de détailler les différents outils en lien avec ces diagrammes à point triple qui permettront le calcul de nos invariants.

On appellera **configuration à point triple** un cercle muni de trois flèches qui forment un triangle et d'autres flèches. On associe un signe à chaque flèche de la configuration.

Remarque. Puisqu'une configuration à point triple est destinée à être "évaluée" sur un diagramme de Gauss à point triple, les définitions introduites dans cette partie seront valables pour les deux.

Définition 1.6.4. Un diagramme de Gauss à point triple correspond au moment critique, dans un mouvement de Reidemeister III entre trois brins, où les trois croisements forment un point triple. Pour plus de clarté dans les futurs calculs, on nomme les différents croisements :

- d est le croisement entre le brin le plus haut et le brin le plus bas,
- hm est le croisement entre le brin le plus haut (high) et le brin du milieu (medium),
- ml est le croisement entre le brin du milieu (medium) et le brin le plus bas (low).

Remarque. On peut facilement retrouver cette nomenclature dans un diagramme de Gauss à point triple : ml correspond à la flèche dont la tête ne touche aucune autre tête de flèche et d correspond à la flèche dont la tête touche une tête et le pied touche un pied. hm est alors la flèche restante.

Un exemple est illustré dans la figure 1.18.

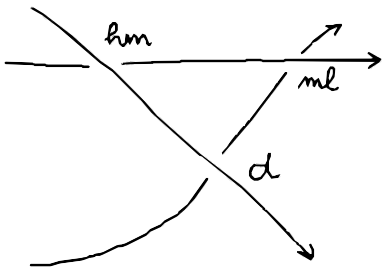


FIGURE 1.18 – Exemple de noms des croisements d'un mouvement de Reidemeister III

Puisqu'il y a une coorientation naturelle (illustrée dans la figure 1.19) pour Σ^1_{tri}, lorsque l'on considère un diagramme de l'intersection $rot(\gamma) \cap \Sigma^1_{tri}$, on peut définir le signe d'un tel diagramme à point triple.

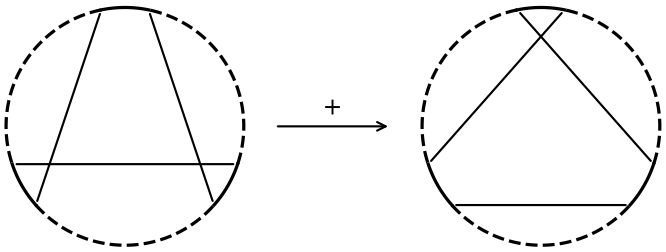


FIGURE 1.19 – La coorientation de Σ^1_{tri}

Définition 1.6.5. Soit $p \in rot(\gamma) \cap \Sigma^1_{tri}$. Alors, on définit le **signe de** p comme suit :

— $sign(p) = 1$ si l'orientation de $rot(\gamma)$ coïncide avec la coorientation de $\Sigma^1_t ri$,

— $sign(p) = -1$ sinon.

Pour faciliter le calcul de ce signe informatiquement, il est pratique d'introduire deux nouvelles notions : le type global et le type local d'un diagramme de Gauss à point triple.

Définition 1.6.6. Il y a exactement deux types globaux sans marquage : positif ou négatif. Les deux types sont résumés dans la figure 1.20. Lorsque l'on rajoute les marquages de d, hm et ml, on obtient le type global d'un point triple. On peut alors créer une nomenclature pour désigner le type global d'un point triple :

$([ml], [hm])^\epsilon$, où $\epsilon \in \{-1, +1\}$

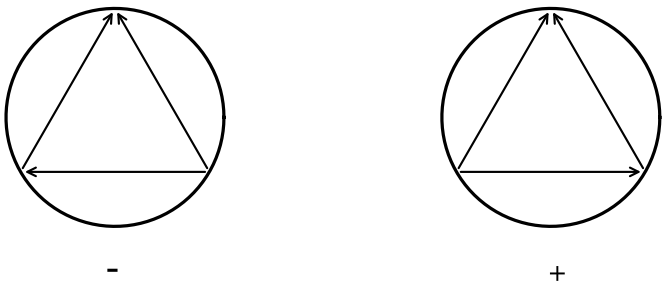


FIGURE 1.20 – Les deux types globaux sans marquage d'un point triple

Définition 1.6.7. Dans le cas des tresses nodales, il y a exactement 6 types locaux différents (en fait, il y a 8 types locaux pour un noeud détaillés dans [10] et puisqu'on considère des tresses, il y en a 2 de moins) . Le type local dépend des croisements impliqués dans le mouvement de Reidemeister III. Les 6 types locaux sont détaillés dans la figure 1.21.

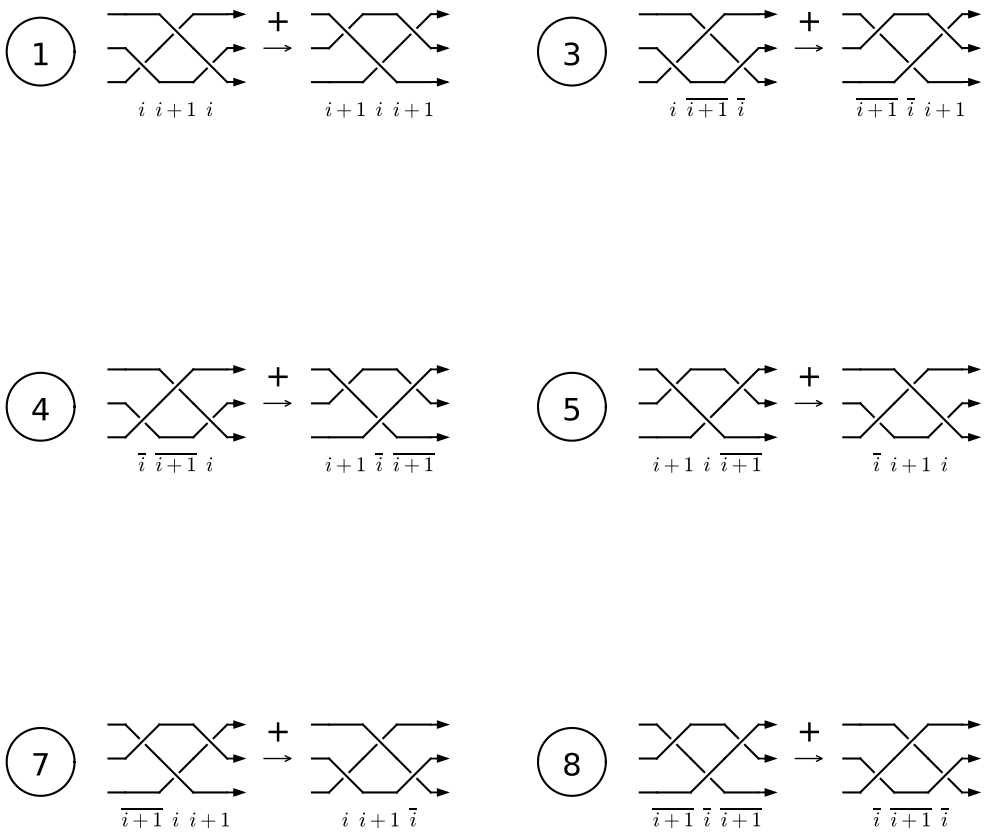


FIGURE 1.21 – Les 6 types locaux d'un point triple (en gardant les noms donnés dans [10])

Proposition 1.6.0.1. *Soit $p \in rot(\gamma) \cap \Sigma^1_{tri}$, alors on a :*

$$sign(p) = type_{global}(p) \times type_{local}(p)$$

Chapitre 2

Les 1-cocycles

Dans ce chapitre, nous détaillons la construction de 1-cocycles qui représentent des classes non triviales de $H^1(M_n; \mathbb{Z}[x_1, x_2, ..., x_1^{-1}, x_2^{-1}, ...])$ en s'inspirant de [11] qui permettront de calculer des invariants pour les 4-tresses dans le tore solide. Pour rappel, M_n est l'espace de tous les diagrammes de n-tresses nodales dans le tore solide. L'invariance repose essentiellement sur la détermination de poids qui consistent en des formules de Gauss de configurations à point triple. L'étude a été pensée et approfondie dans le cas des 4-tresses mais puisque le programme en annexe a été pensé pour des n-tresses avec $n \geq 4$, nous traiterons des n-tresses dans ce chapitre avec des précisions à chaque fois pour le cas $n = 4$.

Définition 2.0.1. Une **configuration de Gauss à point triple** consiste en un cercle avec un triangle de flèches marquées avec un certain nombre de flèches marquées supplémentaires.

Ces configurations de Gauss à point triple seront évaluées sur des diagrammes de Gauss d'une tresse dans le tore solide au moment critique d'un mouvement de Reidemeister III lorsque les trois croisements forment un point triple (les flèches correspondantes forment alors un triangle).

Définition 2.0.2. Un 1-cocycle sera dit de **degré de Gauss** d si son poids correspondant est constitué de configurations à point triple contenant exactement d flèches.

2.1 1-cocycles de degré de Gauss 0

Théorème 2.1.1. *Soit γ une n-tresse fermée dans le tore solide. On définit :*

$$\Gamma(rot(\gamma)) = \sum_{p \ de \ type \ (a,b)^\epsilon} sign(p)$$

avec $(a, b)^\epsilon$ un type de configuration à point triple autorisé

Alors $\Gamma(rot(.))$ est un invariant pour les n-tresses nodales fermées dans le tore solide

Démonstration. Pour montrer que $\Gamma(rot(.))$ est un invariant de tresses, on va montrer que Γ est invariant par homotopies : si γ et γ' sont deux tresses qui sont reliées par des mouvements de Reidemeister, alors $rot(\gamma)$ est homotope à $rot(\gamma')$. La problématique principale de la preuve est la suivante : si l'homotopie traverse une strate de codimension 2 en un point P, il faut montrer que Γ vaut 0 sur un méridien m de P : s' est homotope à $s + m$. Une illustration a été faite dans

la figure 2.1.
La preuve du théorème se fait donc en montrant que Γ vaut 0 sur les méridiens de

$$\Sigma^{(2)} = \Sigma^{(2)}_{quad} \cup \Sigma^{(2)}_{trans-self} \cup \Sigma^{(2)}_{self-flex} \cup \Sigma^{(2)}_{inter}$$

Une étude plus approfondie de $\Sigma^{(2)}$ peut être trouvée dans [10]. Mais on peut comprendre que $\Sigma^{(2)}$ est l'espace des cas singuliers de diagrammes de n-tresses nodales fermées dans le tore solide lorsque l'on s'autorise à "bouger localement deux zones du diagramme". Si on applique cette intuition à $\Sigma^{(1)}$, on obtient bien que $\Sigma^{(1)}$ correspond aux cas singuliers de diagrammes de n-tresses nodales fermées dans le tore solide lorsque l'on effectue un mouvement de Reidemeister (dans ces mouvements, il n'y a qu'une "zone" qui est bougée). Ainsi, on peut intuiter que $\Sigma^{(2)}$ correspond à des cas singuliers de diagrammes de n-tresses nodales fermées dans le tore solide lorsque l'on effectue deux mouvements de Reidemeister simultanément. Si les deux mouvements ont lieu chacun dans des zones "éloignées", on se retrouve dans $\Sigma^{(2)}_{inter}$. $\Sigma^{(2)}_{quad}$ correspond au cas où l'on effectue deux mouvements de Reidemeister III et que l'on obtient un point quadruple. $\Sigma^{(2)}_{trans-self}$ correspond au cas où l'on effectue un mouvement de Reidemeister II et un mouvement de Reidemeister III. $\Sigma^{(2)}_{self-flex}$ correspond au cas où l'on effectue deux mouvements de Reidemeister II. La compréhension de ces différentes strates est plus facile lorsque l'on observe un de leurs méridiens (on les détaille dans la suite).

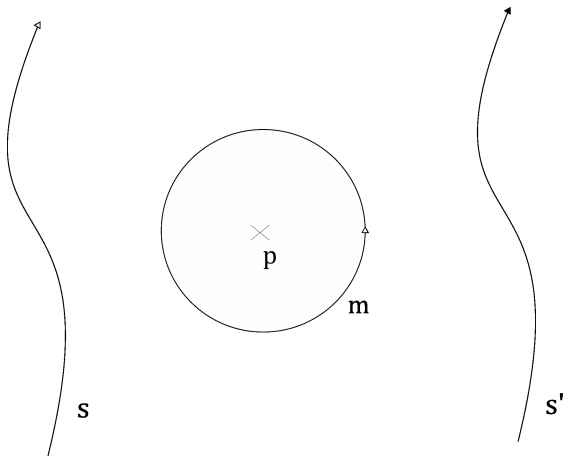


FIGURE 2.1 – Comment "contourner" une homotopie qui passerait par $\Sigma^{(2)}$: s' est homotope à $s + m$

Un méridien de $\Sigma^{(2)}_{quad}$ intersecte $\Sigma^{(1)}_{tri}$ en quatre points. Ces points peuvent être réunis en paires de signes opposés. Ainsi, on a $\Gamma(m) = 0$ pour tout méridien m de $\Sigma^{(2)}_{quad}$. Un exemple de méridien est illustré dans la figure 2.2.

On observe également que chaque type de méridien de $\Sigma^{(2)}_{trans-self}$ intersecte $\Sigma^{(1)}_{tri}$ en deux points qui sont toujours du même type mais de signes opposés. Ainsi, on a $\Gamma(m) = 0$ pour tout méridien m de $\Sigma^{(2)}_{trans-self}$. Un exemple de méridien est illustré dans la figure 2.3.

Enfin, la contribution d'un méridien de $\Sigma^{(2)}_{inter}$ ou de $\Sigma^{(2)}_{self-flex}$ est triviale. Deux exemples de méridiens sont illustrés dans les figures 2.4 et 2.5.

□

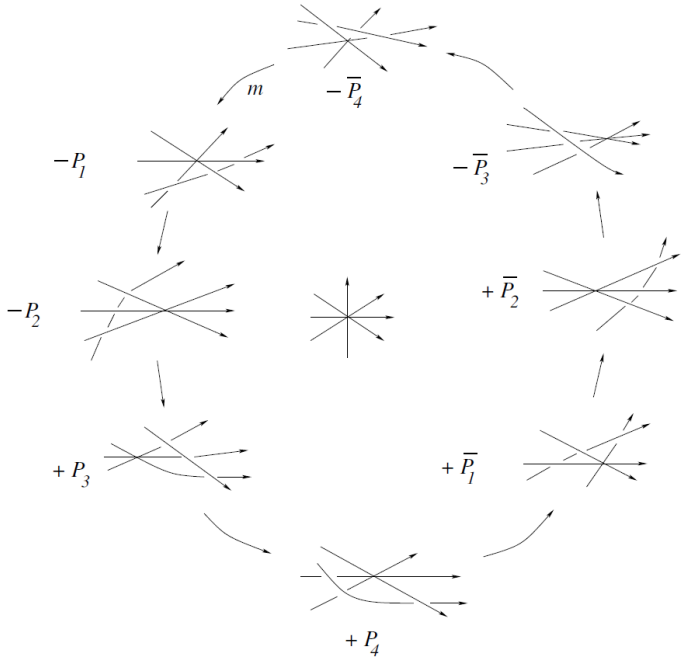


FIGURE 2.2 – Exemple d'un méridien de $\Sigma^{(2)}_{quad}$ avec quatre croisements positifs

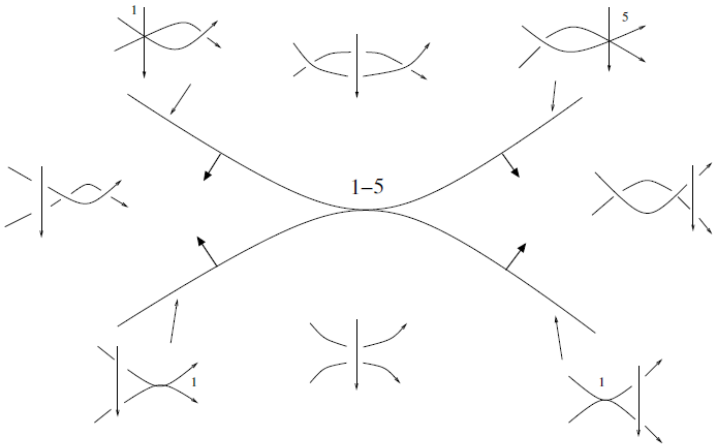


FIGURE 2.3 – Exemple d'un méridien de $\Sigma^{(2)}_{trans-self}$

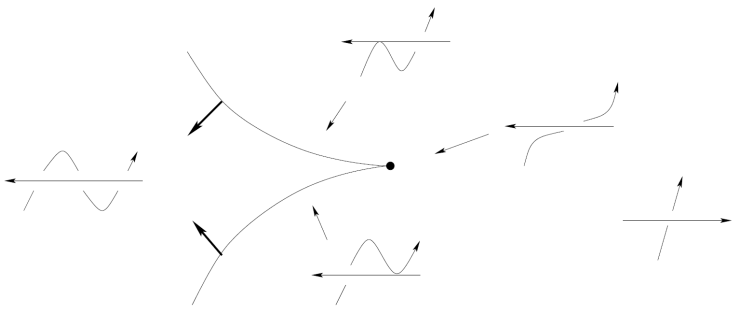


FIGURE 2.4 – Exemple d'un méridien de $\Sigma^{(2)}_{self-flex}$

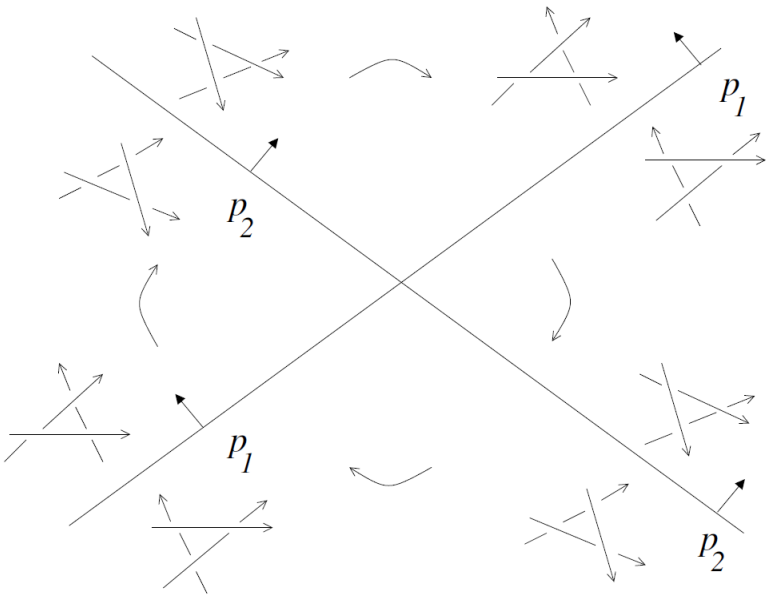


FIGURE 2.5 – Exemple d'un méridien de $\Sigma^{(1)}_{tri} \cap \Sigma^{(1)}_{tri}$

Remarque. On peut remarquer que pour un arc $S \in M$, $\Gamma(S)$ est ici l'intersection algébrique de S avec $\Sigma^{(1)}_{tri}$. De plus, Γ est bien un 1-cocycle : c'est une application de l'ensemble des lacets de M_n vers l'ensemble des polynômes de Laurent à coefficients entiers.

Exemple. Dans tout ce manuscrit, on travaillera toujours avec le même exemple qui permet de faire des calculs simples et rapides pour apprendre à manipuler les différents outils introduits dans ce travail. On pose $S_{exemple} = \sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1\sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1$. On va alors calculer le 1-cocycle de degré 0 associé au type $(2,3)^+$ pour $rot(S_{exemple}) : \Gamma(rot(S_{exemple})) = -0+0+0-0 = 0$. Plus précisément, il y a exactement 12 mouvements de Reidemeister qui interviennent dans $rot(S_{exemple})$ et seulement 4 correspondent à des diagrammes de Gauss à point triple de type $(2,3)^+$ et on les présente dans les figures 2.6, 2.7 et 2.8.

2.2 1-cocycles de degré de Gauss 1

2.2.1 Les différents mouvements de flèches

A partir d'une configuration de Gauss à point triple, on produit d'autres configurations de Gauss à point triple en utilisant certains mouvements de flèches afin d'obtenir un **poids** I^1_i (c'est-à-dire une somme formelle de configurations de Gauss à point triple que l'on évaluera à terme sur des diagrammes de Gauss à point triple) qui interviendra dans le calcul d'un 1-cocycle Γ. Ces différents mouvements sont détaillés dans [11].

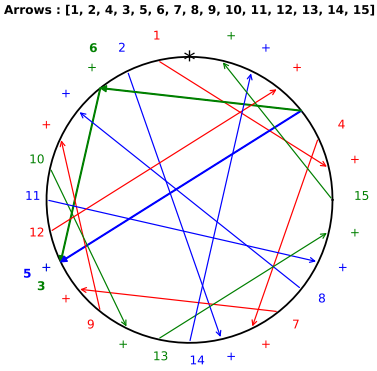
Sous-configurations interdites

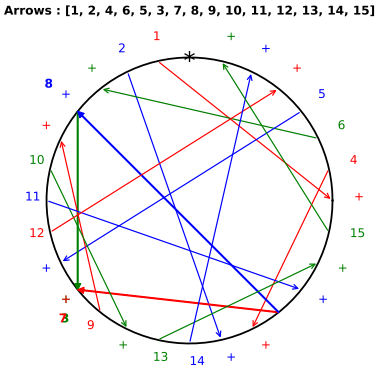
Avant de définir les différents mouvements, on dresse une liste de sous-configurations qui sont interdites dans une configuration de Gauss. Puisque l'on travaille avec des tresses dans le tore solide, il y a des sous-configurations qui n'apparaissent jamais dans un diagramme de Gauss d'une tresse dans le tore solide. Tout d'abord, du fait que l'on travaille dans le tore solide, en étudiant les lacets standards que l'on a associés à chaque croisement, les marquages sont conditionnés entre eux et certaines sous-configurations sont de ce fait interdites. D'autre part, puisque l'on travaille avec des tresses, les lacets standards ne peuvent être d'homotopie 0. Ainsi, si l'on a une sous-configuration de deux flèches non croisées comme illustrée dans la figure 2.9, il faut nécessairement $0 < p < q < n$ où n est toujours le nombre de brins. Evidemment, cette contrainte peut être prolongée si l'on "change" un croisement par son opposé : cela revient à changer le sens d'une flèche et remplacer son marquage p par $n-p$. On dresse ainsi la liste des sous-configurations interdites dans la figure 2.10 pour $n = 4$.

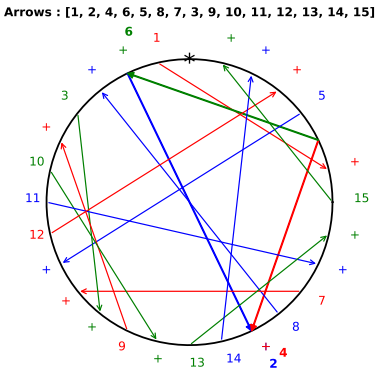
Définition 2.2.1. Une **configuration interdite** est une configuration qui contient une des sous-configurations interdites définies précédemment.

Le mouvement "TriSlide move" pour une flèche par rapport au triangle

Ce mouvement pour une flèche concerne uniquement les flèches qui ne sont pas dans le triangle. On va "slider" un bout de la flèche (tête ou pied) à travers le triangle à condition de ne pas obtenir de configuration interdite. La flèche ne peut slider que si l'une de ses extrémités est directement à côté d'un sommet du triangle. Ces mouvements interviennent dans l'étude de $\Sigma^{(2)}_{quad}$ en ce qui concerne l'invariance. Ils permettent ainsi de s'assurer de l'invariance lorsque l'on effectue deux mouvements de Reidemeister III simultanément et que l'on obtient un point quadruple. On illustre ce mouvement sur un exemple dans la figure 2.11.







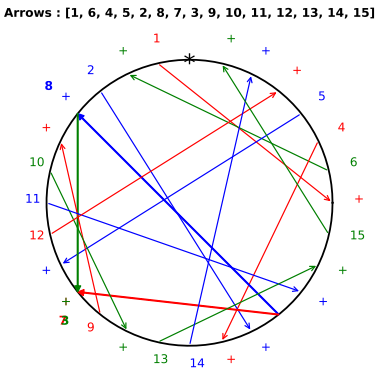
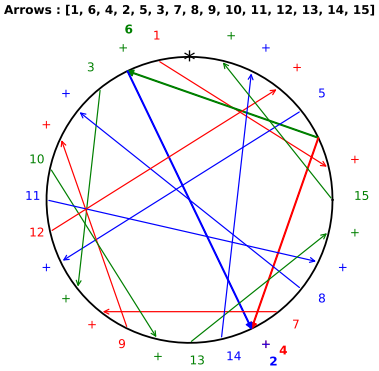
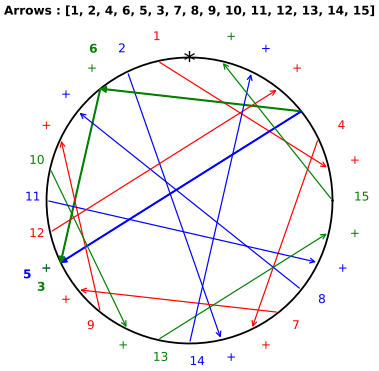
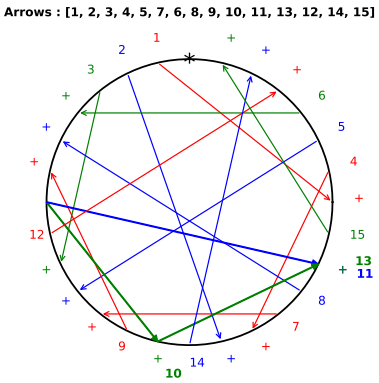


FIGURE 2.6 – Les 12 diagrammes de Gauss à point triple qui interviennent dans $rot(S_{exemple})$







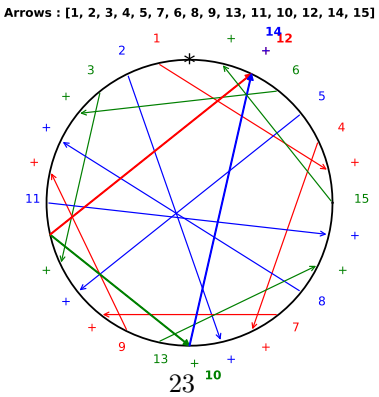


FIGURE 2.7 – Les 12 diagrammes de Gauss à point triple qui interviennent dans $rot(S_{exemple})$ (suite)

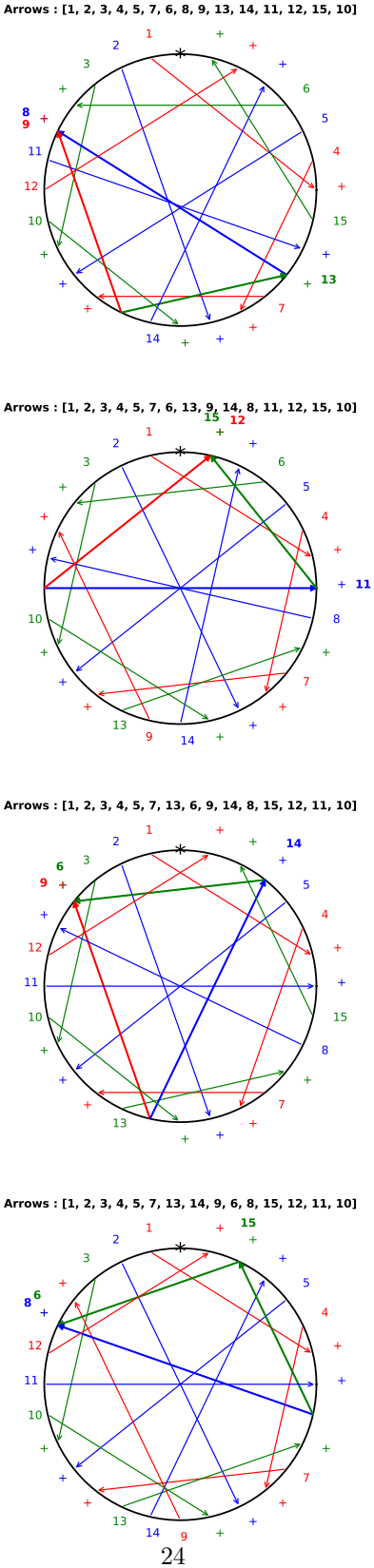


FIGURE 2.8 – Les 12 diagrammes de Gauss à point triple qui interviennent dans $rot(S_{exemple})$ (fin)

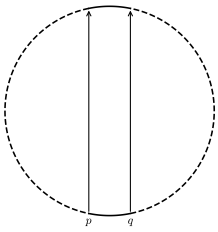


FIGURE 2.9 – Sous-configuration de deux flèches non croisées

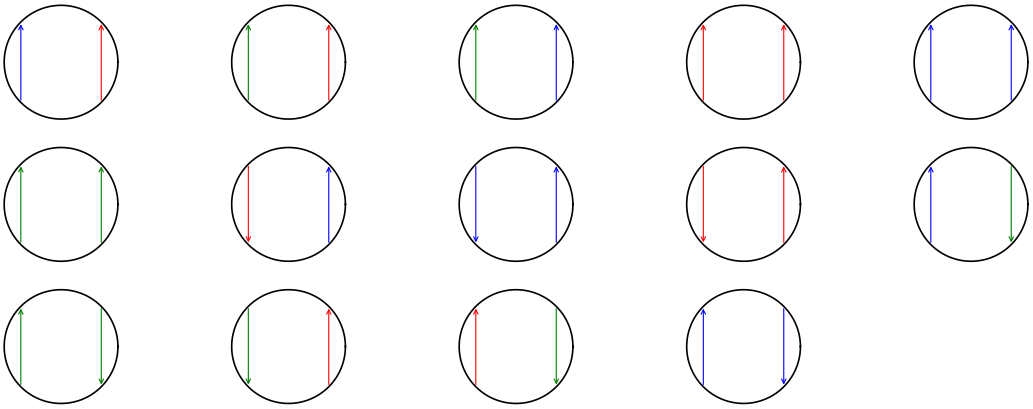


FIGURE 2.10 – Les sous-configurations de 2 flèches non croisées interdites pour $n = 4$

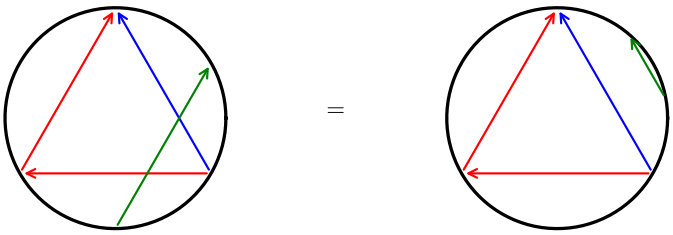


FIGURE 2.11 – Exemple d'un TriSlide move

Le mouvement "Exchange move"

Ce mouvement est un échange entre une flèche qui n'est pas dans le triangle et une flèche du triangle qui ont le même marquage. L'échange n'est possible que si les deux flèches sont dans

une position de Reidemeister Move II (flèches croisées). Ces mouvements interviennent dans l'étude de $\Sigma^{(2)}_{trans-self}$ et permettent de s'assurer de l'invariance lorsqu'on effectue un mouvement de Reidemeister III et un mouvement de Reidemeister II simultanément et que l'on obtient un point d'autotangence avec une branche transverse. Puisque les contributions arrivent avec des signes opposés dans $\Sigma^{(2)}_{trans-self}$, la configuration générée par un Exchange move doit être ajoutée avec un coefficient $-\lambda$ par rapport au coefficient λ de la configuration de départ. On illustre ce mouvement sur un exemple dans la figure 2.12.

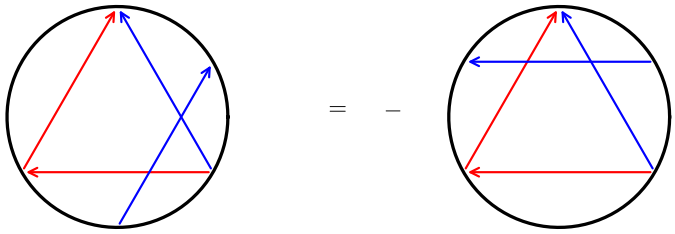


FIGURE 2.12 – Exemple d'un Exchange move

2.2.2 Poids des 1-cocycles

On note $W = \{W_i^1\}$ la liste des poids construits à partir des différents moves définis dans la sous-section précédente. Le programme en annexe permet d'en dresser la liste complète pour le cas des 4-tresses dans les figures 2.13 et 2.14 mais également pour $n > 4$. Le détail de l'algorithme sera précisé dans le chapitre 4.

Exemple. On va évaluer W^1_{10} pour p un diagramme de Gauss à point triple qui intervient dans $rot(S_{exemple})$. On le rappelle dans la figure 2.15. Ici, il y a exactement 4 flèches rouges (donc de classe d'homotopie 1) : 1, 4, 9 et 12. En additionnant les contributions de chacune, on obtient $W^1_{10}(p) = 1 + 1 - 1 + 1 = 2$

2.2.3 Invariance et 1-cocycles de degré 1

Théorème 2.2.1. *Soit γ une n-tresse fermée dans le tore solide. On définit :*

$$\Gamma_i(rot(\gamma)) = \sum_{p \text{ de type } type(W_i^1)} sign(p) x^{W_i^1(p)}$$

avec $W_i^1 \in W$ Alors $\Gamma_i(rot(.))$ est un invariant pour les n-tresses nodales fermées dans le tore solide.

Démonstration. Comme pour la preuve du théorème 2.1.1, on veut montrer que Γ **vaut 0 sur les méridiens de**

$$\Sigma^{(2)} = \Sigma^{(2)}_{quad} \cup \Sigma^{(2)}_{trans-self} \cup \Sigma^{(2)}_{self-flex} \cup \Sigma^{(2)}_{inter}$$

.

On commence par une réduction de cas afin de simplifier la preuve :

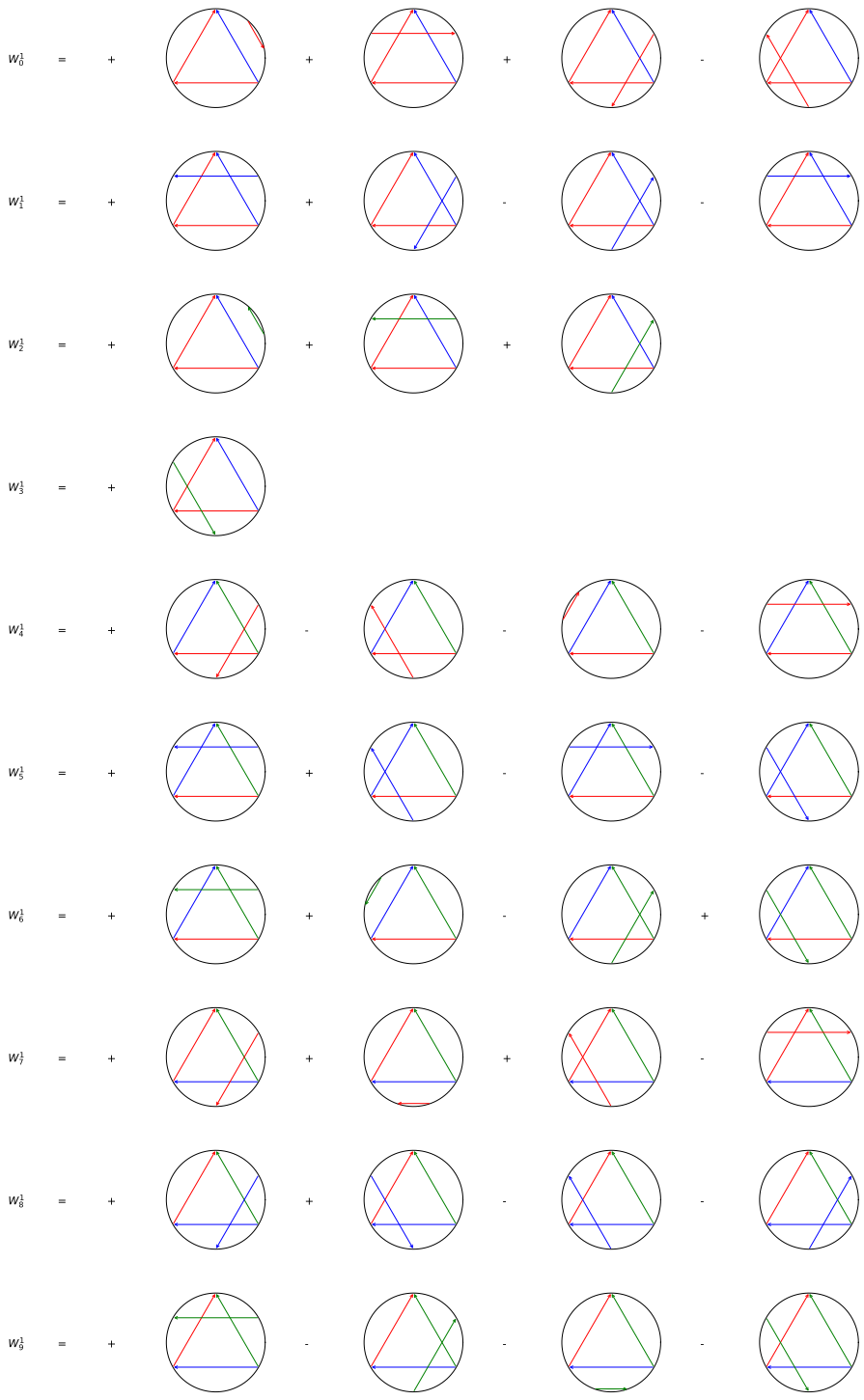


FIGURE 2.13 – Liste des poids de degré 1

FIGURE 2.14 – Liste des poids de degré 1 (fin)

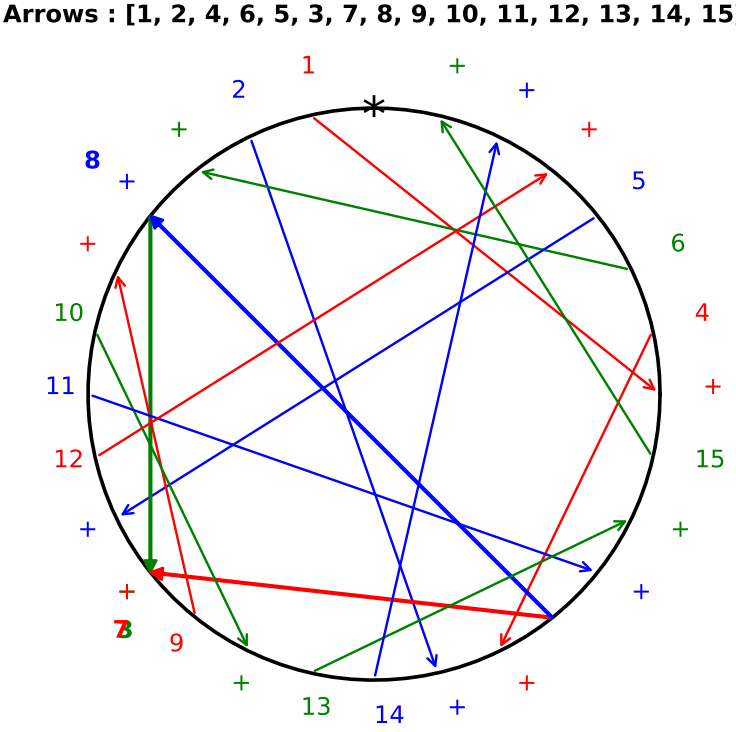


FIGURE 2.15 – Le diagramme de Gauss à point triple de l'exemple

— Pour $\Sigma^{(2)}_{quad}$, si l'on a :

  * $\Gamma(m) = 0$ pour un type local de méridien m de $\Sigma^{(2)}_{quad}$;
  * $\Gamma(m) = 0$ pour tous les méridiens m de $\Sigma^{(2)}_{trans-self}$;
  * $\Gamma(m) = 0$ pour tous les méridiens m de $\Sigma^{(2)}_{inter}$;

  Alors, par l'étude de $\Sigma^{(3)}_{trans-trans-self}$, on peut déduire que $\Gamma(m) = 0$ pour tous les types de méridiens de $\Sigma^{(2)}_{quad}$.

— Pour $\Sigma^{(2)}_{trans-self}$, si l'on a :

  * $\Gamma(m) = 0$ pour un type local de $\Sigma^{(2)}_{trans-self}$;
  * $\Gamma(m) = 0$ pour tous les méridiens de $\Sigma^{(2)}_{self-flex}$;
  * $\Gamma(m) = 0$ pour tous les méridiens m de $\Sigma^{(2)}_{inter}$;

  Alors, par l'étude de $\Sigma^{(3)}_{trans-self-flex}$, on peut déduire que $\Gamma(m) = 0$ pour tous les types de méridiens de $\Sigma^{(2)}_{trans-self}$.

— Pour $\Sigma^{(2)}_{inter}$, par l'étude de $\Sigma^{(1)} \cap \Sigma^{(2)}_{trans-self}$, on peut se réduire à l'étude de Γ sur des points triples avec uniquement des croisements positifs et des points d'autotangence d'un seul des deux types.

On commence donc l'étude de $\Sigma^{(2)}$ par $\Sigma^{(2)}_{inter} = (\Sigma^{(1)}_{tri} \cap \Sigma^{(1)}_{tri}) \cup (\Sigma^{(1)}_{tri} \cap \Sigma^{(1)}_{tan}) \cup (\Sigma^{(1)}_{tan} \cap \Sigma^{(1)}_{tan})$:

— Pour $\Sigma^{(1)}_{tri} \cap \Sigma^{(1)}_{tri}$, puisque qu'on peut trouver deux sphères disjointes qui contiennent les deux points triples, si l'on considère les flèches de l'un des points triples, leurs positions relatives au triangle formé par l'autre point triple ne change pas. Autrement dit, leurs contributions s'annulent le long du méridien (car I^1_i ne contient que des configurations de point triple à 1 flèche) et on a que Γ vaut 0.

— Pour $\Sigma^{(1)}_{tri} \cap \Sigma^{(1)}_{tan}$, le point triple p de $\Sigma^{(1)}_{tri}$ pourrait apparaître dans le calcul Γ. Concernant les deux flèches de $\Sigma^{(1)}_{tan}$, leurs contributions au poids $I^1_i(p)$ est soit 0 pour les deux, soit elles s'annulent puisqu'elles sont de signes opposés.

— Les points de $\Sigma^{(1)}_{tan} \cap \Sigma^{(1)}_{tan}$ n'apparaissent pas dans le calcul de I^1_i donc ne contribuent pas.

Ainsi, on a $\Gamma(m) = 0$ pour tous les méridiens m de $\Sigma^{(2)}_{inter}$.

Concernant l'étude de $\Sigma^{(2)}_{self-flex}$, comme les points d'autotangences ne contribuent pas, on a $\Gamma(m) = 0$ **pour tous les méridiens** m **de** $\Sigma^{(2)}_{self-flex}$.

Il y a exactement 6 possibilités de points pour $\Sigma^{(2)}_{trans-self}$. Chacune contient exactement deux types locaux de points triples qui sont toujours du même type global. Le long d'un méridien, les deux points triples apparaissent avec des signes opposés et de plus, les deux diagrammes de Gauss diffèrent exactement par un exchange move. Également, le signe de la flèche qui n'est pas dans le triangle change d'un diagramme à l'autre donc Γ vaut 0 sur le méridien. **Ainsi, on a $\Gamma(m) = 0$ pour tous les méridiens m de $\Sigma^{(2)}_{trans-self}$.**

Grâce à la réduction de cas, on peut considérer, sans perte de généralité, qu'un point de $\Sigma^{(2)}_{quad}$ n'est constitué que de croisements positifs. Le long d'un méridien de $\Sigma^{(2)}_{quad}$, il y a exactement 8 points triples qui apparaissent par paires avec des signes opposés. Les différents cas sont résumés dans le tableau de la figure 2.16. Comme chaque diagramme de Gauss d'une paire diffère de l'autre diagramme par un Trislide move, on a que Γ **vaut 0 le long du méridien de** $\Sigma^{(2)}_{quad}$. □

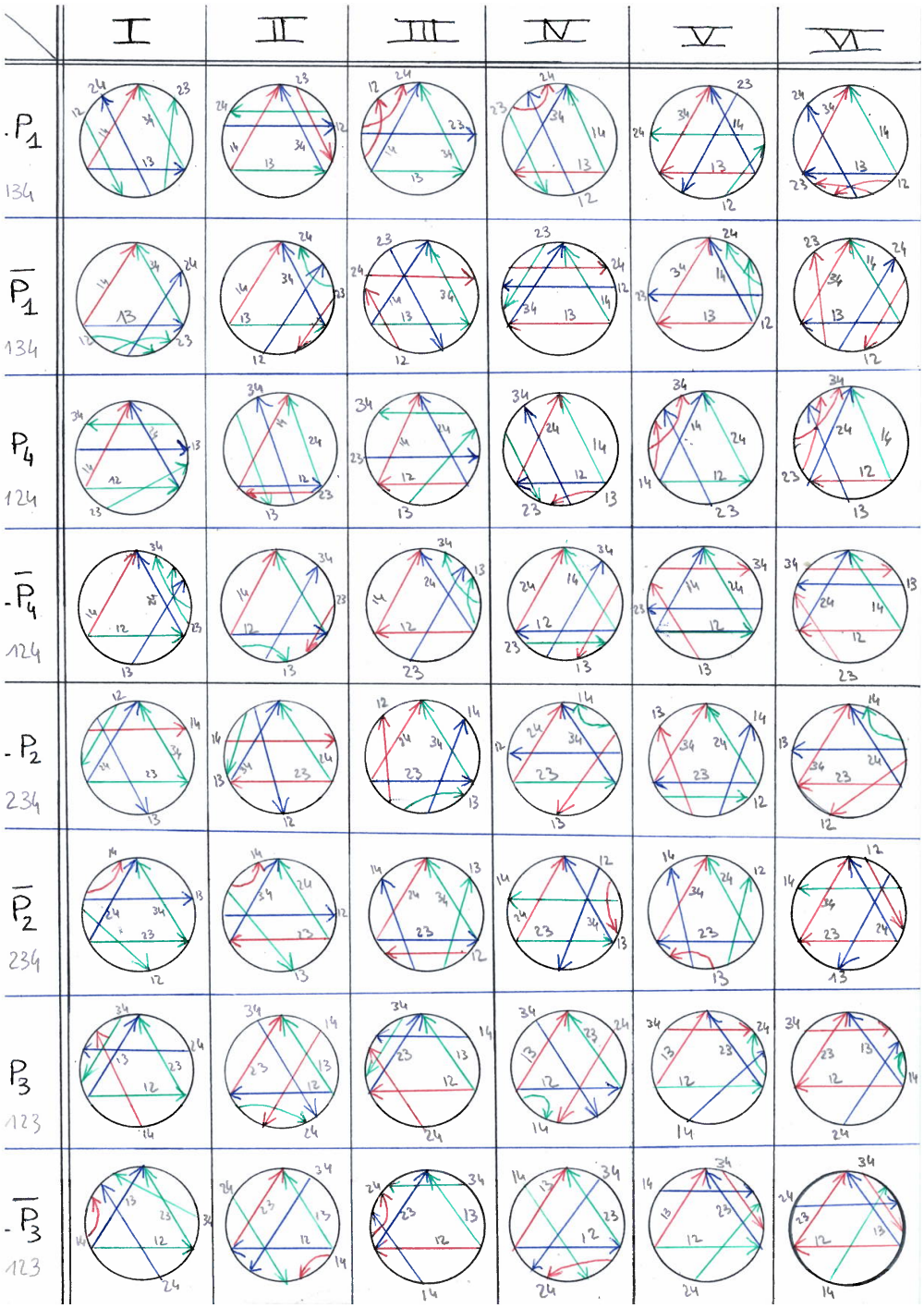


FIGURE 2.16 – Les différents cas dans la résolution de $\Sigma^{(2)}_{quad}$

Exemple. En reprenant les diagrammes de Gauss à point triple présentés dans les figures 2.6, 2.7 et 2.8, on peut calculer :

— $\Gamma_{10}(rot(S_{exemple})) = -x^2 + x^2 + x^2 - x^2 = 0$,
— $\Gamma_{11}(rot(S_{exemple})) = -x^0 + x^0 + x^0 - x^0 = 0$,
— $\Gamma_{12}(rot(S_{exemple})) = -x^{-2} + x^{-2} + x^{-2} - x^{-2} = 0$,
— $\Gamma_{13}(rot(S_{exemple})) = -x^2 + x^2 + x^2 - x^2 = 0$,
— $\Gamma_{14}(rot(S_{exemple})) = -x^0 + x^0 + x^0 - x^0 = 0$,
— $\Gamma_{15}(rot(S_{exemple})) = -x^2 + x^2 + x^2 - x^2 = 0$,
— $\Gamma_{16}(rot(S_{exemple})) = -x + x + x - x = 0$,
— $\Gamma_{17}(rot(S_{exemple})) = -x^4 + x^4 + x^4 - x^4 = 0$,
— $\Gamma_{18}(rot(S_{exemple})) = -x^0 + x^0 + x^0 - x^0 = 0$,
— $\Gamma_{19}(rot(S_{exemple})) = -x^3 + x^3 + x^3 - x^3 = 0$,
— $\Gamma_k(rot(S_{exemple})) = 0$ pour les autres k.

Chapitre 3

Améliorations et résultats pour $n = 4$

Dans ce chapitre, nous ne nous intéresserons qu'à des 1-cocycles de degré 1 pour des **4-tresses nodales fermées dans le tore solide**. On commence par affiner les invariants du chapitre 2. On compare ensuite les invariants de type fini dans le **tore solide** qui sont des invariants pour la conjugaison de tresses (= isotopie des noeuds dans le tore solide) avec nos invariants.

3.1 Améliorations des invariants

3.1.1 Amélioration des poids

Les poids du chapitre 2 sont au nombre de 20. On voudrait en obtenir plus afin d'avoir une famille d'invariants plus fine. Pour cela, on va modifier l'exchange move : au lieu de rajouter une configuration avec un coefficient -1, on rajoute un terme de correction. Plus précisément le terme de correction est égal à :

$$\frac{1}{2}(w(a) - 1), \text{ où } a \in \{d, hm, ml\}$$

On choisit a en fonction de la flèche du triangle qui est échangée. Cela donne exactement 36 nouveaux poids qui correspondent ainsi à 36 nouveaux 1-cocycles $\{\Gamma_i^{1,aff}\}_i$.

Proposition 3.1.0.1. *$I_i^{aff} = \Gamma_i^{1,aff}(rot(.))$ est un invariant pour les tresses fermées dans le tore solide pour tout $i \in \{0, ..., 35\}$.*

Démonstration. La preuve de l'invariance ne diffère de celle du théorème 2.2.1 que pour $\Sigma^{(2)}_{trans-self}$, mais le nouvel exchange move défini permet de s'assurer que nos 1-cocycles s'annulent sur les méridiens de $\Sigma^{(2)}_{trans-self}$. □

On dresse ainsi la liste complète dans les figures 3.1, 3.2, 3.3 et 3.4.

Exemple. Pour rappel, on a : $S_{exemple} = \sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1\sigma_1\sigma_2\sigma_3\sigma_1\sigma_2\sigma_1$. En reprenant les diagrammes de Gauss à point triple présentés dans les figures 2.6, 2.7 et 2.8, on peut calculer :

— $\Gamma_{18}^{1,aff}(rot(S_{exemple})) = -x^3 + x^3 + x^3 - x^3 = 0$,

— $\Gamma_{19}^{1,aff}(rot(S_{exemple})) = -x^1 + x^1 + x^1 - x^1 = 0$,

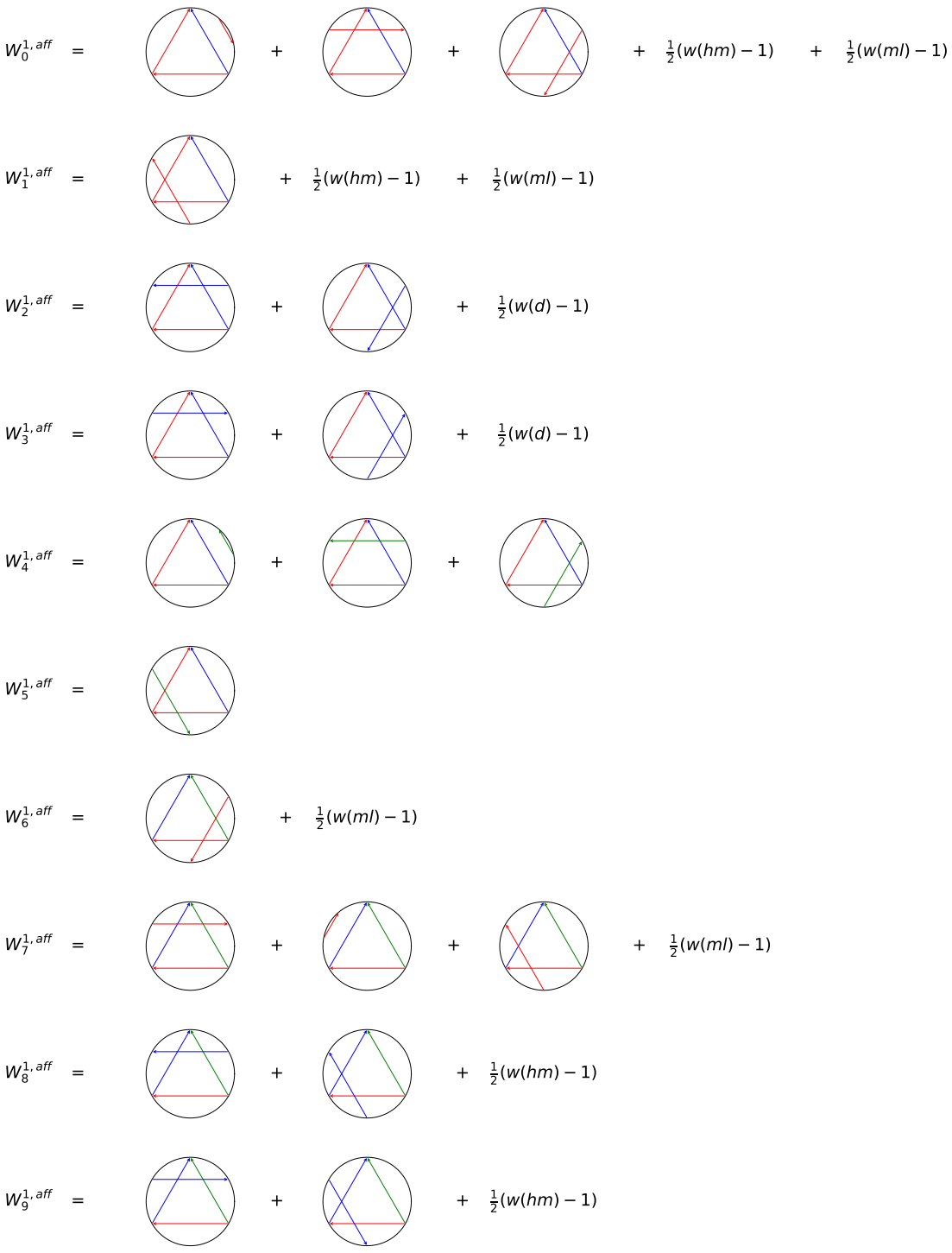


FIGURE 3.1 – Liste des poids de degré 1 affinés

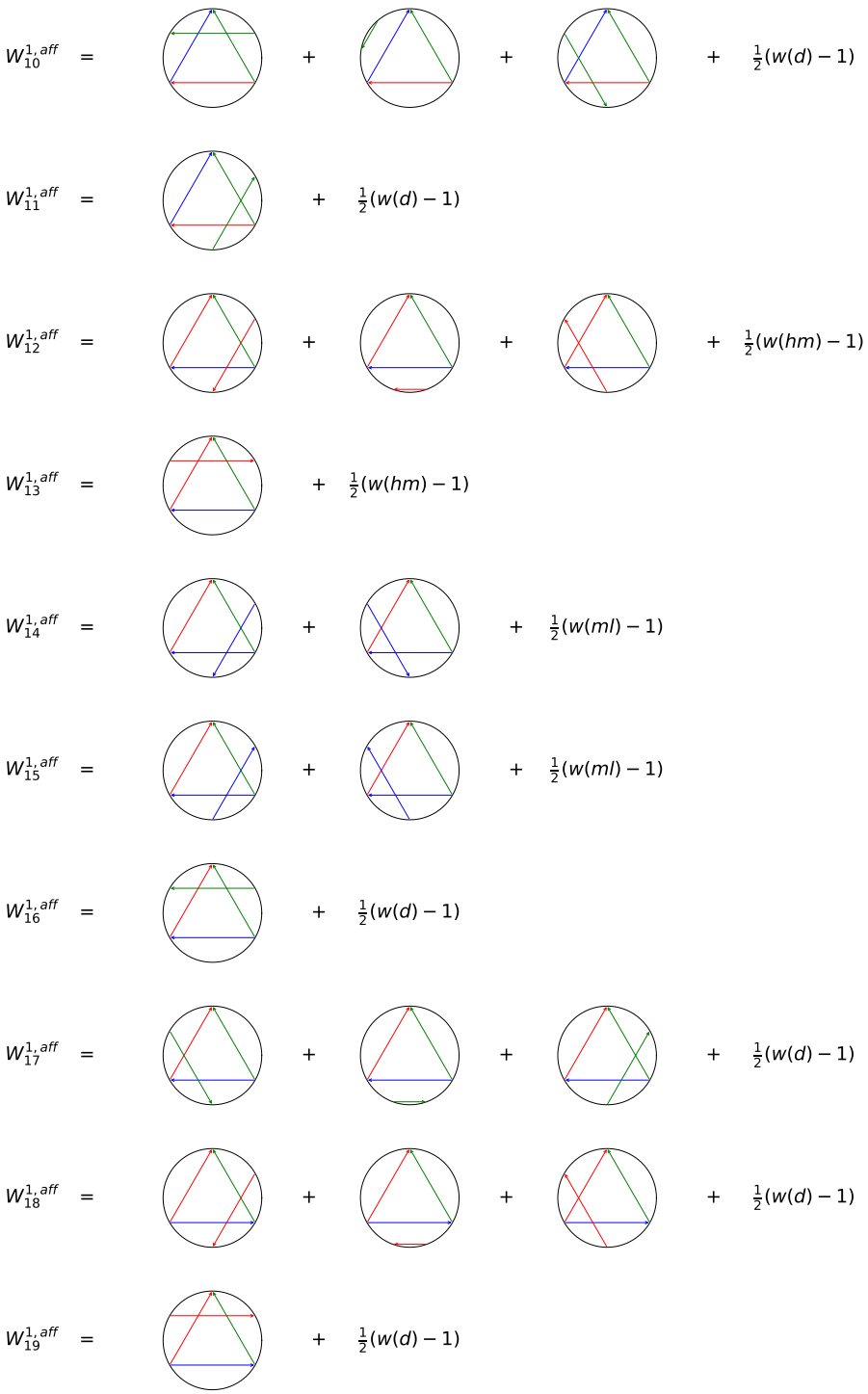


FIGURE 3.2 – Liste des poids de degré 1 affinés (suite)

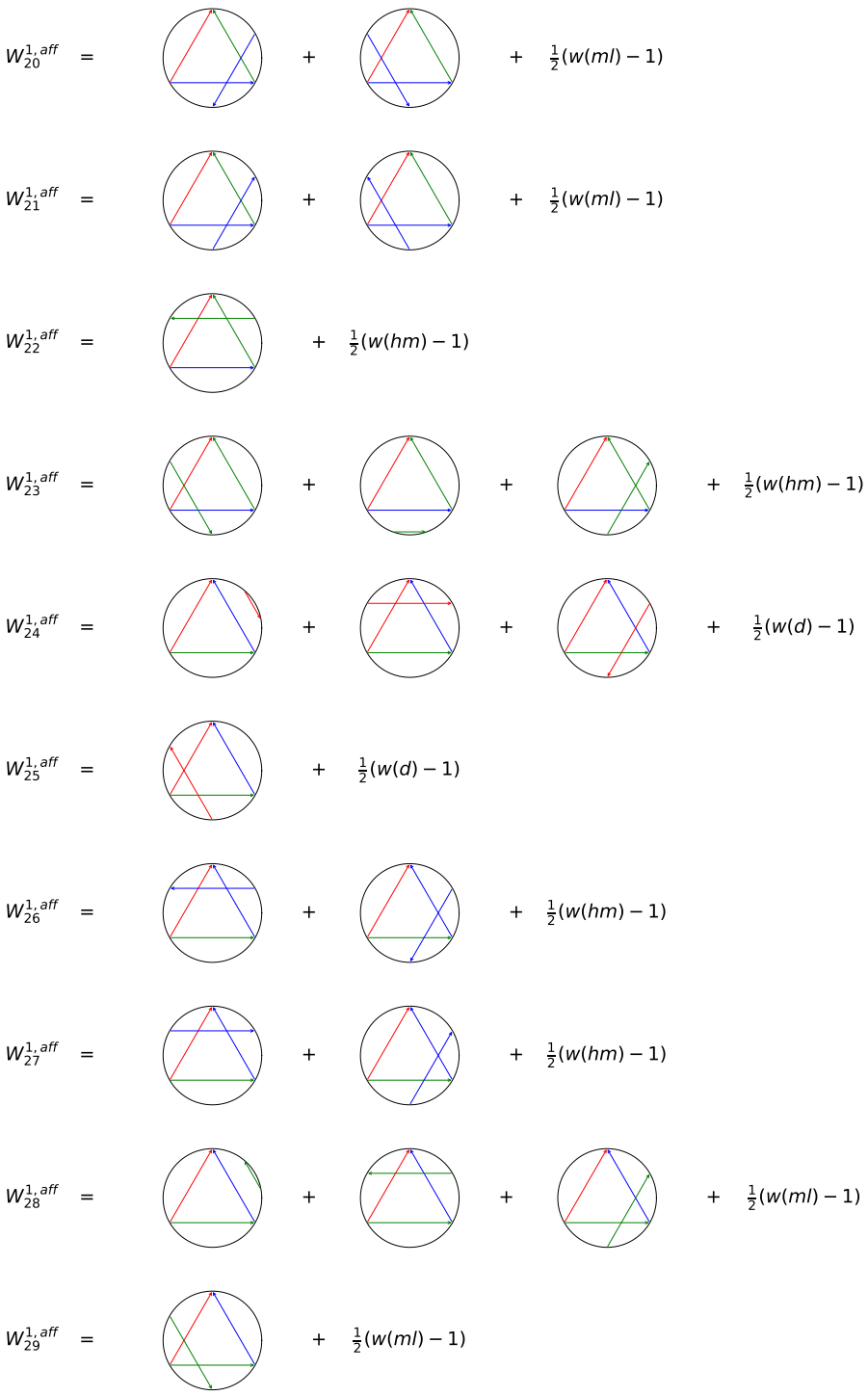


FIGURE 3.3 – Liste des poids de degré 1 affinés (suite 2)

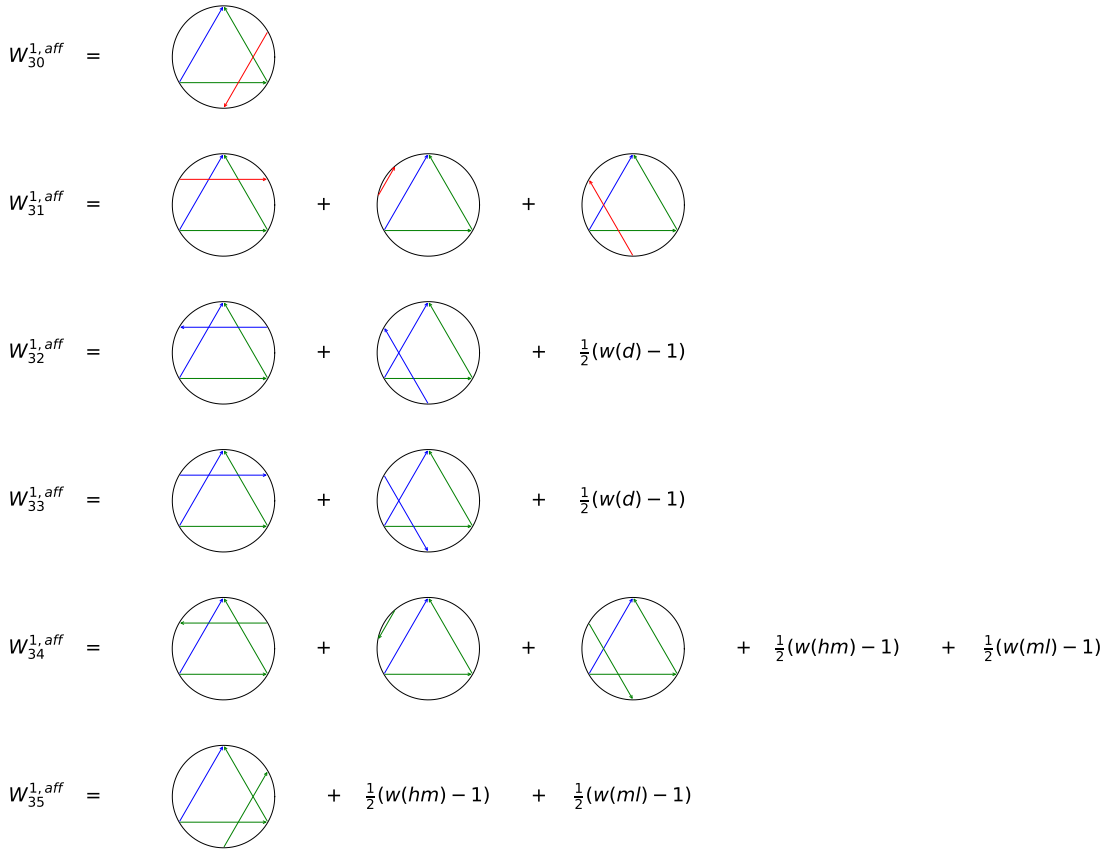


FIGURE 3.4 – Liste des poids de degré 1 affinés (fin)

3.1.2 Une nouvelle forme pour les 1-cocycles

Dans cette sous-section, on garde les mêmes poids affinés que la sous-section précédente. Puisqu'il y en a 6 pour les 6 types, on peut définir une nouvelle forme pour les 1-cocycles en s'inspirant de [9].

Définition 3.1.1. On considère, pour une tresse γ :

$$\Gamma^{1,multi}_{(a,b)^\epsilon}(rot(\gamma)) = \sum_{p \text{ de type } (a,b)^\epsilon} sign(p) x_0^{W^{1,aff}_{6i}} x_1^{W^{1,aff}_{6i+1}} x_2^{W^{1,aff}_{6i+2}} x_3^{W^{1,aff}_{6i+3}} x_4^{W^{1,aff}_{6i+4}} x_5^{W^{1,aff}_{6i+5}}$$

Proposition 3.1.0.2. $\Gamma^{1,multi}_{(a,b)^\epsilon}$ *est un invariant pour les tresses fermées dans le tore solide pour tout type* $(a,b)^\epsilon$.

Démonstration. Il suffit de reprendre la preuve du théorème 2.2.1 et de lire la partie du théorème 3.1.0.1. □

Remarque. Les restrictions à une variable (en posant $x_i = 1$ pour toutes sauf une) de cette famille de 6 invariants redonnent les 1-cocycles de la sous-section précédente.

Conjecture 1. La famille de ces 6 invariants est plus fine que la famille des 36 invariants de la sous-section précédente.

3.2 Invariants de type fini

Il existe des invariants pour la conjugaison de tresses (= isotopie des noeuds dans le tore solide) qui sont des invariants de type fini. On va donc étudier ces invariants afin de les comparer aux notres ensuite. Dans cette section, on définit l'espace des formules de Gauss homogènes à 2 flèches pour les 4-tresses fermées dans le **tore solide**. On fournira également une estimation de sa dimension à partir d'un algorithme et on comparera cet espace à la famille de 1-cocycles de degré 1 du chapitre 2.

3.2.1 L'espace des formules de Gauss homogènes à 2 flèches

Définition 3.2.1. Une configuration à 2 flèches marquées est la donnée d'un cercle avec 2 flèches qui sont marquées par 1, 2 ou 3.

On définit l'évaluation d'une configuration à 2 flèches marquées C sur un diagramme de Gauss GD comme suit : l'évaluation de C sur GD est égale à $< C, GD >= \sum_{(a,b)} w(a) \times w(b)$ où la somme porte sur les couples (a,b) de flèches de GD qui correspondent à C.

On note alors $\mathcal{C}_2$ le $\mathbb{R}$-espace vectoriel des combinaisons linéaires formelles de configurations à 2 flèches marquées. On peut noter que la dimension de $\mathcal{C}_2$ est égale au nombre total de configurations différentes à 2 flèches marquées. On aimerait définir une formule de Gauss homogène à 2 flèches comme un élément de $\mathbb{C}_2$ qui est un invariant pour les 4-tresses nodales fermées dans le tore solide. On va ainsi définir un sous-espace vectoriel $\mathcal{I}_2$ de $\mathcal{C}_2$ constitué uniquement d'invariants.

Premièrement, puisque les éléments de $\mathcal{I}_2$ sont des invariants de 4-tresses, il y a des configurations qui n'apparaissent jamais en tant que sous-diagramme d'un diagramme de Gauss d'une 4-tresse fermée nodale dans le tore solide. On liste ces configurations dans la figure 3.5.

On demandera ainsi, en terme de coordonnées, que les coefficients devant les configurations de $\mathcal{R}_i$ soient tous nuls pour les éléments de $\mathcal{I}_2$.

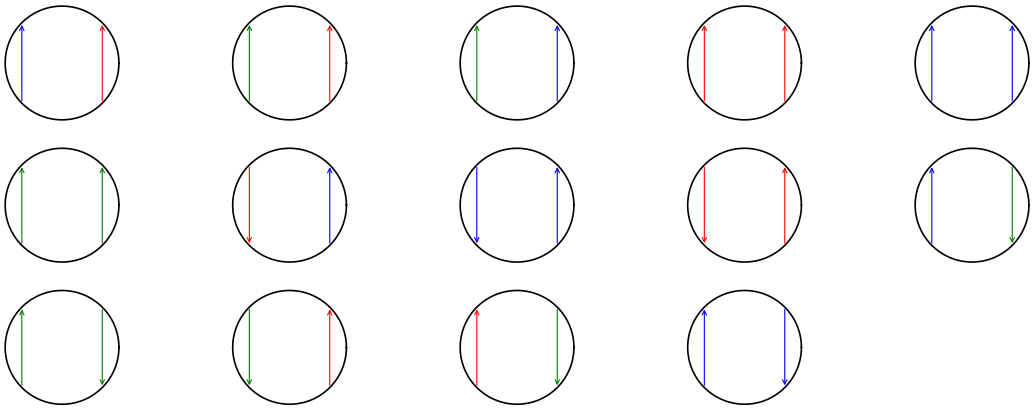


FIGURE 3.5 – $\mathcal{R}_i$

Ensuite, les différents mouvements de Reidemeister donnent également des conditions sur les coefficients. Plus précisément, on veut que les coefficients devant les configurations de $\mathcal{R}_{II}$ (présentées dans la figure 3.6) soient tous nuls. Et les configurations de $\mathcal{R}_{III}$ définies dans la figure 3.7 doivent satisfaire les relations suivantes :

$$Coef(C^c_{(2,1)}) + Coef(C^c_{(1,2)}) + Coef(C^o_{(1,1)}) = 2Coef(C^c_{(1,2)}) \quad \text{(I)}$$

$$Coef(C^c_{(3,2)}) + Coef(C^c_{(2,3)}) + Coef(C^o_{(3,3)}) = 2Coef(C^c_{(2,3)}) \quad \text{(II)}$$

$$Coef(C^c_{(3,1)}) + Coef(C^c_{(1,2)}) + Coef(C^o_{(2,3)}) = Coef(C^p_{(1,3)}) + Coef(C^p_{(1,2)}) + Coef(C^c_{(3,2)}) \quad \text{(III)}$$

$$Coef(C^c_{(2,1)}) + Coef(C^c_{(1,3)}) + Coef(C^o_{(2,3)}) = Coef(C^p_{(1,2)}) + Coef(C^p_{(1,3)}) + Coef(C^c_{(2,3)}) \quad \text{(IV)}$$

$$Coef(C^c_{(3,2)}) + Coef(C^c_{(1,3)}) + Coef(C^o_{(1,2)}) = Coef(C^p_{(2,3)}) + Coef(C^p_{(1,3)}) + Coef(C^c_{(1,2)}) \quad \text{(V)}$$

$$Coef(C^c_{(3,1)}) + Coef(C^c_{(2,3)}) + Coef(C^o_{(1,2)}) = Coef(C^p_{(1,3)}) + Coef(C^p_{(2,3)}) + Coef(C^c_{(2,1)}) \quad \text{(VI)}$$

Ainsi, on peut définir le **sous-espace vectoriel** $\mathcal{I}_2$ de $\mathcal{C}_2$ constitué des combinaisons linéaires :

— qui ne contiennent aucune configuration de $\mathcal{R}_I$ et de $\mathcal{R}_{II}$;

— qui respectent les relations (I) à (VI).

Il est connu que $\mathcal{I}_2$ est constitué d'invariants de type 2 (voir [6]).

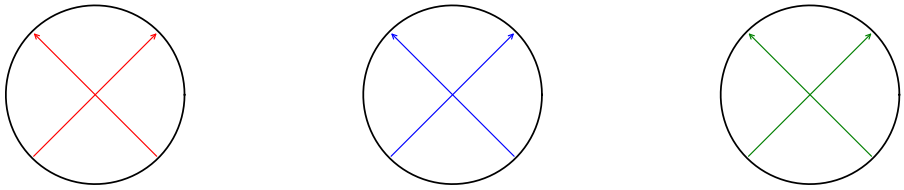


FIGURE 3.6 – $\mathcal{R}_{II}$

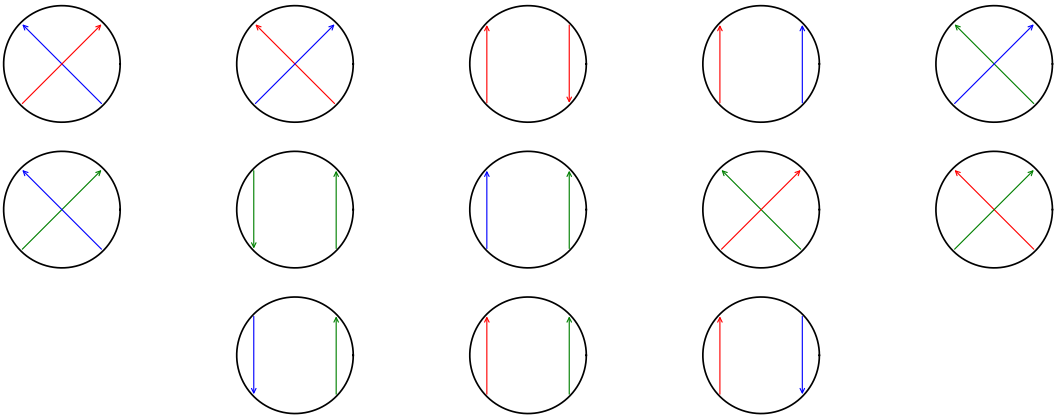


FIGURE 3.7 – Dans le sens de la lecture : $C^c_{(2,1)}$, $C^c_{(1,2)}$, $C^o_{(1,1)}$, $C^p_{(1,2)}$, $C^c_{(3,2)}$, $C^c_{(2,3)}$, $C^o_{(3,3)}$, $C^p_{(2,3)}$, $C^c_{(3,1)}$, $C^c_{(1,3)}$, $C^o_{(2,3}$), $C^p_{(1,3)}$, $C^o_{(1,2)}$

3.2.2 Etude de la dimension de $\mathcal{I}_2$

On va étudier la dimension de $\mathcal{I}_2$ sous deux angles différents. D'abord en tant que sous-espace vectoriel constitué de combinaisons linéaires formelles et ensuite en tant que sous-espace vectoriel d'applications sur les 4-tresses nodales fermées dans le tore solide. Pour différencier les deux points de vue, on notera $\mathcal{I}_2^{form}$ pour le premier et $\mathcal{I}_2^{Gauss}$ pour le second.

Définition 3.2.2. Une **formule de Gauss** est un élément de $\mathcal{I}_2^{Gauss}$.

Dimension de $\mathcal{I}_2^{form}$

En considérant les relations qui définissent $\mathcal{I}_2$, on obtient un système linéaire que l'on peut résoudre afin d'obtenir une base de $\mathcal{I}_2$ et sa dimension.

Proposition 3.2.0.1. *En tant que sous-espace vectoriel de combinaisons linéaires formelles de configurations à 2 flèches marquées, $\mathcal{I}_2$ est de dimension 8. Une base est donnée dans la figure 3.8.*

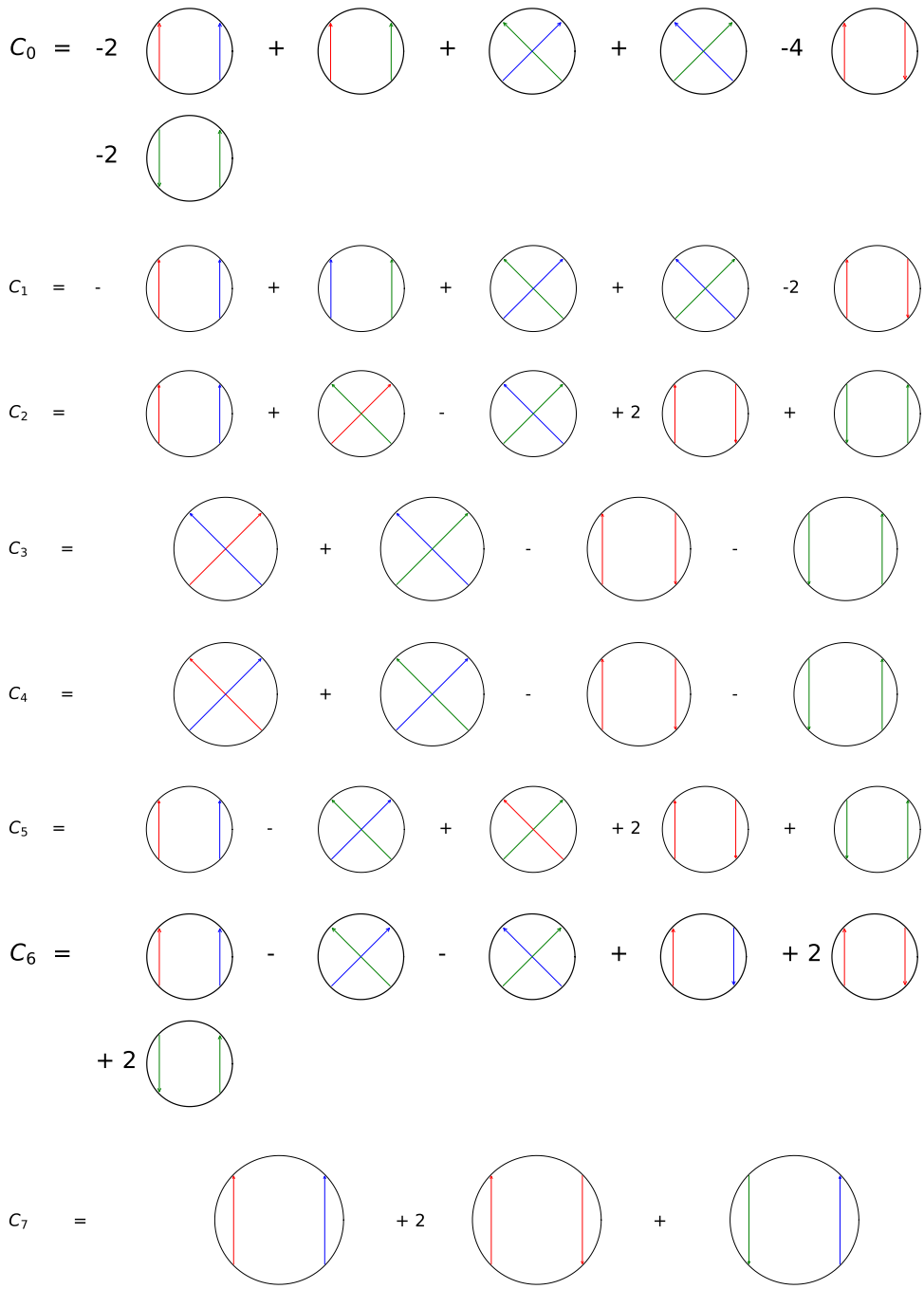


FIGURE 3.8 – Base pour $\mathcal{I}_2^{form}$

Démonstration. Il suffit de résoudre le système d'équations donné par les relations de $\mathcal{R}_{III}$. □

Dimension de $\mathcal{I}_2^{Gauss}$

Sur les milliers de tests, la dimension estimée de $\mathcal{I}_2$ est de 3. Ainsi, en considérant les Formules de Gauss de la figure 3.8, il nous faudrait donc montrer 5 relations (en tant qu'applications sur les 4-tresses nodales fermées dans le tore solide). Malheureusement, nous n'avons réussi à n'en montrer que 2.

Pour commencer, on va passer par des Formules de Gauss à 1 flèche marquée pour montrer une première relation. On définit dans la figure 3.9 pour cela W_1, W_2 et W_3 qui sont bien invariants par mouvements de Reidemeister.

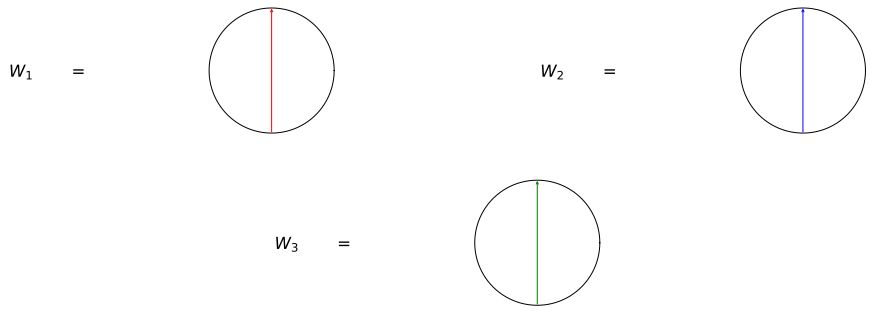


FIGURE 3.9 – Formules de Gauss à 1 flèche marquée

Proposition 3.2.0.2. $W_1 = W_3$

Démonstration. On pose $I = W_1 - W_3$ et on va montrer que I est un invariant homotopique. Si l'on considère des homotopies, alors un croisement est homotope à son image miroir. Ainsi, pour qu'une formule de Gauss F soit un invariant homotopique, il faut et il suffit que la contribution de tout croisement à F soit la même que la contribution de son image miroir. En terme de diagrammes de Gauss, si un croisement est d'un signe ϵ et de marquage α, son image miroir sera de signe $-\epsilon$ et de marquage $4-\alpha$. On voit facilement que I est invariant par homotopie en faisant la vérification sur les 3 types possibles de marquage.

Comme I est un invariant homotopique, il est constant sur toutes les 4-tresses nodales fermées dans le tore solide. En effet, elles sont toutes homotopes à la tresse $[1, 2, 3]$ dont le diagramme de Gauss est représenté dans la figure 3.10. Ainsi, puisque $I([1, 2, 3]) = 0$, on a $I = 0$. □

A partir de la relation $W_1 = W_3$, on en déduit la relation :

$$W_1 W_2 = W_3 W_2$$

Or, chaque terme de cette égalité peut-être réécrit, en regardant les contributions, comme une formule de Gauss à 2 flèches marquées. Plus précisément, on a les égalités de la figure 3.11

Ainsi, on a démontré une première relation ce qui permet de d'affirmer $dim(I_2^{Gauss}) \leq 7$.

Pour démontrer la seconde relation, on va utiliser la même méthode en passant par un invariant homotopique.

Proposition 3.2.0.3. *Soit I la formule de Gauss définie dans la figure 3.12. Alors $I = 0$.*

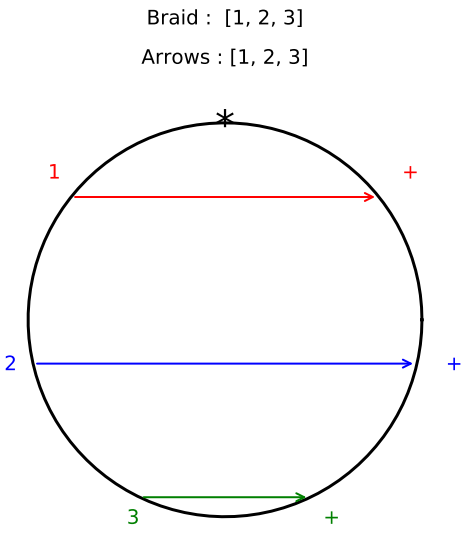


FIGURE 3.10 – Diagramme de Gauss pour la 4-tresse $[1, 2, 3]$

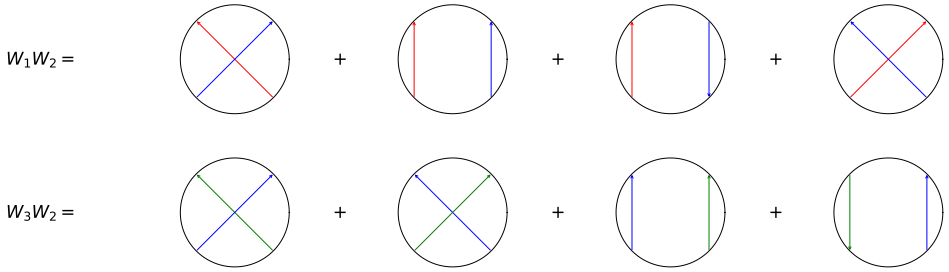


FIGURE 3.11 – W_1W_2 et W_3W_2

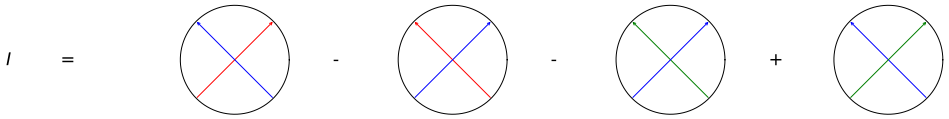


FIGURE 3.12 – Une nouvelle formule de Gauss

Démonstration. Comme précédemment, on montre que I est un invariant homotopique et qu'il vaut 0 sur la tresse $[1,2,3]$. □

Ainsi, avec cette deuxième relation (qui n'est pas linéairement dépendante de la première), on a démontré la proposition suivante.

Proposition 3.2.0.4. $dim(I_2^{Gauss}) \leq 6$

3.2.3 Etude des 1-cocycles combinatoires

Définition 3.2.3. On définit les 1-cocycles combinatoires I_i^{comb} par :

$$I_i^{comb} = \frac{d\Gamma_i^{1,aff}(rot(.))}{dx}(1), \forall i \in \{0, ..., 35\}$$

Remarque. Cela revient à considérer les poids raffinés définis dans la section précédente. Cette écriture sous forme de dérivée permet surtout d'extraire facilement ces invariants dans le programme.

Proposition 3.2.0.5. *I_6^{comb}, I_7^{comb} et I_8^{comb} sont linéairement dépendants.*

Démonstration. Il suffit de calculer I_6^{comb}, I_7^{comb} et I_8^{comb} sur des exemples pertinents.

Pour la tresse $S_1 = [-1,2,3,1,2,3,1,2,1,1,2,3,1,2,1,1,2,3,1,2,1,1,2,3,1,2,1,1,2,3,1,2,1,1,2,3,1,2,1,1,2,3,1,2,1]$, on a :

— $I_6^{aff}(S_1) = -x^2$

— $I_7^{aff}(S_1) = -x^9$

— $I_8^{aff}(S_1) = -x^6$

Pour la tresse $S_2 = [3,2,2,3,1,2,3,1,2,3,1,2,3,1,2,3,2,1,2,1,2,3,1,2,1]$, on a :

— $I_6^{aff}(S_2) = -2$

— $I_7^{aff}(S_2) = -2$

— $I_8^{aff}(S_2) = -2x$

Pour la tresse $S_3 = [-2,1,2,-1,2,3,3,2,3]$, on a :

— $I_6^{aff}(S_2) = -x$

— $I_7^{aff}(S_2) = -x^6$

— $I_8^{aff}(S_2) = -x^4$

Ainsi, en étudiant la matrice $M = \begin{pmatrix} -2 & 0 & -1 \\ -9 & 0 & -6 \\ -6 & -2 & -4 \end{pmatrix}$ On obtient bien que I_6^{comb}, I_7^{comb} et I_8^{comb} sont linéairement dépendants. □

Ainsi, à partir de cette proposition, on peut affirmer (en supposant que $dim(I_2^{Gauss}) = 3$) que les 1-cocycles combinatoires sont au moins aussi fins, voire plus, que les formules de Gauss de I_2^{Gauss}.

Théorème 3.2.1. *Les 1-cocycles combinatoires I_i^{comb} sont plus fins que les invariants de I_2^{Gauss}.*

Exemple. Les tresses $S_1 = [3, -2, 1, 3, -2, 3, 1]$ et $S_2 = [1, -3, 2]$ (illustrées dans les figures 3.13 et 3.14) ont les mêmes images par les invariants de I_2^{Gauss} :

$$[2, 1, -1, 1, 1, -1, -2, -1]$$

Cependant, elles sont différenciées par les 1-cocycles combinatoires :

$S_1 : [0, 0, 0, 0, 0, 0, 2, 1, -3, -3, 1, 2, -1, -2, 3, 3, -2, -1, 1, 2, -3, -3, 2, 1, -1, -2, 3, 3, -1, -2, 0, 0, 0, 0, 0, 0]$

$S_2 : [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, -1, 0, 0, -1, 0, 0, 1, 0, 0, 1, 0, 0, -1, 0, 0, 0, -1, 0, 0, 0, 0, 0, 0]$

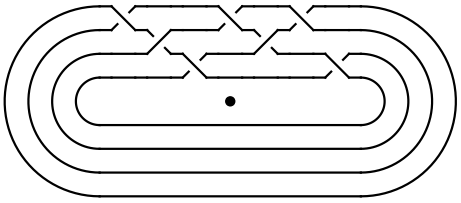


FIGURE 3.13 – $S_1 = [3, -2, 1, 3, -2, 3, 1]$

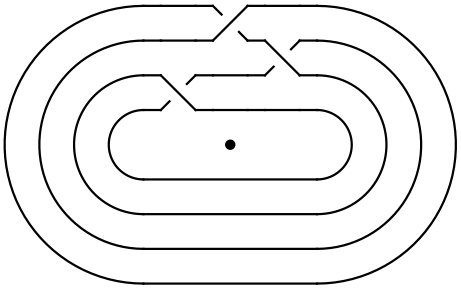


FIGURE 3.14 – $S_2 = [1, -3, 2]$

3.3 Perspectives et questionnement

3.3.1 Tresses non inversibles

Définition 3.3.1. Une **tresse inversible** (en tant que noeud) est une tresse qui est isotope à son image miroir (son inverse).

Un des problèmes que l'on s'est posé a été : est-ce que nos différentes familles d'invariants affinés permettent de différencier une tresse non inversible de son inverse. La réponse est : non pour les tresses qui ont au plus 6 croisements (toutes les tresses possibles ont été testées). Egalement, nous avons lancé de très nombreux tests (plusieurs centaines voire milliers) sur des tresses aléatoires dont la longueur était aléatoire aussi. L'expérience n'a rien donné. On peut donc supposer que les différentes familles d'invariants affinés ne permettent pas de distinguer les tresses non inversibles.

3.3.2 Degré supérieur

Une section entière est dédiée à la construction de 1-cocycles de degré supérieur dans l'annexe A. On y détaille un mouvement et des sous-configurations interdites supplémentaires et on dresse la liste pour $n = 4$.

3.3.3 Avec des 1-cocycles améliorés ?

On pourrait alors se demander si la famille des 1-cocycles de degré 3 ou si une amélioration de la famille des 1-cocycles de degré 2 permettrait de distinguer les tresses non inversibles.

Pour la première possibilité, un manque de puissance de calcul nous empêche d'effectuer des tests concluants. Quant à la seconde possibilité, elle mériterait un peu plus d'approfondissement. En effet, le terme de correction remplaçant un exchange move pour une configuration à au moins 2 flèches est plus complexe que pour les configurations à 1 flèche. En effet, quand une flèche est échangée, il faut prendre en compte la contribution des flèches restantes. Un candidat possible (et sa généralisation aux autres poids) pourrait être celui illustré dans la figure 3.15. Le principe est de "factoriser" par la flèche du triangle qui est échangé et la sous-configuration restante est obtenue en enlevant la flèche régulière qui est censée être échangée. Mais alors, il y a à nouveau des mouvements de flèches dans ces sous-configurations qui ont une flèche de moins.

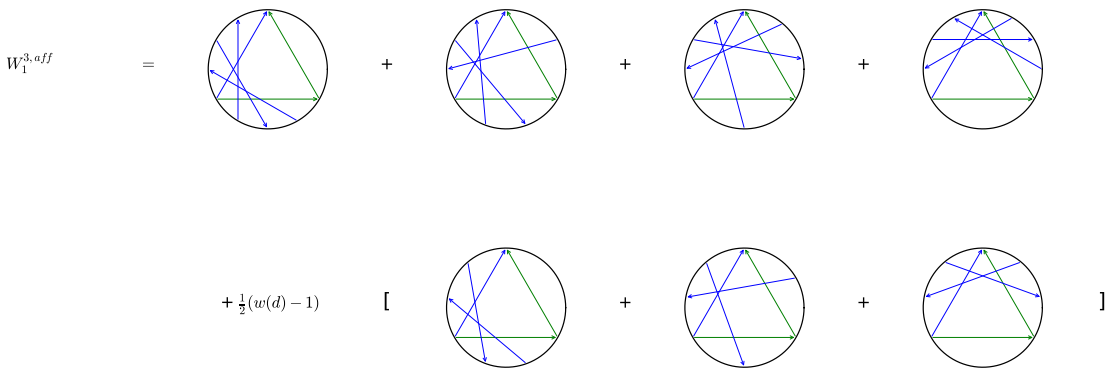


FIGURE 3.15 – Un exemple de poids de degré 3 amélioré

3.3.4 Généralisation à des n-tresses ?

Le programme en annexe a été pensé pour s'adapter pour faire les calculs pour des tresses nodales à n brins. Un lecteur avisé pourra donc généraliser l'étude de cette thèse à des n-tresses nodales fermées dans le tore solide. Les mouvements qui conduisent à la génération des poids puis des invariants restent les mêmes, et la preuve de l'invariance est une preuve générale qui n'utilise pas le cadre de $n = 4$. Ce qui changera cependant sera le nombre de types possibles pour une configuration à point triple. Un coup d'oeil rapide au programme en annexe permet de connaître les différentes fonctions à modifier pour les adapter à un n général. Une section de l'annexe est dédié à cette généralisation.

Proposition 3.3.0.1. *Dans le cadre de l'étude des n-tresses nodales fermées dans le tore solide, il y a exactement* $(n-1)(n-2)$ *types de configurations à point triple.*

Démonstration. On sépare l'étude des types en fonction de leurs types globaux sans marquage (positif ou négatif)

Pour une configuration de type négatif, on a l'égalite suivante :

$$[d] = [hm] + [ml]$$

De plus, on sait $[d], [hm], [ml] \in \{1, ..., n-1\}$. Alors, pour $[hm] = 1$, on a $[ml] \in \{1, ..., n-2\}$ et de manière générale, on a :

$$\text{Pour } [hm] = j, [ml] \in \{1, ..., n-j-1\}, j \in \{1, ..., n-2\}$$

Ainsi, on peut affirmer que le nombre de types négatifs de configurations à point triple pour des n-tresses nodales dans le tore solide est de :

$$\sum_{k=1}^{n-2} k = \frac{(n-1)(n-2)}{2}$$

De la même manière, on montre qu'il y a $\frac{(n-1)(n-2)}{2}$ types positifs de configurations à point triple pour des n-tresses : on a la relation :

$$[hm] = [d] + n - [ml]$$

Et pour $[d] = j, j \in \{1, ..., n-2\}$, $[ml] \in \{j+1, ..., n-1\}$.

□

Chapitre 4

Explications des algorithmes

Ce chapitre représente la majeure partie de ce travail de thèse. L'implémentation de ce programme a nécessité de s'autoformer à Python et à la Programmation Orientée Objet principalement. Mais le temps passé à coder ce programme était nécessaire afin de proposer un outil performant et utilisable facilement. Chaque bout de code a été minutieusement commenté, détaillé et le format notebook a permis de découper le code en plusieurs parties indépendantes qui remplissent chacune une fonction bien précise. On détaille ainsi dans ce chapitre le fonctionnement de chacun de ces parties.

4.1 Evaluer un 1-cocycle pour une tresse fermée

L'idée générale de la méthode pour calculer un 1-cocycle est intuitive. On la résume dans l'algorithme 1. Il est cependant nécessaire de détailler certaines étapes. Notamment, dans les sous-sections suivantes, on détaillera le calcul du diagramme de Gauss pour une tresse fermée dans le tore solide, l'application du lacet *rot* pour une tresse et la méthode pour détecter un sous-diagramme correspondant à une configuration du 1-cocycle.

Algorithme : Calculer un 1-cocycle pour une tresse fermée S
Entrées : S
Sorties : Cocycle
// Traduire S en Diagramme de Gauss
$D \leftarrow Diagramme\_de\_Gauss(S)$;
// Initialiser le calcul du cocycle
$Cocycle \leftarrow 0$;
// Pousser le lacet *rot* à D
pour *tous les diagrammes singuliers Ds résultant de* $rot(S)$ *(RM3)***faire**
    // Evaluer l'invariant pour Ds
    $Cocycle \leftarrow Cocycle$+Evaluer la combinaison linéaire de configurations sur Ds ;
fin

Algorithme 1 : Calculer un 1-cocycle pour une tresse fermée S

4.1.1 Diagramme de Gauss pour les tresses fermées dans le tore solide

Calculer le marquage d'une flèche

Afin de calculer le diagramme de Gauss pour une tresse dans le tore solide, il est nécessaire de calculer le marquage d'une flèche et donc la classe d'homotopie du lacet standard associé à un croisement. Or, un croisement dans une tresse fermée correspond à un générateur de B_4. On commence d'abord par calculer la classe d'homotopie du lacet standard associé au dernier générateur. Celle-ci dépend uniquement de la valeur du générateur et de la permutation opérée sur les brins par la tresse entière. Le calcul est résumé dans l'algorithme 2

4.1.2 Le lacet canonique *rot*

Appliquer le lacet *rot* à une tresse fermée est équivalent à appliquer Δ^2 au travers de notre tresse :

$\gamma \to \gamma\Delta^2\Delta^{-2} \to \Delta\gamma'\Delta\Delta^{-2} \to \Delta^2\gamma\Delta^{-2} = \Delta^{-2}\Delta^2\gamma = \gamma$

Or, le passage de Δ au travers d'une tresse peut être découpé pour chaque élément de la tresse et ainsi être traduit en termes de mouvements de Reidemeister.

Définition 4.1.1. Passage de Δ à travers les générateurs de B_4

$$1\Delta = 112\underline{31}21 \overset{CM}{=} 1\underline{121}321 \xrightarrow{\text{RM3}} 121\underline{232}1 \xrightarrow{\text{RM3}} 12\underline{13}231 \overset{CM}{=} 1231\underline{231} \overset{CM}{=} 1231213 = \Delta 3$$

$$\bar{1}\Delta = \underline{\bar{1}1}23121 \xrightarrow{\text{RM2 }\times 2} 1\bar{1}2\underline{31}21 \overset{CM}{=} 1\bar{1}\underline{21}321 \xrightarrow{\text{RM3}} 121\underline{\bar{2}32}1 \xrightarrow{\text{RM3}} 12\underline{13}2\underline{\bar{3}1} \overset{CM}{=} 1231213\bar{3} = \Delta\bar{3}$$

$$2\Delta = \underline{212}3121 \xrightarrow{\text{RM3}} 12\underline{13}121 \overset{CM}{=} 1231\underline{121} \xrightarrow{\text{RM3}} 1231212 = \Delta 2$$

$$\bar{2}\Delta = \underline{\bar{2}12}3121 \xrightarrow{\text{RM3}} 12\underline{\bar{1}3}121 \overset{CM}{=} 123\underline{\bar{1}1}21 \xrightarrow{\text{RM2 }\times 2} 1231\underline{\bar{1}21} \xrightarrow{\text{RM3}} 1231212\bar{2} = \Delta\bar{2}$$

$$3\Delta = \underline{31}23121 \overset{CM}{=} 1\underline{323}121 \xrightarrow{\text{RM3}} 123\underline{212}1 \xrightarrow{\text{RM3}} 1231211 = \Delta 1$$

$$\bar{3}\Delta = \underline{\bar{3}1}23121 \overset{CM}{=} 1\underline{\bar{3}23}121 \xrightarrow{\text{RM3}} 123\underline{\bar{2}12}1 \xrightarrow{\text{RM3}} 12312\underline{\bar{1}1} \xrightarrow{\text{RM2 }\times 2} 1231213\bar{1} = \Delta\bar{1}$$

Remarque. Lorsque l'on pousse Δ à travers une tresse **positive** (dont tous les croisements sont positifs), on n'utilise jamais de mouvement de Reidemeister II. Ce sont ces mouvements de Reidemeister II qui font que le programme n'est utilisable que pour les tresses positives avec des versions de Sagemath postérieures à 8.1.

4.1.3 Détecter un sous-diagramme correspondant à une configuration

La problématique de cette sous-section est de détecter tous les sous-diagrammes extraits à partir d'un diagramme de Gauss singulier Ds (provenant d'un mouvement de Reidemeister III) qui correspondent à une configuration à point triple donnée C. Le premier point à tester pour la correspondance est de vérifier si Ds correspond au bon type de point triple. Ensuite, on va tester les m-uplets de flèches qui correspondent à C de manière récursive. La méthode du calcul

Algorithme : Déterminer le marquage de la flèche associée au dernier générateur σ d'une tresse S

Entrées : σ,S
Sorties : *Classe_homotopie*

Permutation ← Permutation opérée sur les brins par S;
si $\sigma = 1$ *ou* $\sigma = -1$**alors**
  si *Permutation = (1,2,3,4) ou Permutation = (1,2,4,3)***alors**
    Classe_homotopie ← 1;
  fin
  si *Permutation = (1,3,2,4) ou Permutation = (1,4,2,3)***alors**
    Classe_homotopie ← 2;
  fin
  si *Permutation = (1,3,4,2) ou Permutation = (1,4,3,2)***alors**
    Classe_homotopie ← 3;
  fin
fin
si $\sigma = 2$ *ou* $\sigma = -2$**alors**
  si *Permutation = (1,2,3,4) ou Permutation = (1,4,2,3)***alors**
    Classe_homotopie ← 1;
  fin
  si *Permutation = (1,2,4,3) ou Permutation = (1,3,4,2)***alors**
    Classe_homotopie ← 2;
  fin
  si *Permutation = (1,3,2,4) ou Permutation = (1,4,3,2)***alors**
    Classe_homotopie ← 3;
  fin
fin
si $\sigma = 3$ *ou* $\sigma = -3$**alors**
  si *Permutation = (1,2,3,4) ou Permutation = (1,3,4,2)***alors**
    Classe_homotopie ← 1;
  fin
  si *Permutation = (1,3,2,4) ou Permutation = (1,4,2,3)***alors**
    Classe_homotopie ← 2;
  fin
  si *Permutation = (1,2,4,3) ou Permutation = (1,4,3,2)***alors**
    Classe_homotopie ← 3;
  fin
fin
// Traiter les générateurs négatifs
si $\sigma < 0$**alors**
  Classe_homotopie ← $4 - Classe\_homotopie$;
fin

Algorithme 2 : Déterminer le marquage de la flèche associée au dernier générateur σ d'une tresse

de la contribution de Ds est résumée dans l'algorithme 3 et la détection récursive des m-uplets de flèches est détaillée dans l'algorithme 4. Elle repose essentiellement sur la récursivité et sur l'utilisation d'un dictionnaire qui traduit la correspondance entre les flèches de Ds et les flèches de C.

Algorithme : Evaluer la contribution d'un diagramme de Gauss singulier Ds pour une configuration à point triple C

Entrées : Ds, C
Sorties : $Contrib$

```
// Initialisation de la contribution
Contrib ← 0;
si Type_RM3(Ds) = Type_RM3(C) alors
    // Détection de tous les m-uplet de flèches qui correspondent à C de manière récursive
    List_m ← Test_recursif(Ds);
    // Calcul de la contribution des m-uplets de flèches de Ds
    pour chaque m-uplet de flèches A de List_m faire
        Contrib ← Contrib + Contribution(A);
    fin
fin
```

Algorithme 3 : Evaluer la contribution d'un diagramme de Gauss singulier Ds pour une configuration à point triple C

4.2 Générer tous les 1-cocycles de degré m

L'algorithme 5 choisi pour générer tous les 1-cocycles de degré m est plutôt naturel. On va cependant détailler les différents outils utilisés et les différents étapes de son exécution.

Remarque. Cet algorithme n'est pas optimal (notamment la partie où l'on génère toutes les configurations autorisées) et pourrait potentiellement être optimisé par un informaticien avisé.

4.2.1 Configurations autorisées

Puisque les configurations apparaissant dans les 1-cocycles seront évaluées sur des sous-diagrammes issus d'une tresse fermée dans le tore solide, il faut qu'elles représentent des situations "possibles". On dira qu'une configuration est *autorisée* si elle peut apparaître dans le diagramme d'une tresse fermée. On verra par la suite qu'il n'y a qu'un nombre fini de *types* possibles de mouvement de Reidemeister III.

Déterminer tous les types possibles de configurations de Gauss avec un croisement triple

Une configuration de Gauss avec un croisement triple correspond au point singulier lors d'un mouvement de Reideimeister III. Donc, tout comme dans $\mathbb{R}^3$, le croisement triple correspond à un "triangle de flèches". Alors, les règles concernant les marquages des flèches s'appliquent également au triangle de flèches. En considérant tous les marquages possibles pour les 3 flèches et en appliquant les contraintes d'être dans un tore solide, on obtient tous les types possibles de configurations de Gauss avec croisement triple.

Algorithme : Détecter les m-uplets de flèches d'un diagramme de Gauss singulier Ds qui correspondent à une contribution à point triple C

Entrées : Ds, C, n = nb de flèches
Sorties : $List\_m$

```
// On initialise les différentes variables
List_m ← [];
d, hm, ml ← fleches_du_triangle(Ds);
D, HM, ML ← fleches_du_triangle(C);
dictionnaire ← (d, hm, ml ↔ D, HM, ML);
diagramme_modele ← [D, HM, ML];
indice ← 0;
diagramme_test ← [d, hm, ml];
def test_recursif(diagramme_modele,diagramme_test,indice,dictionnaire):
    si indice =nb de flèches de C:
        List_m ← ajouter diagramme_test;
    sinon:
        fleche_modele ← C[indice];
        pour chaque flèche f qui est comme fleche_modele:
            diagramme_modele ← diagramme_modele + fleche_modele;
            diagramme_test ← diagramme_test + f;
            indice ← indice + 1;
            dictionnaire ← dictionnaire + (f ↔ fleche_modele);
            test_recursif(diagramme_modele, diagramme_test, indice, dictionnaire);
test_recursif(diagramme_modele, diagramme_test, indice, dictionnaire);
```

Algorithme 4 : Détecter les m-uplets de flèches d'un diagramme de Gauss singulier Ds qui correspondent à une contribution à point triple C

Algorithme : Générer toutes les combinaisons linéaires invariantes de configurations avec point triple pour les 1-cocycles

Entrées : m
Sorties : $List\_comblin$

```
// On initialise les différentes variables
Liste_comblin ← 0 ;
Liste_configurations ← 0;
pour tous les types de configurations possiblesfaire
    Liste_configurations ← +Toutes les configurations autorisées de m flèches;
fin
tant que toutes les configurations de Liste_configurations n'ont pas été traitéesfaire
    pour chaque configuration C de Liste_configurationsfaire
        Liste_comblin ← +CombLin(C);
    fin
fin
```

Algorithme 5 : Générer toutes les combinaisons linéaires invariantes de configurations avec point triple pour les 1-cocycles

Plus précisément, pour une n-tresse, il faut :

$$ml = d + n - hm, \text{ pour les configurations de signe positif,}$$

$$d = hm + ml, \text{ pour les configurations de signe négatif.}$$

Les différents types possibles pour le cas de 4-tresses sont résumés dans les figures 4.1 et 4.2

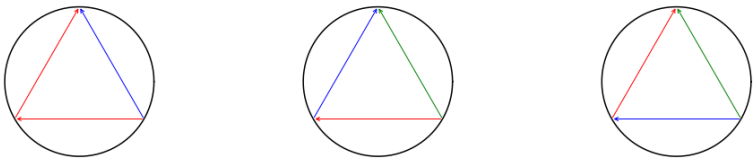


FIGURE 4.1 – Les 3 types de configurations négatives avec point triple pour les 4-tresses

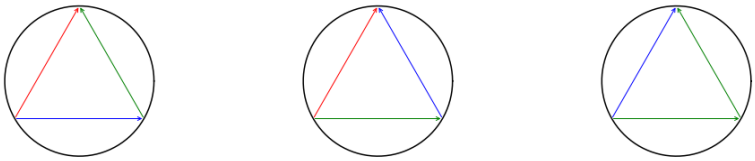


FIGURE 4.2 – Les 3 types de configurations positives avec point triple pour les 4-tresses

Contraintes héritées des tresses

On souhaite considérer des configurations de Gauss avec point triple autorisées. La question du type de point triple a été traitée ci-dessus. A présent, on s'intéresse aux flèches qui ne sont pas dans le triangle. Puisque les configurations autorisées représentent des sous-diagrammes de diagrammes de Gauss d'une tresse fermée dans le tore solide, elles héritent de contraintes liées au fait de travailler avec des tresses fermées dans le tore solide.

En particulier, considérer des sous-diagrammes d'un diagramme de Gauss d'une n-tresse implique qu'il ne doit y avoir **aucun** lacet de marquage 0 ou n dans la configuration. On traite le cas de $n = 4$ dans la figure 4.3.

4.2.2 Générer une combinaison linéaire invariante de configurations à partir d'une configuration M

A chaque mouvement d'une flèche, cela crée une nouvelle configuration. L'idée de l'algorithme 6 est donc d'effectuer tous les mouvements de toutes les flèches de toutes les configurations produites à partir de M :

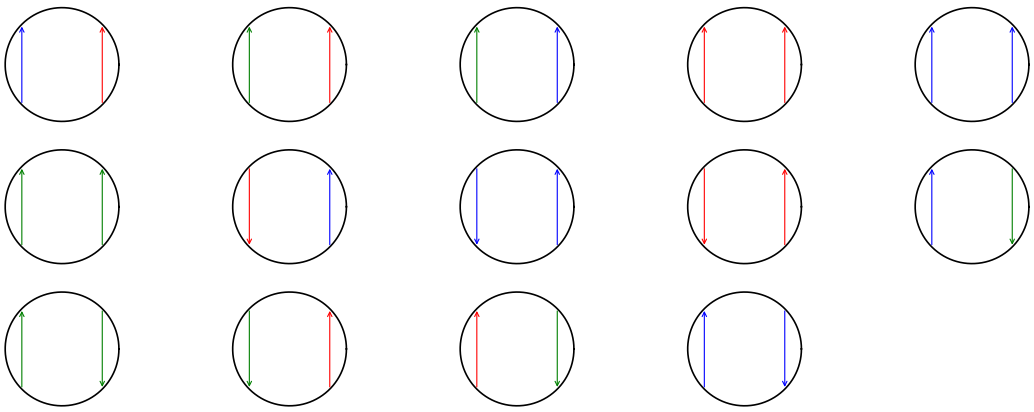


FIGURE 4.3 – Exemple de configurations interdites pour $n = 4$

Algorithme : Générer une combinaison linéaire invariante de configurations à partir d'une configuration M

Entrées : M
Sorties : $CombLin$
$CombLin \leftarrow M$;
tant que *toutes les configurations de* $CombLin$ *n'ont pas été traitées***faire**
  pour *chaque configuration* C *de* $CombLin$**faire**
    pour *chaque flèche* f *de* C**faire**
      $CombLin \leftarrow$ ajouter $Tous\_les\_SlideMoves(f)$;
      $CombLin \leftarrow$ ajouter $Tous\_les\_ExchangeMoves(f)$;
      $CombLin \leftarrow$ ajouter $Tous\_les\_TriSlideMoves(f)$;
    fin
  fin
fin

Algorithme 6 : Générer une combinaison linéaire invariante de configurations à partir d'une configuration M

Annexe A

Annexes

A.1 1-cocycles de degré supérieur

Ce qui a été fait dans le chapitre 2 peut être généralisé à des configurations à n flèches en rajoutant un mouvement et des sous-configurations interdites.

A.1.1 Nouveau mouvement et configurations interdites

Le mouvement "Slide move" pour un couple de flèches qui ne sont pas dans le triangle

Ce mouvement est essentiellement le même que le "TriSlide move" mais pour deux flèches qui ne sont pas dans le triangle. On demande l'invariance par ce nouveau mouvement afin de s'assurer de l'invariance des 1-cocycles par homotopies à travers la strate $\Sigma^{(1)}_{tri} \cap \Sigma^{(1)}_{tri}$. On en donne un exemple dans la figure A.1

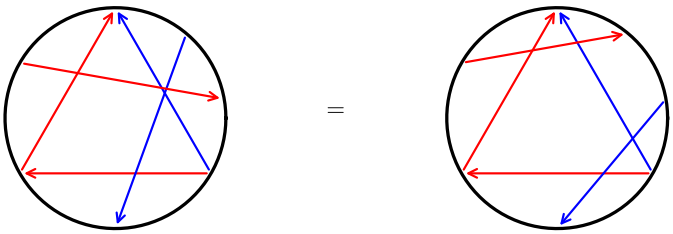


FIGURE A.1 – Exemple d'un Slide move

Sous-configurations interdites supplémentaires

A présent puisque l'on s'intéresse à créer des invariants, ils doivent être en particulier invariants par mouvement de Reidemeister II. Afin que les 1-cocycles soient invariants par homotopies à travers la strate $\Sigma^{(1)}_{tri} \cap \Sigma^{(1)}_{tan}$, on rajoute des sous-configurations interdites. Les poids ne doivent contenir aucune sous-configuration avec un couple de flèches qui correspondrait à un mouvement de Reidemeister II. Ainsi, on s'interdit des sous-configurations de deux flèches croisées de marquage

identique comme illustrée dans la figure A.2. Ainsi, pour $n = 4$, il n'y a que trois sous-configurations interdites, comme présenté dans la figure A.3.

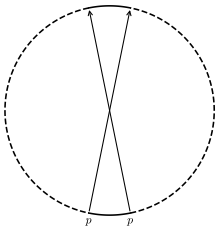


FIGURE A.2 – Sous-configuration de deux flèches croisées

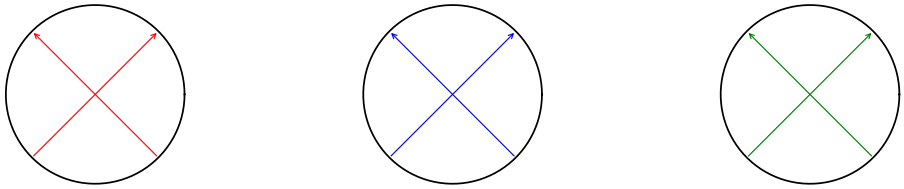


FIGURE A.3 – Les sous-configurations de 2 flèches croisées interdites pour $n = 4$

A.1.2 Exemples avec des 1-cocycles de degré 2

A partir des règles établies quant à la génération de 1-cocycles de degré supérieur, on implémente un algorithme dressant la liste de tous les poids pour les 1-cocycles de degré 2 pour des 4-tresses dans le tore solide. Une partie de la liste est illustrée dans les figures A.4 à A.21. Le détail de l'algorithme sera précisé dans le chapitre 4.

A.2 Généralisation à $n > 4$

Le programme joint en annexe est en parti pensé pour s'adapter à un n quelconque même s'il a surtout été utilisé pour $n = 4$. Concrètement, pour qu'il soit utilisable à $n > 4$, il faudrait reprendre la partie de calcul des BraidData et le calcul du lacet rot.

Dans cette sous-section, on dresse la liste des poids de degré 1 affinés pour $n = 5$. Le marquage d'homotopie 4 a été illustré en orange.

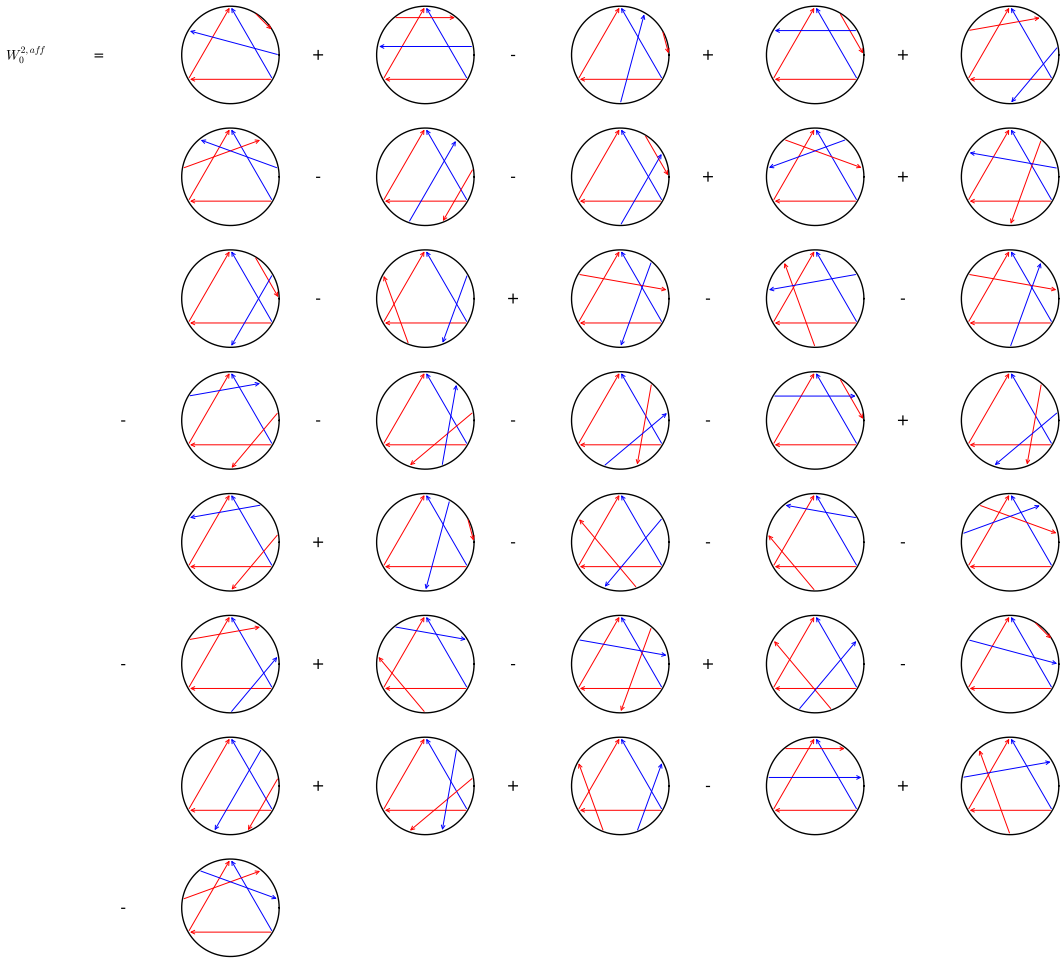


FIGURE A.4 – Liste des poids de degré 2 pour $n = 4$

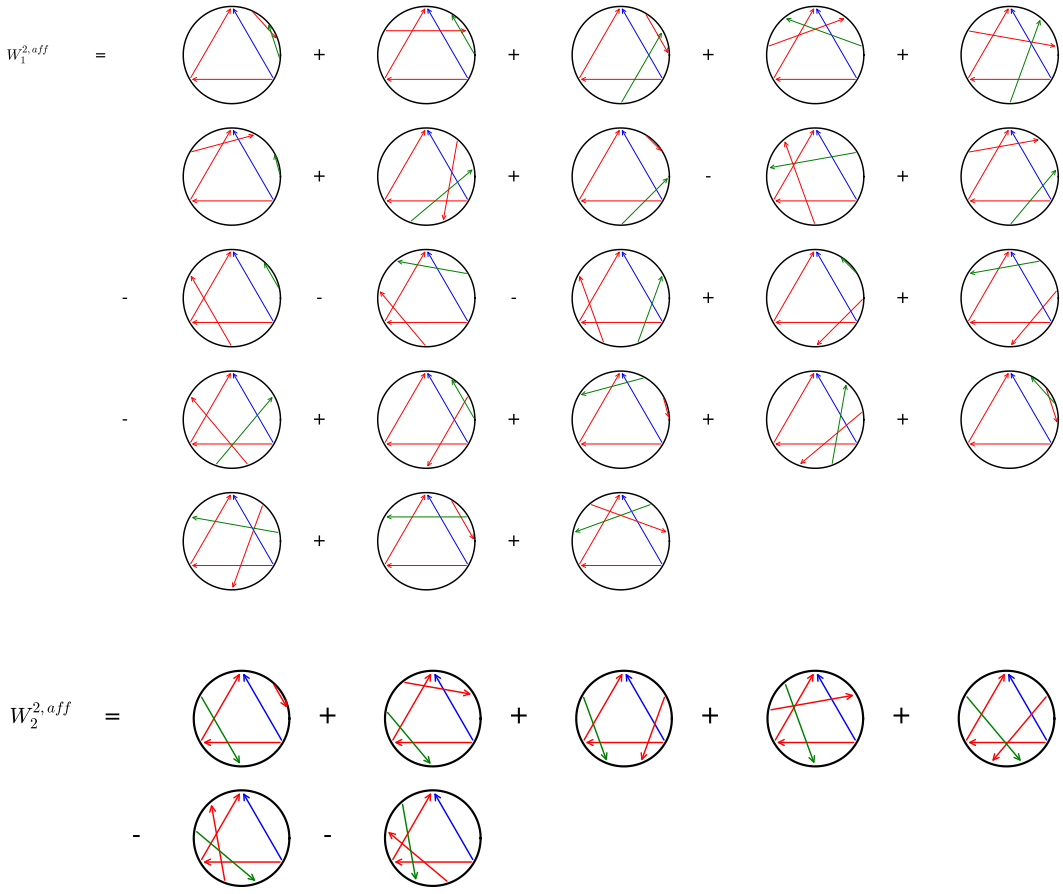


FIGURE A.5 – Liste des poids de degré 2 pour $n = 4$ (suite)

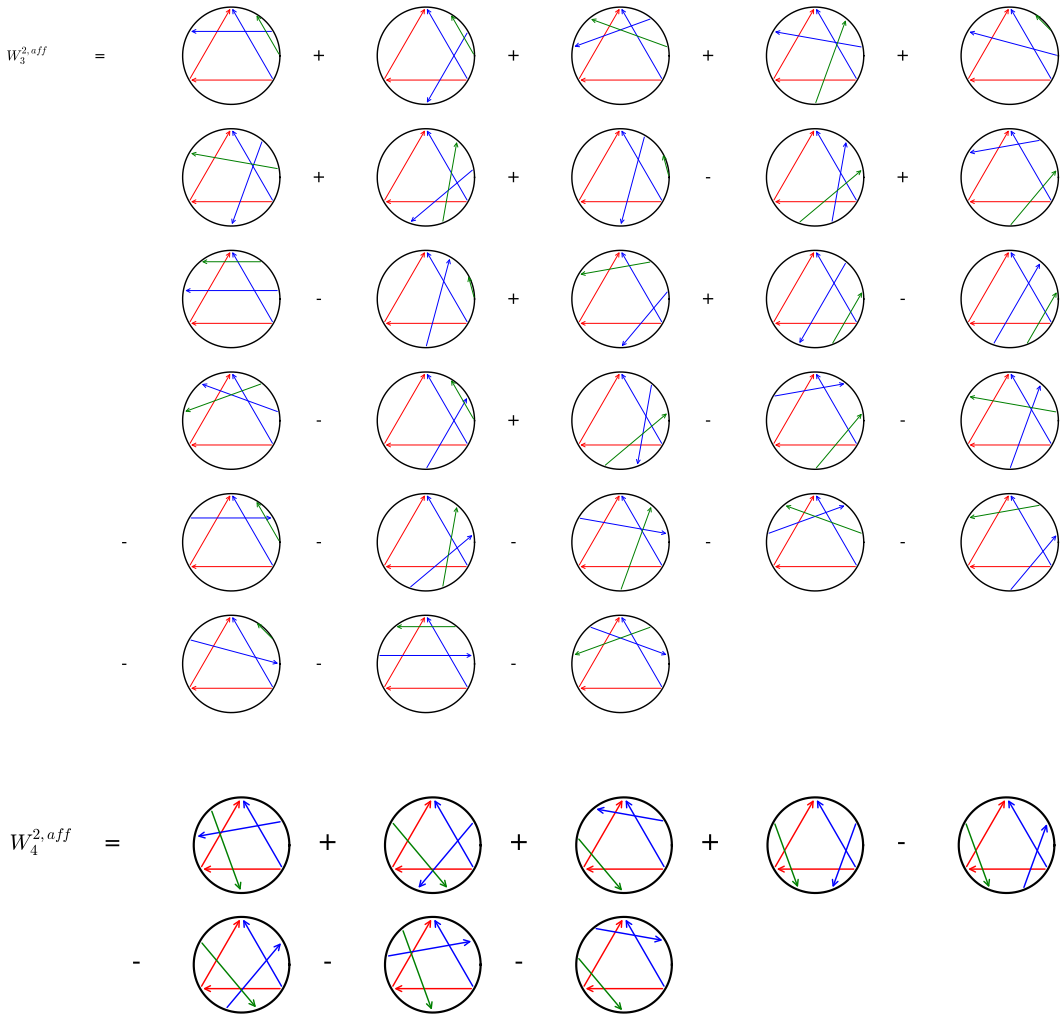


FIGURE A.6 – Liste des poids de degré 2 pour $n = 4$ (suite)

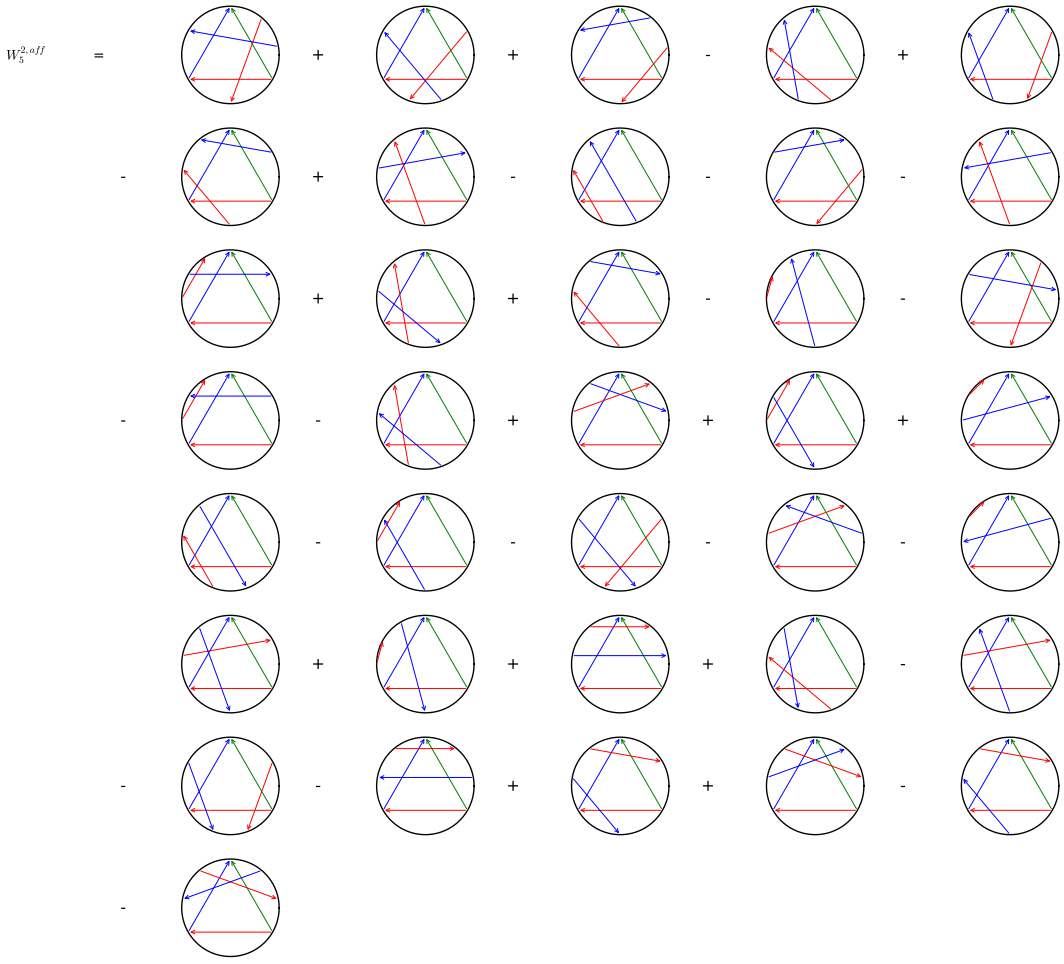


FIGURE A.7 – Liste des poids de degré 2 pour $n = 4$ (suite)

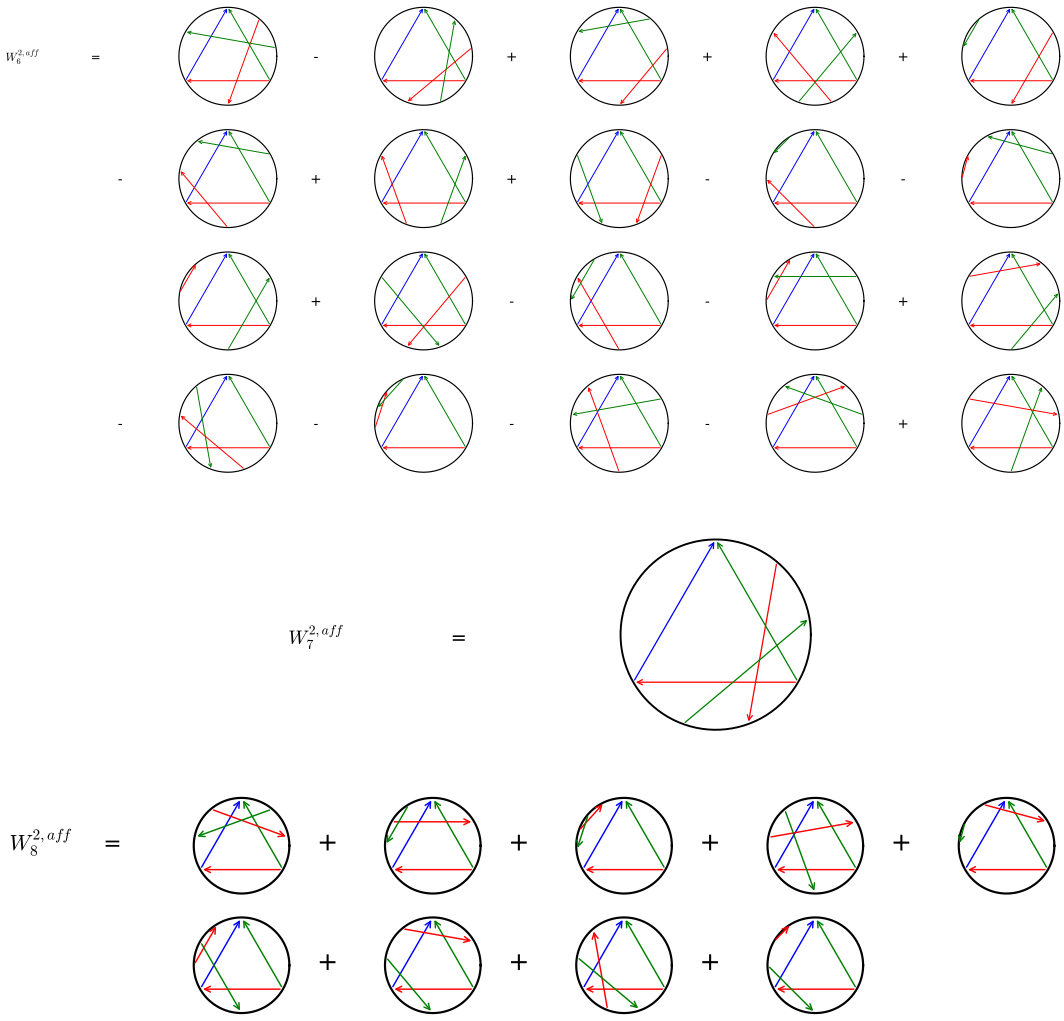


FIGURE A.8 – Liste des poids de degré 2 pour $n = 4$ (suite)

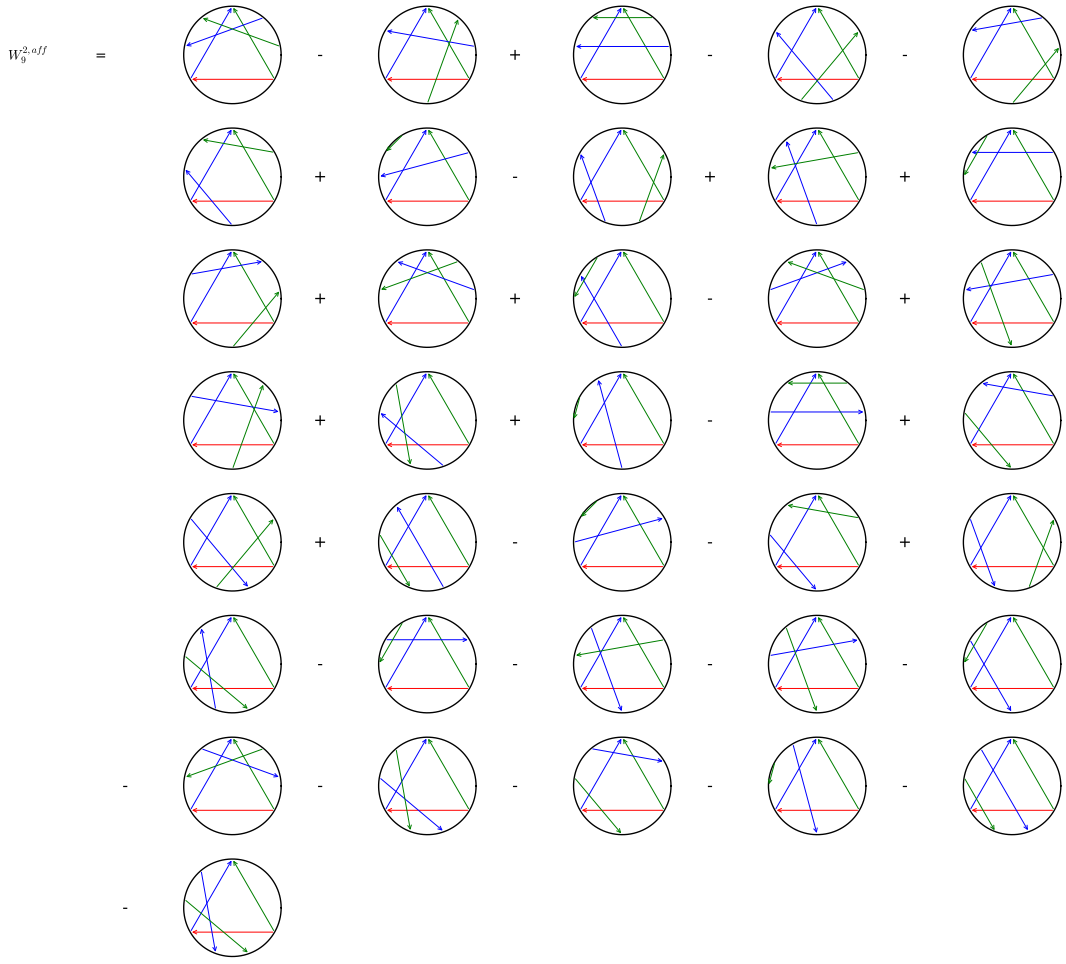


FIGURE A.9 – Liste des poids de degré 2 pour $n = 4$ (suite)

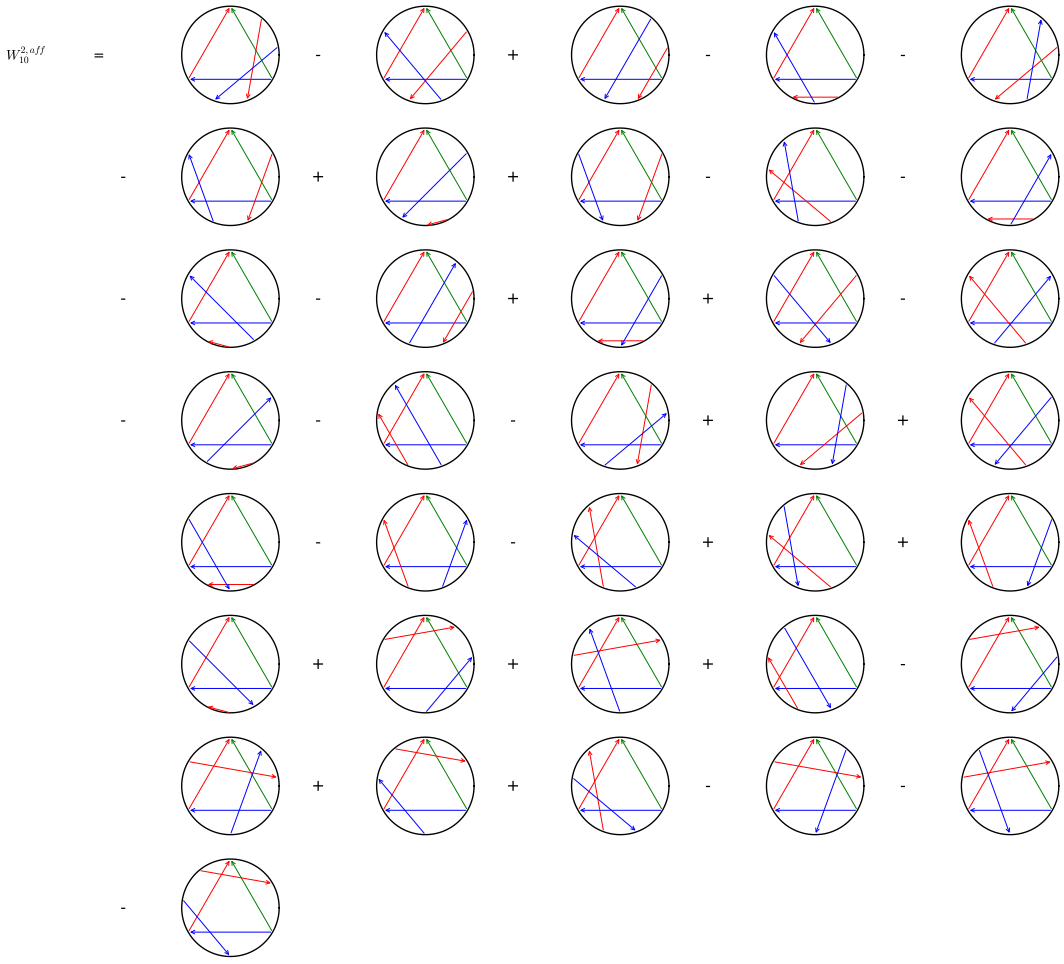


FIGURE A.10 – Liste des poids de degré 2 pour $n = 4$ (suite)

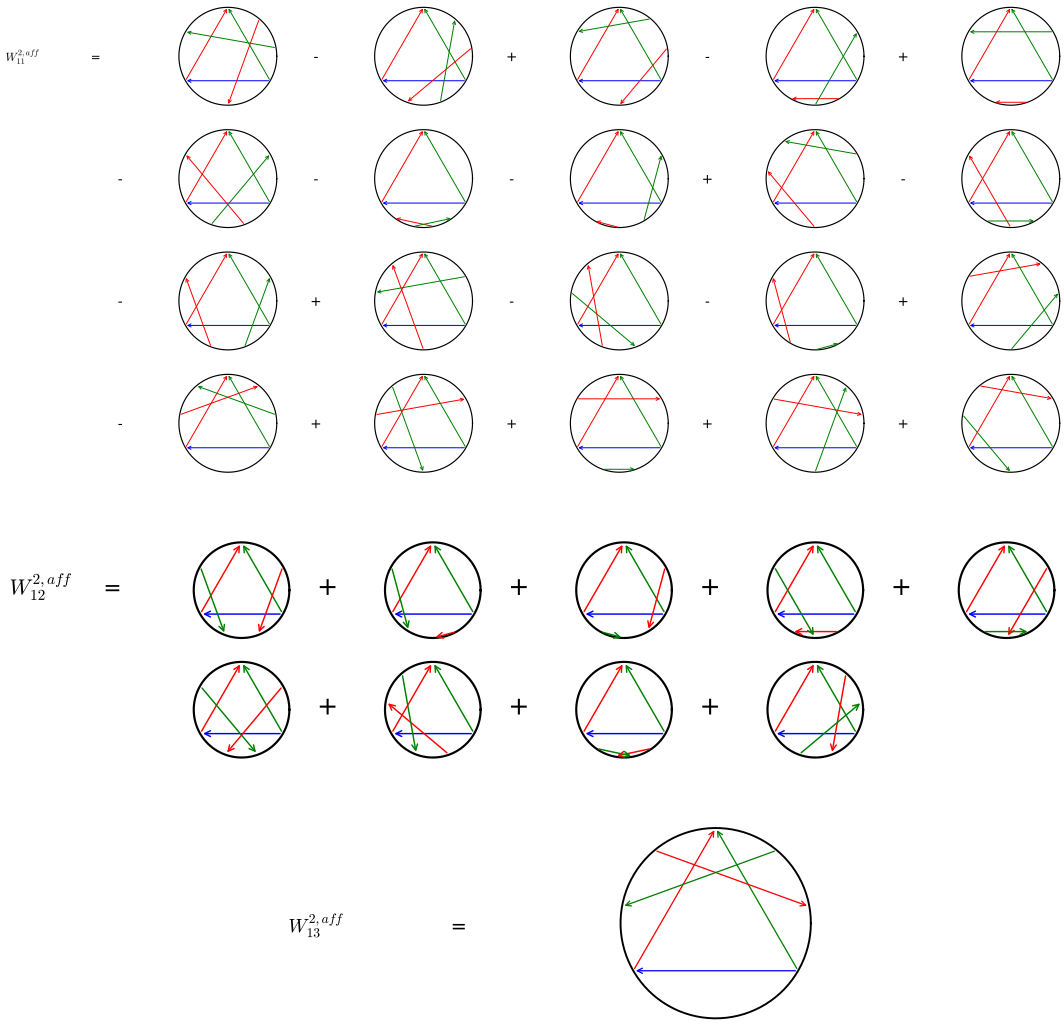


FIGURE A.11 – Liste des poids de degré 2 pour $n = 4$ (suite)

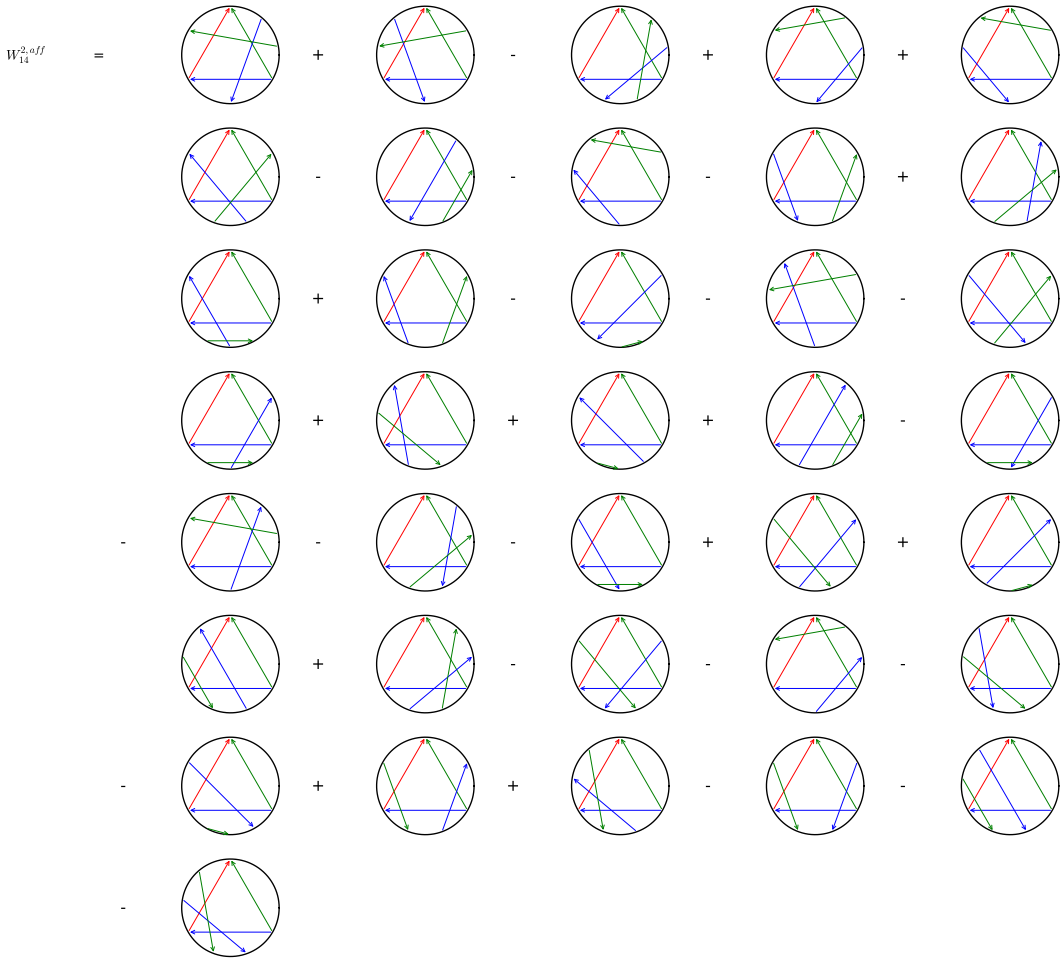


FIGURE A.12 – Liste des poids de degré 2 pour $n = 4$ (suite)

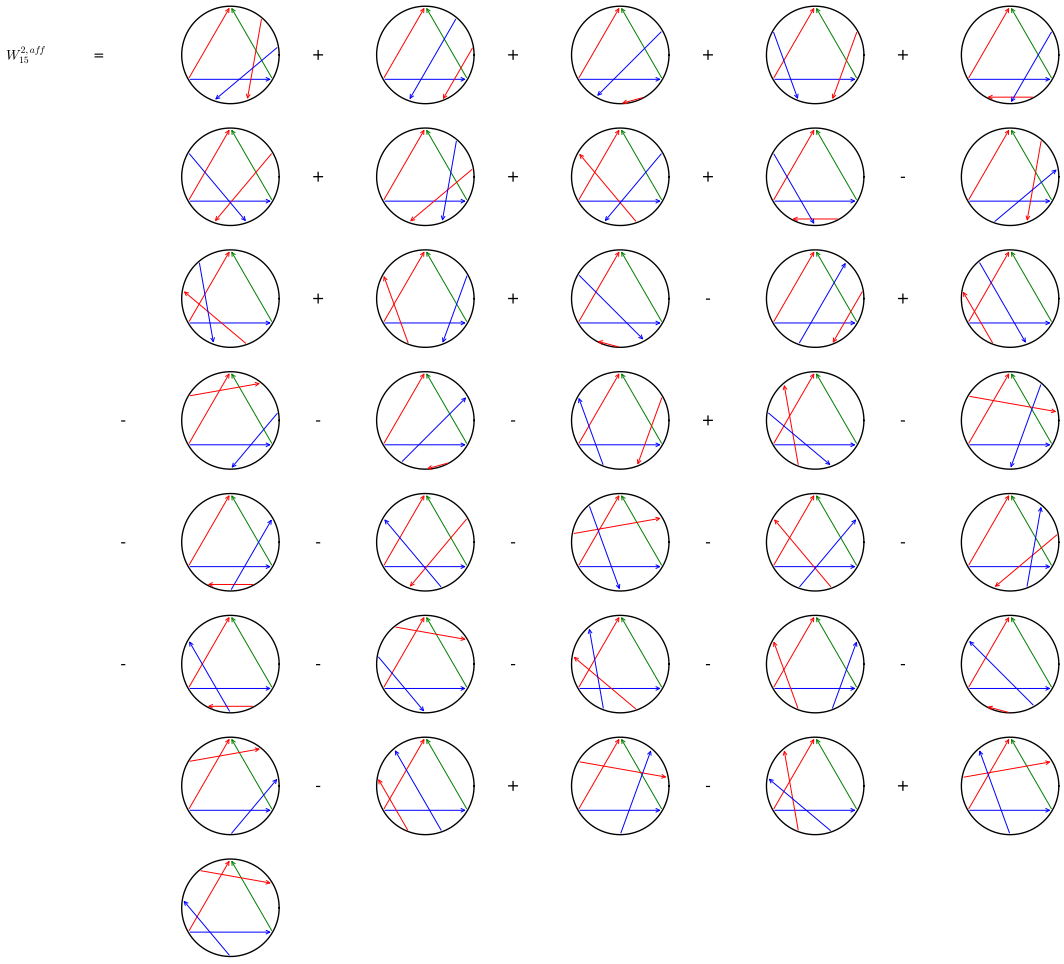


FIGURE A.13 – Liste des poids de degré 2 pour $n = 4$ (suite)

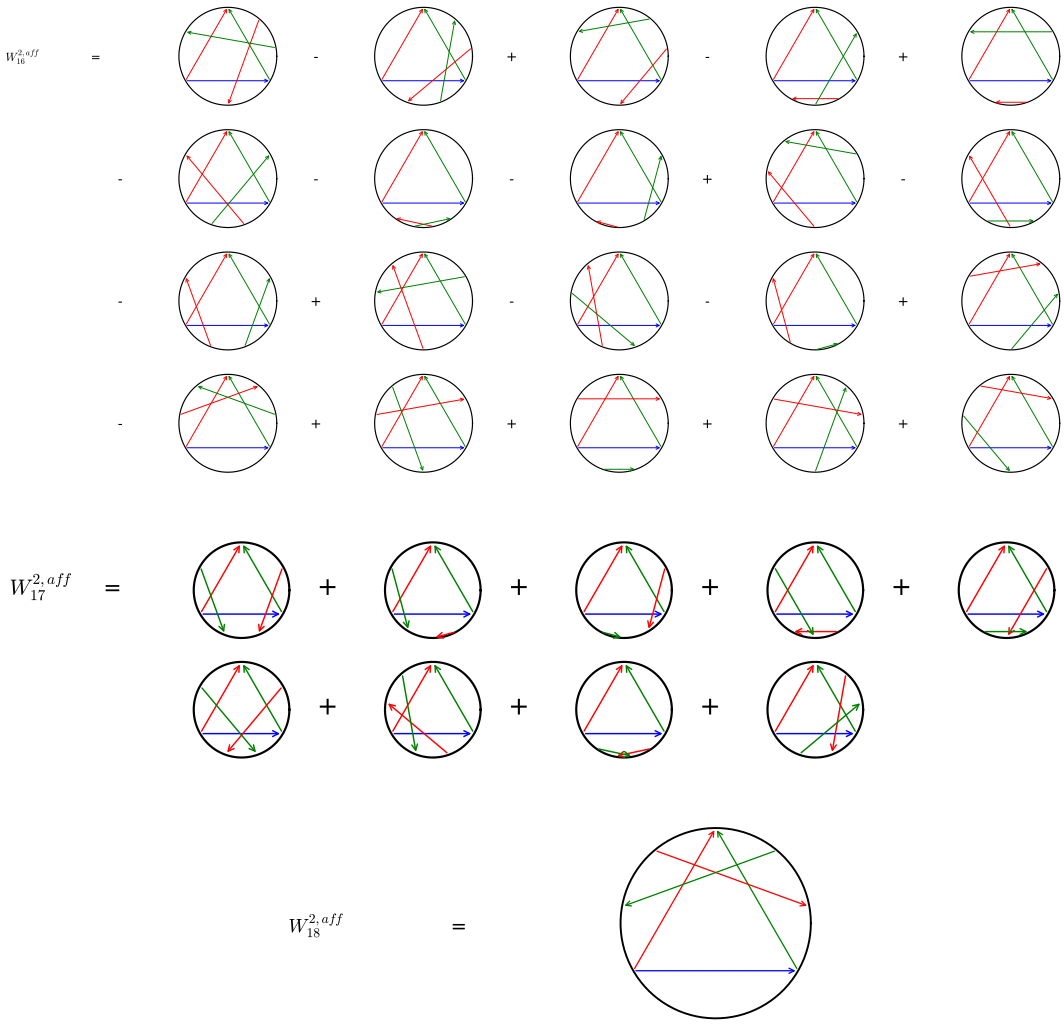


FIGURE A.14 – Liste des poids de degré 2 pour $n = 4$ (suite)

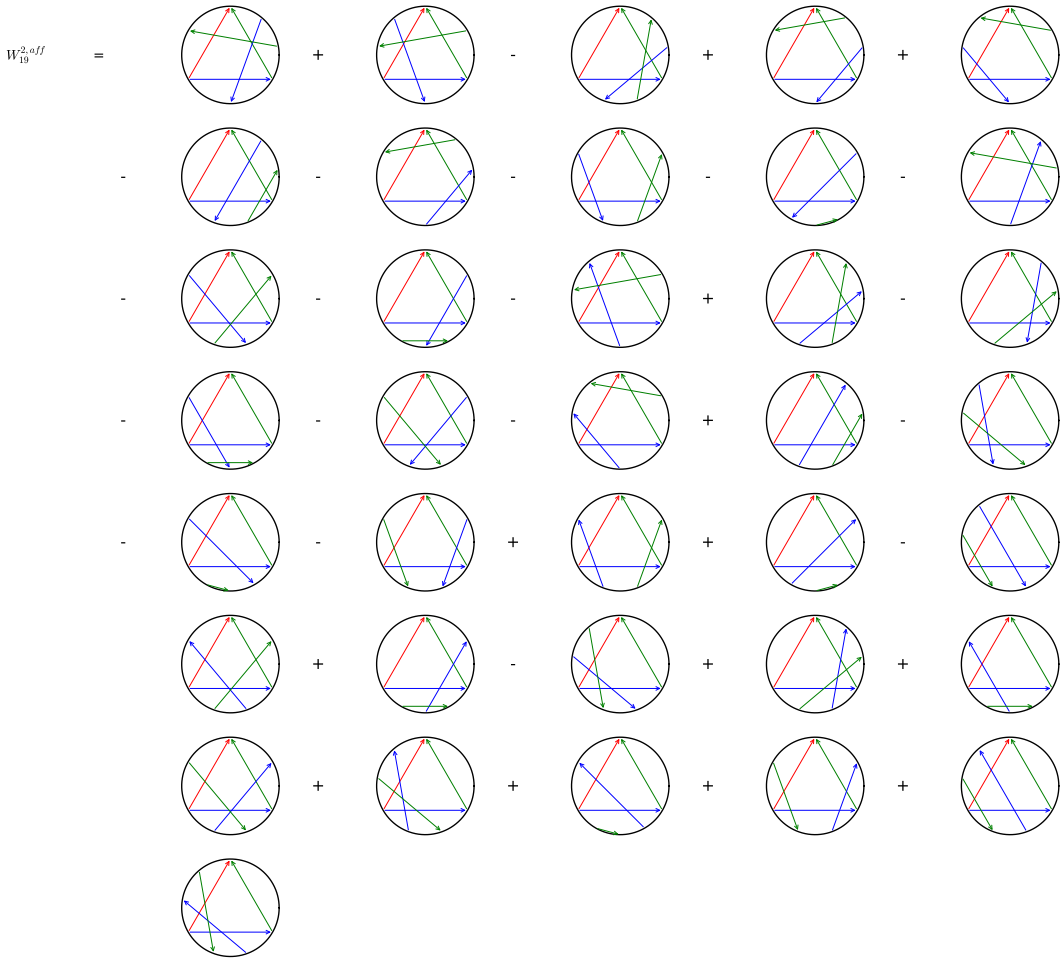


FIGURE A.15 – Liste des poids de degré 2 pour $n = 4$ (suite)

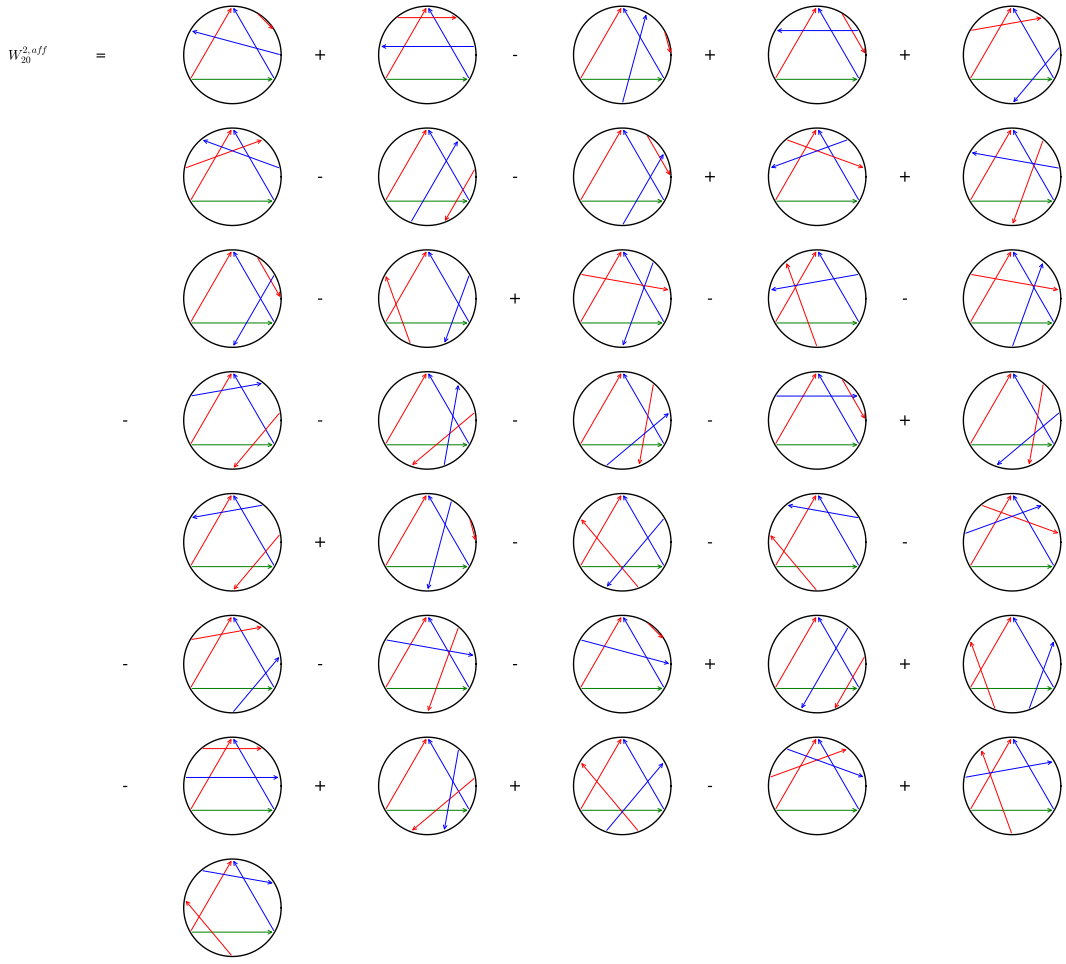


FIGURE A.16 – Liste des poids de degré 2 pour $n = 4$ (suite)

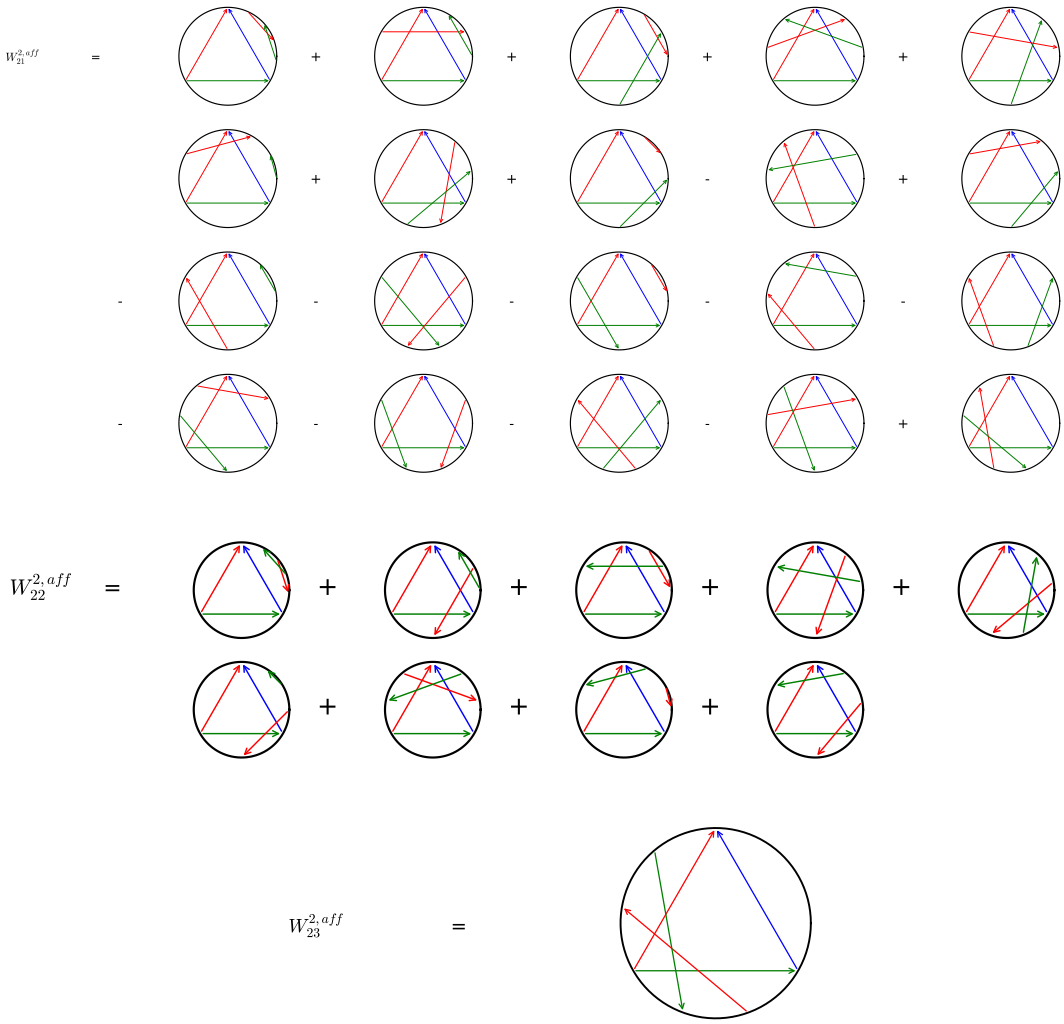


FIGURE A.17 – Liste des poids de degré 2 pour $n = 4$ (suite)

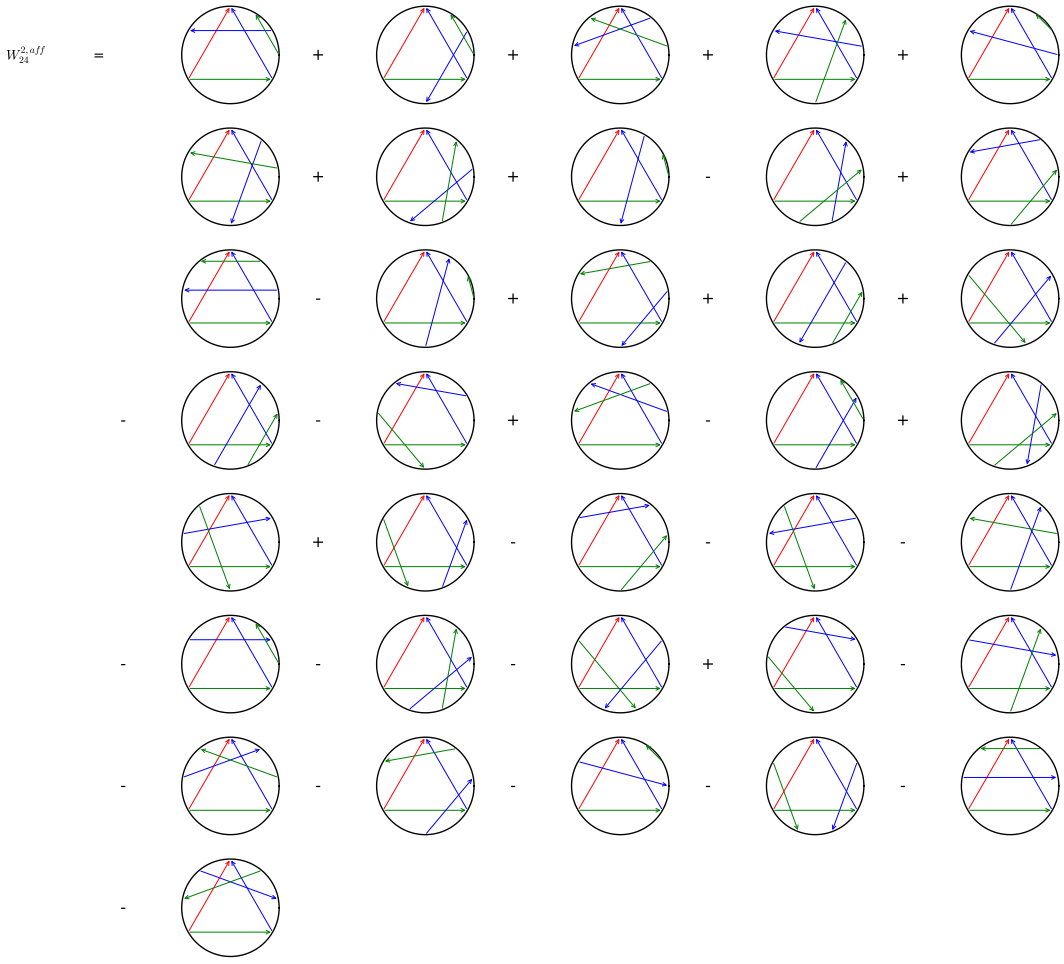


FIGURE A.18 – Liste des poids de degré 2 pour $n = 4$ (suite)

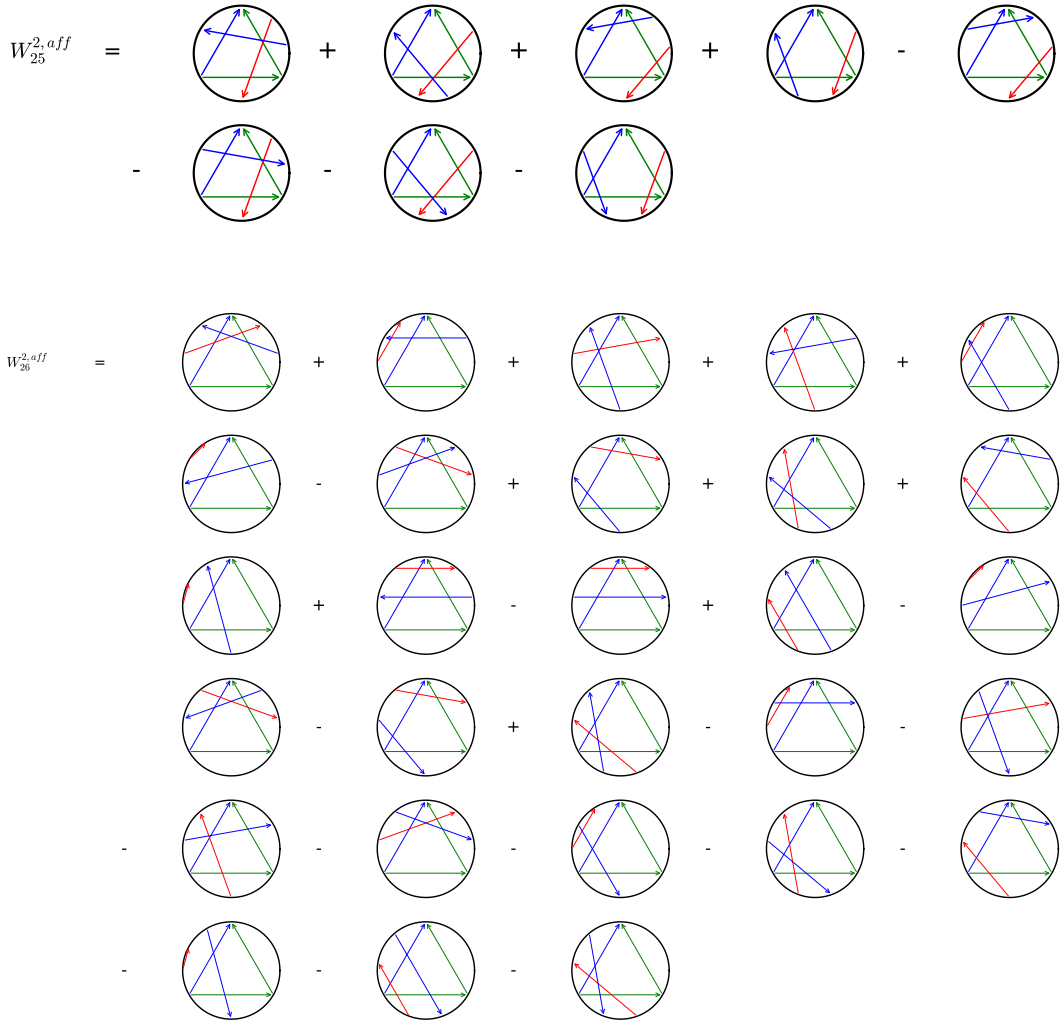


FIGURE A.19 – Liste des poids de degré 2 pour $n = 4$ (suite)

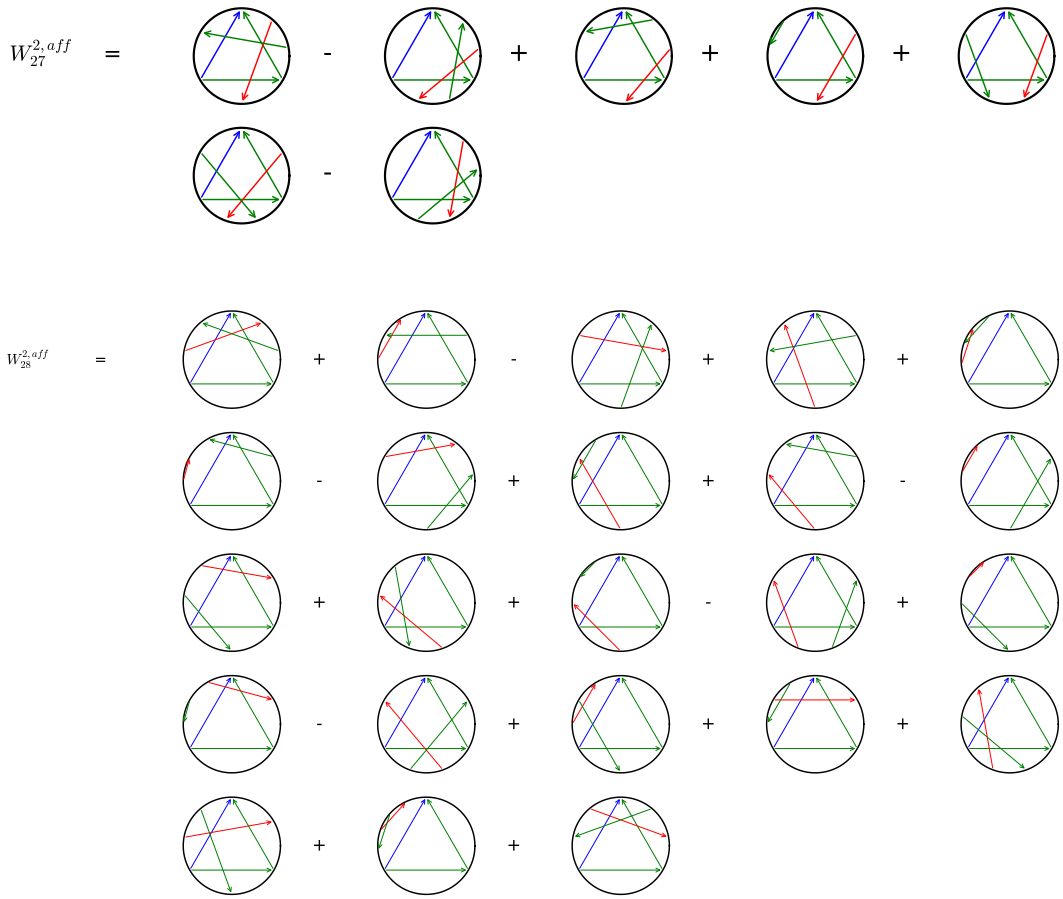


FIGURE A.20 – Liste des poids de degré 2 pour $n = 4$ (suite)

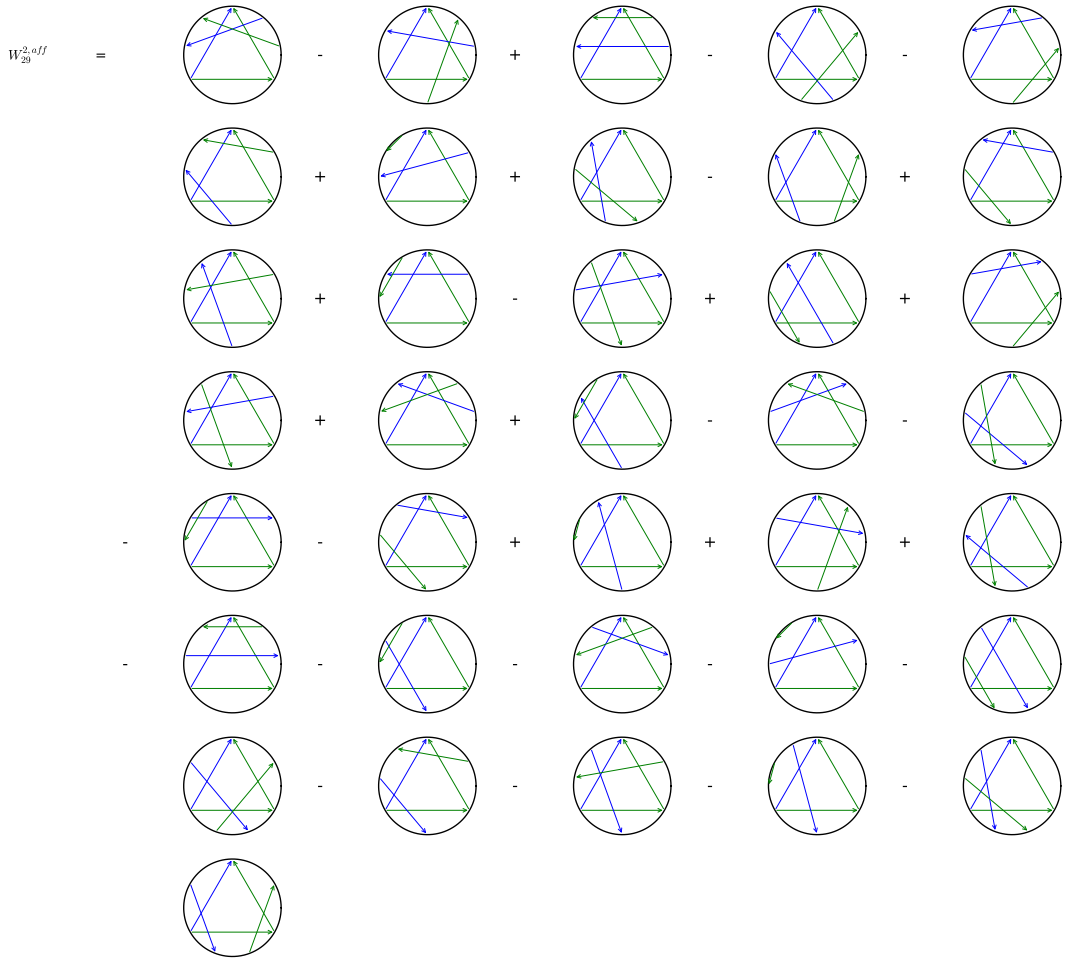


FIGURE A.21 – Liste des poids de degré 2 pour $n = 4$ (fin)

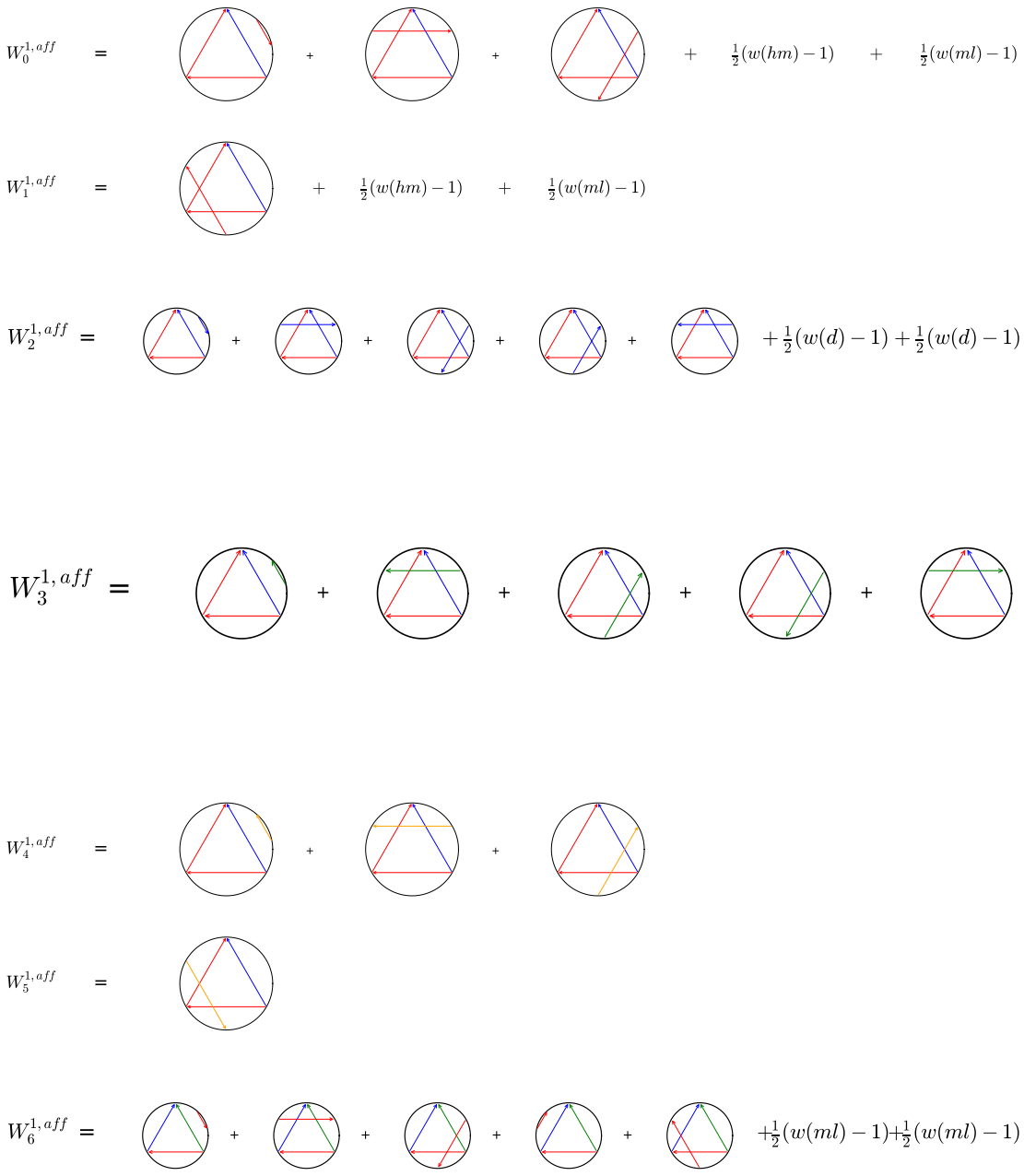


FIGURE A.21 – Liste des poids de degré 1 pour $n = 5$

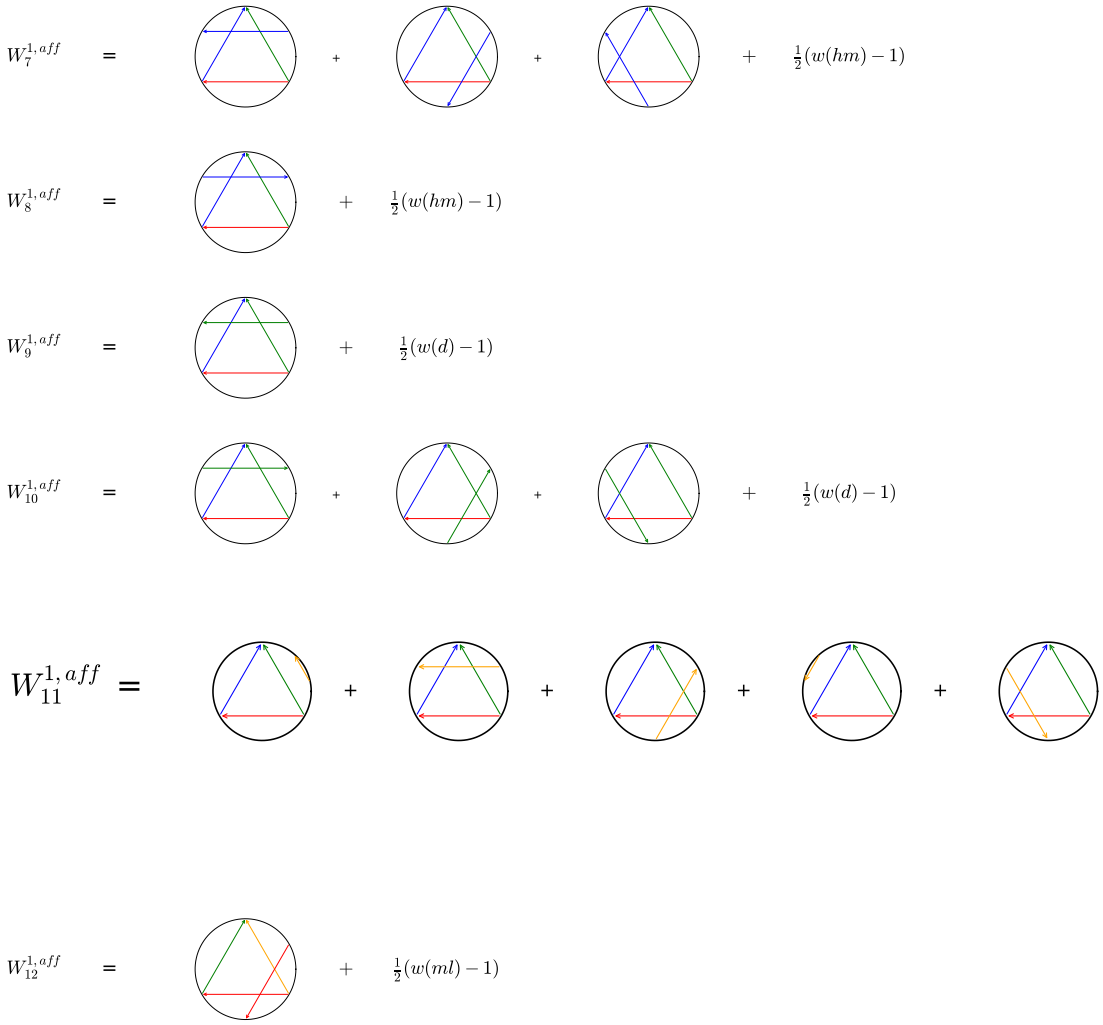


FIGURE A.22 – Liste des poids de degré 1 pour $n = 5$ (suite)

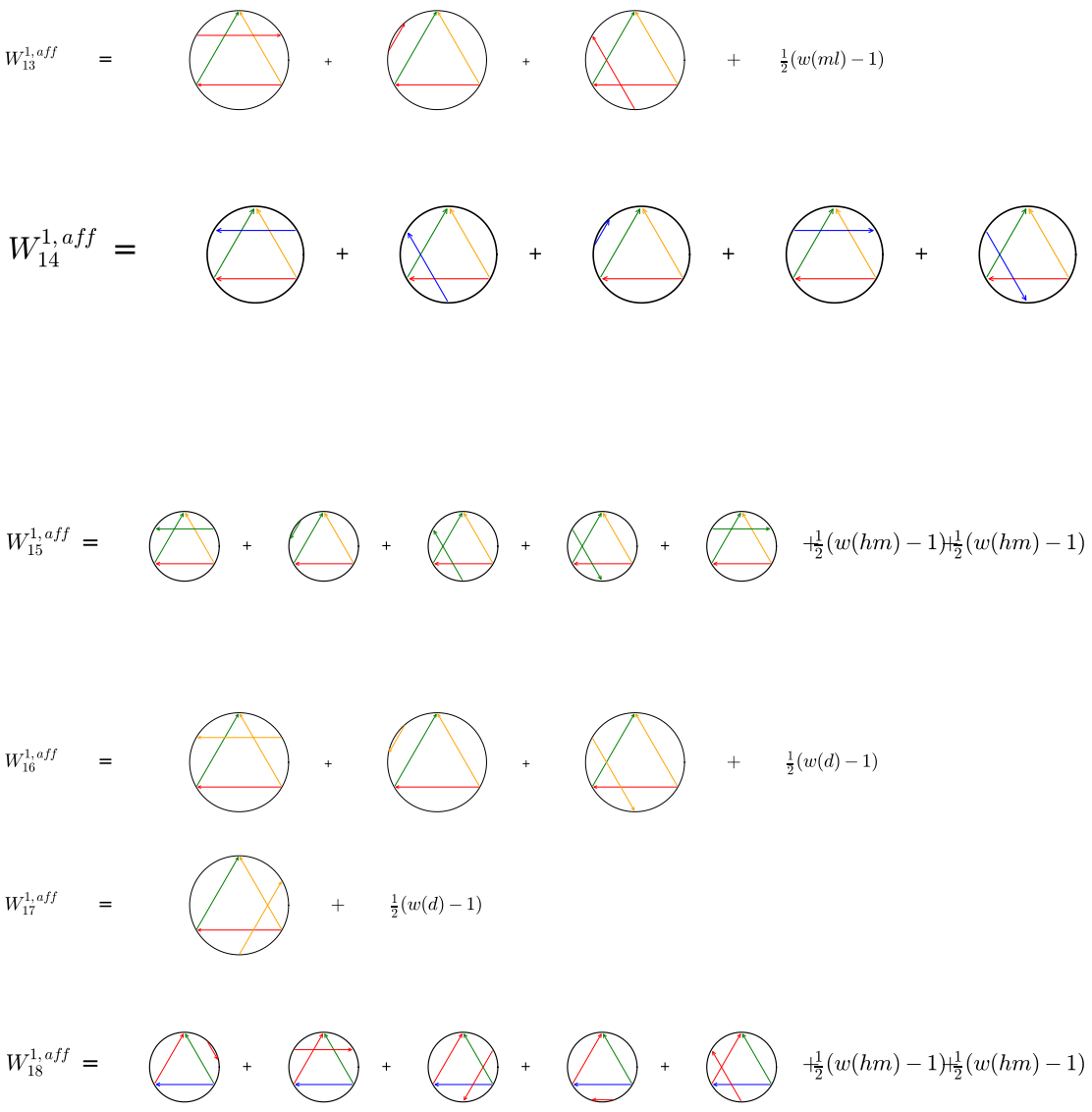


FIGURE A.23 – Liste des poids de degré 1 pour $n = 5$ (suite)

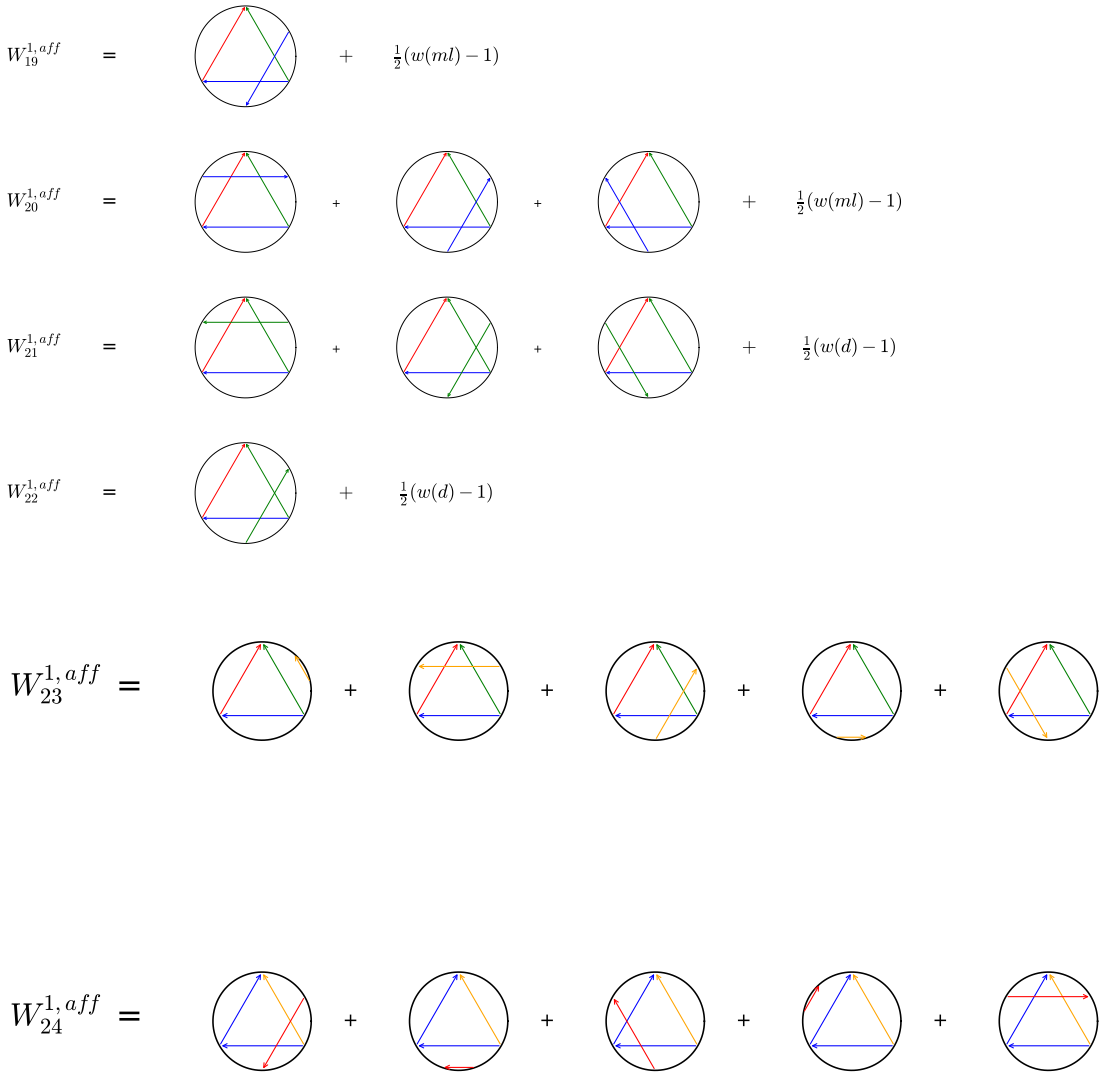


FIGURE A.24 – Liste des poids de degré 1 pour $n = 5$ (suite)

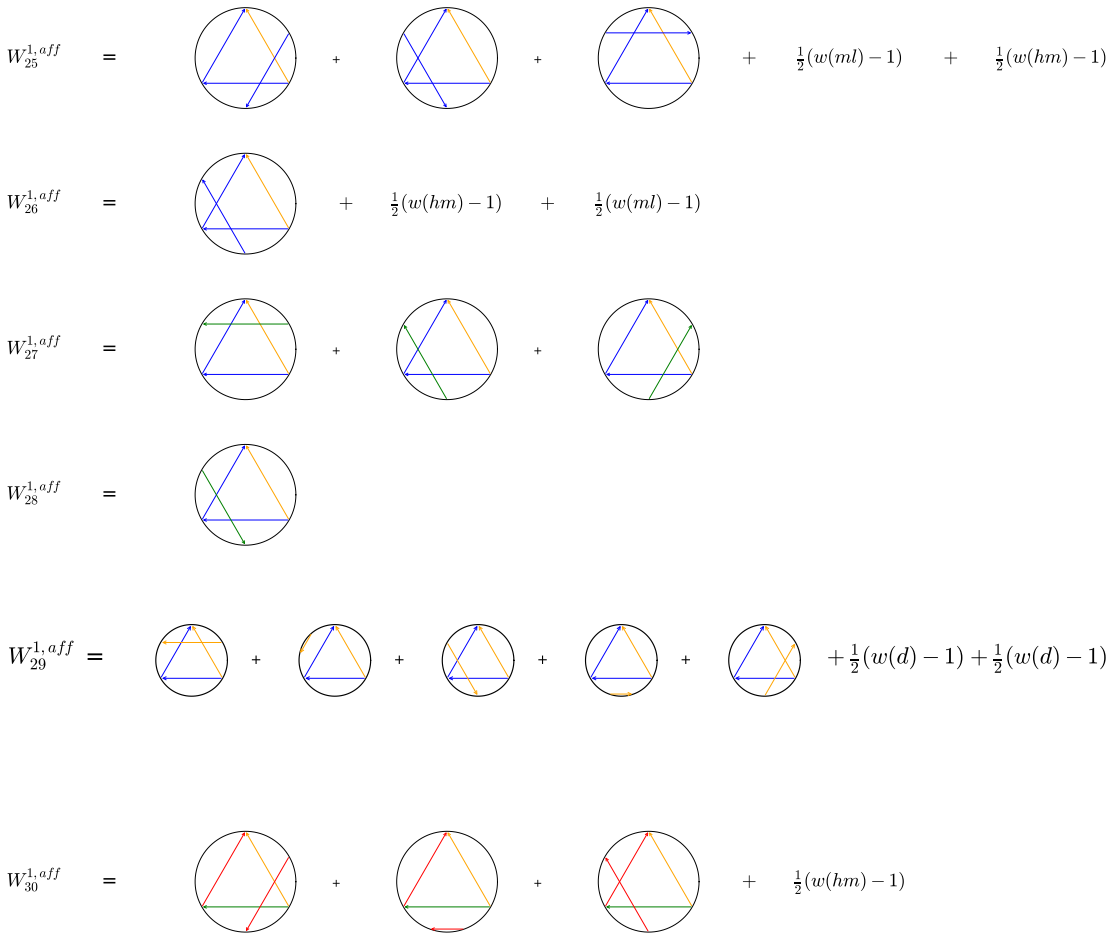


FIGURE A.25 – Liste des poids de degré 1 pour $n = 5$ (suite)

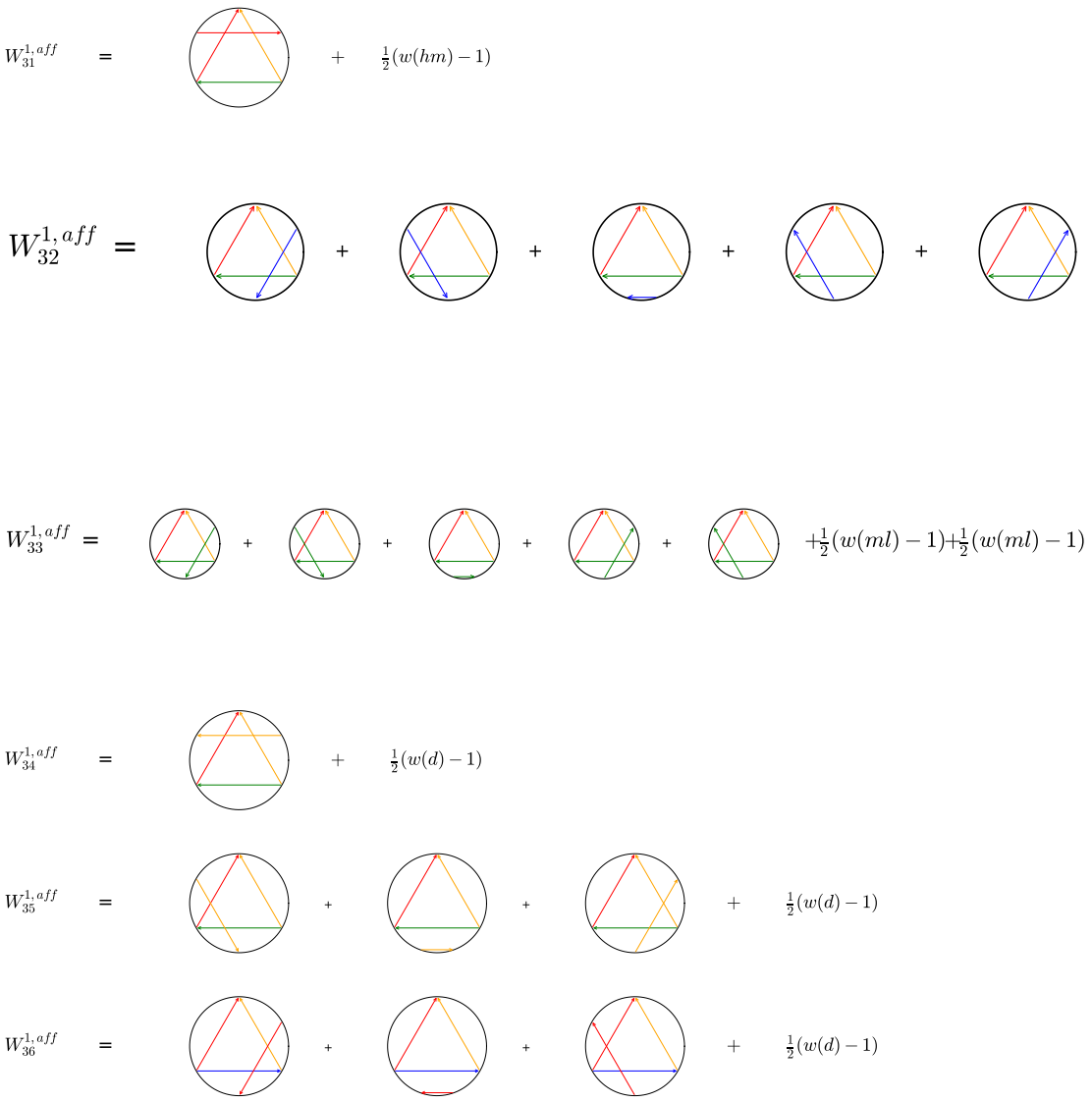


FIGURE A.26 – Liste des poids de degré 1 pour $n = 5$ (suite)

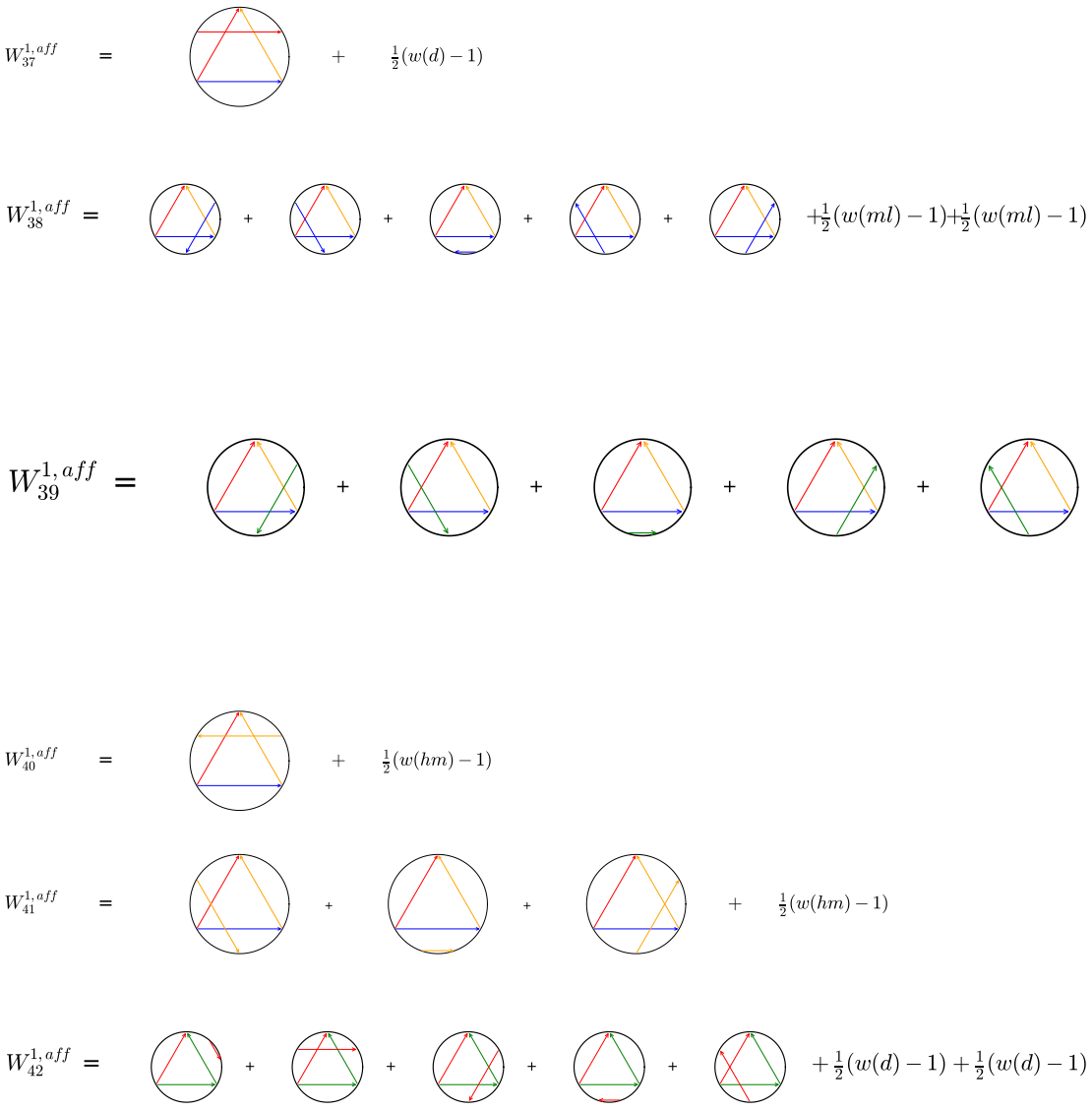


FIGURE A.27 – Liste des poids de degré 1 pour $n = 5$ (suite)

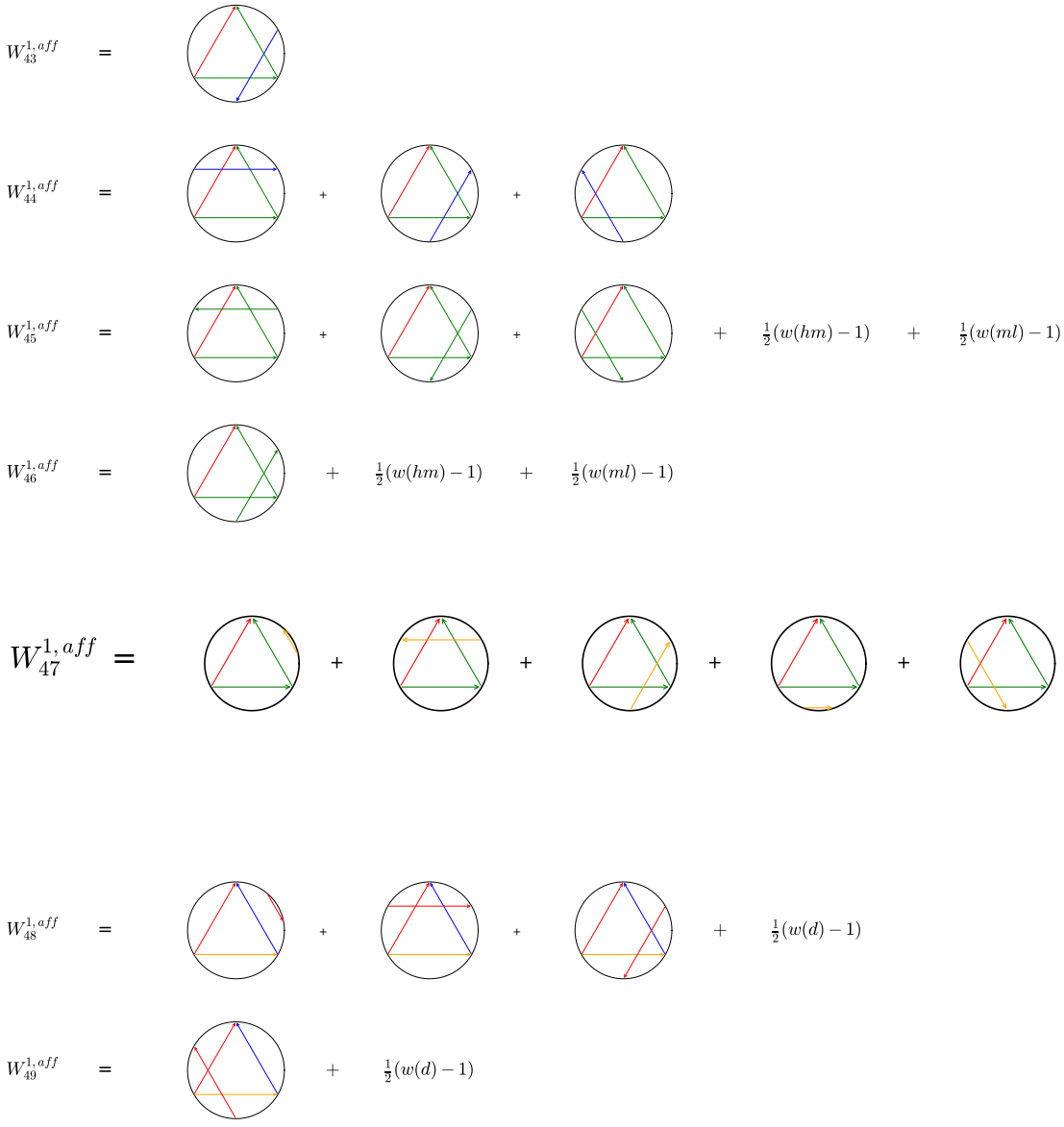
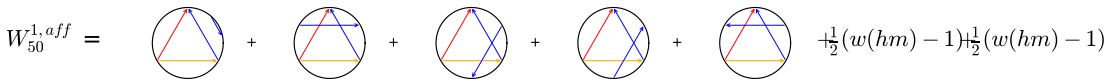
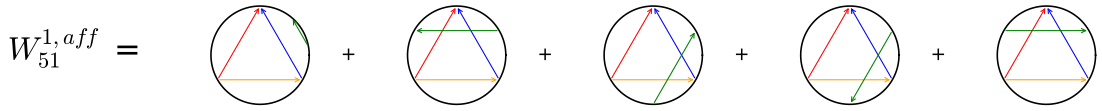
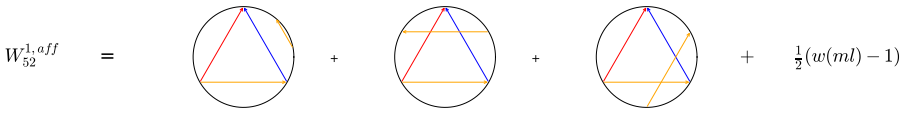
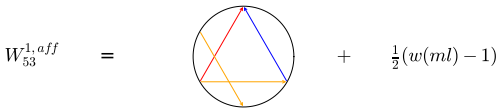


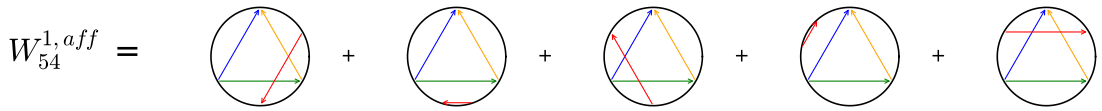
FIGURE A.28 – Liste des poids de degré 1 pour $n = 5$ (suite)

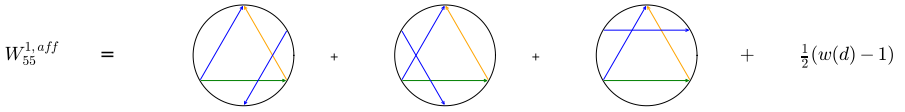












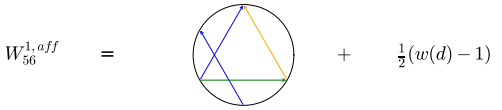


FIGURE A.29 – Liste des poids de degré 1 pour $n = 5$ (suite)

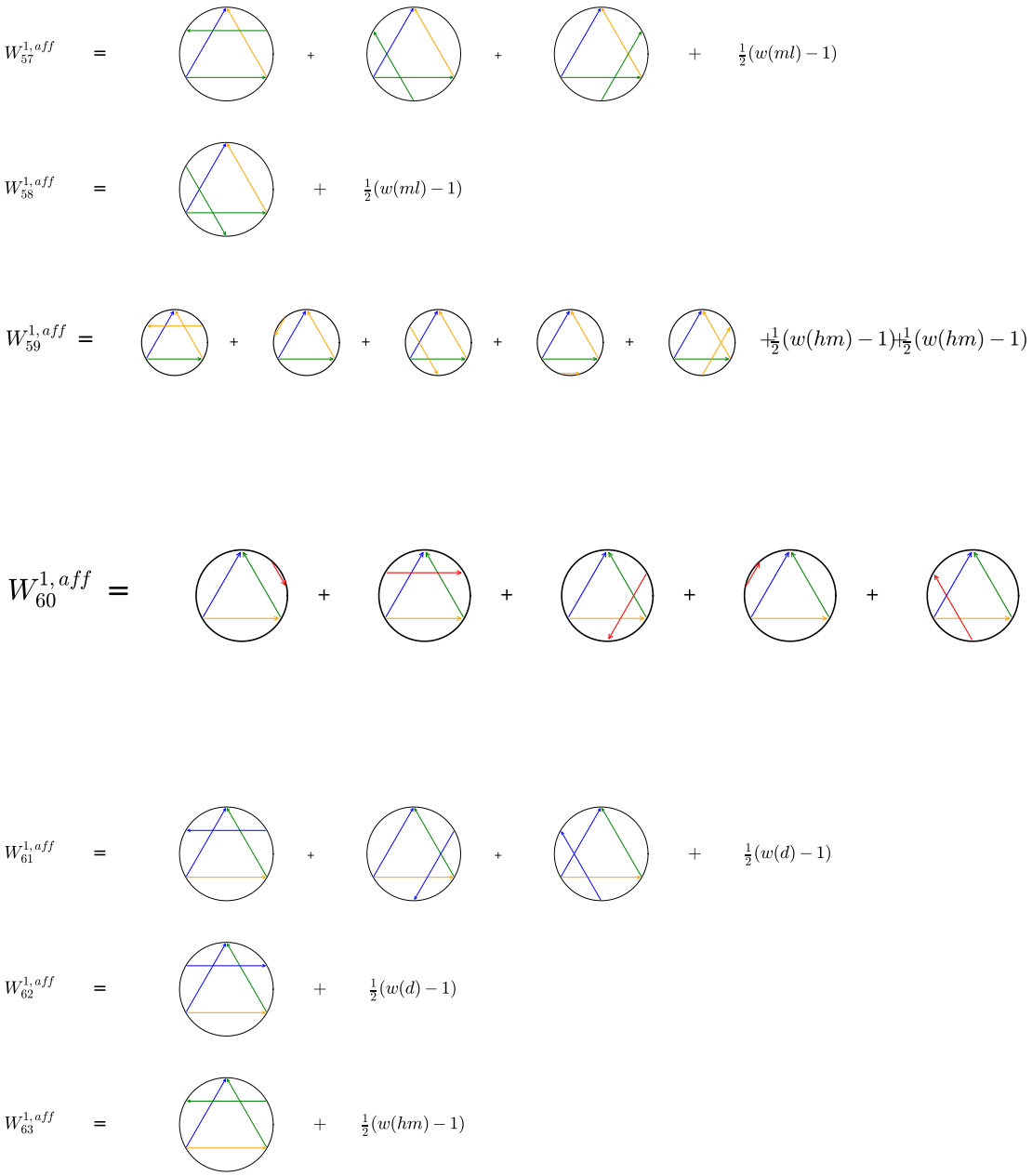


FIGURE A.30 – Liste des poids de degré 1 pour $n = 5$ (suite)

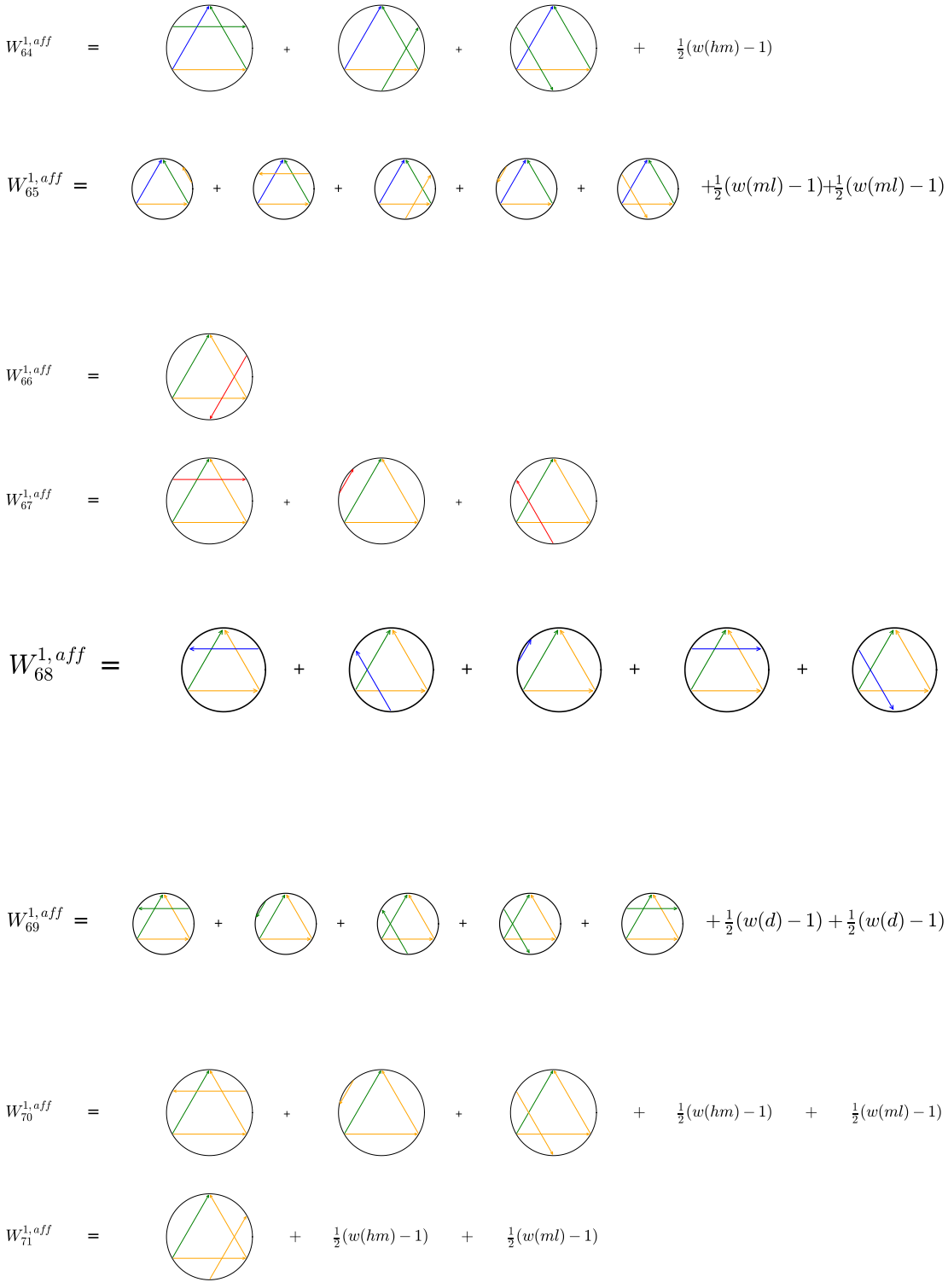


FIGURE A.31 – Liste des poids de degré 1 pour $n = 5$ (suite)

A.3 Programme

Le programme suivant est disponible en libre accès à l'adresse :
`https://github.com/SushiLeila/1cocycles_for_nbraids`

C'est un fichier de type jupyter notebook, utilisable avec Sagemath 8.1 et les versions antérieures. Le programme est utilisable tel quel pour les 4-tresses, peu importe le degré des 1-cocycles que l'on souhaite calculer. Il suffit pour cela d'exécuter les cellules dans l'ordre et de décommenter (en utilisant CTRL + /) les cellules supplémentaires (pour rechercher des tresses par exemple). Un utilisateur qui souhaiterait améliorer et adapter le programme trouvera un code commenté, organisé, et déjà pensé pour être adapté à des n-tresses. Actuellement, seule la partie de génération des poids est utilisable avec n quelconque. Pour rendre le programme totalement opérationnel pour des n-tresses avec n quelconque, il faudra reprendre et adapter les différentes parties suivantes :

— Définition de la couleur associée à une classe d'homotopie > 3,

— Calcul de la classe d'homotopie associée à un croisement,

— Calcul de l'élément de Garside pour $n > 4$,

— Calcul du lacet rot.

Egalement, une interface "user friendly" est en cours de développement. Ce programme est le coeur de ce travail de thèse. Il est le fruit de deux longues années d'autoformation et de programmation. Ainsi, afin de venir compléter le programme, l'interface sera disponible sur le même lien github lorsqu'elle sera terminée. Pour les lecteurs qui souhaiteraient poser des questions, notamment sur le programme, vous pouvez me joindre à l'adresse jhok(at)math.univ-toulouse.fr. Je vous remercie de votre lecture.

program_final_version

1 1-Cocycles for n-braids

1.1 I - Main functions

```
In [ ]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.rcParams.update({'figure.max_open_warning': 0})
        import warnings
        warnings.filterwarnings("ignore")
        from numpy import *
        from copy import *
```

Define the number of brands for the braid

```
In [ ]: Nbraid = 4
        B = BraidGroup(Nbraid)
```

Generating all possible types

```
In [ ]: def generate_all_possible_global_types(Nbraid):
            '''A function that generates all possible global types
            for a triple point of a Nbraid-braid'''
            ListType=[]
            #Negative type
            for ml in range(1,Nbraid-1):
                for hm in range(1,Nbraid-ml):
                    ListType.append([ml,hm,'-'])
            #Positive type
            for d in range(1,Nbraid-1):
                for ml in range(d+1,Nbraid):
                    hm = d + Nbraid - ml
                    ListType.append([ml,hm,'+'])
            return ListType
```

```
In [ ]: ListType = generate_all_possible_global_types(Nbraid)
```

1.1.1 I.a. Translating n-braids into BraidData

Functions to represent closed braids in the solid torus

```
In [ ]: #Function to represent closed braids
        def plot_one_strand(x,y,color='k',intercross = 0.5):
            '''A function that plots a straight strand at
            coordinate (x,y) of length intercross '''
            abscisse = linspace(x,x+intercross,50)
            ordonne = zeros(50)
            plt.plot(abscisse,ordonne + y, color)

        def plot_intercrossing(x,y,N,color='k',intercross=0.5):
            '''A function that plots the intercrossing of an N-braid
            with strand 1 at coordinate (x,y) of length intercross'''
            for i in range(0,N):
                plot_one_strand(x,y+i,color,intercross)

        def plot_local_crossing(x,i,sign,size=0.05,color='k',length=1):
            '''A function that plot locally a crossing at coordinate (x,i)'''
            if sign < 0 :
                #Overcrossing
                a = linspace(x,x+length,50)
                b = a/length + (i-(x/length))
                plt.plot(a,b,color)
                #Undercrossing
                xmil = x + (length/2)
                a = linspace(x,xmil-size,50)
                b = -a/length + i + 1 + (x/length)
                plt.plot(a,b,color)
                xmil = x + (length/2)
                a = linspace(xmil+size,x+length,50)
                b = -a/length + i + 1 + (x/length)
                plt.plot(a,b,color)
            elif sign > 0 :
                #Overcrossing
                a = linspace(x,x+length,50)
                b = -a/length + i + 1 + (x/length)
                plt.plot(a,b,color)
                #Undercrossing
                xmil = x + (length/2)
                a = linspace(x,xmil-size,50)
                b = a/length + (i-(x/length))
                plt.plot(a,b,color)
                xmil = x + (length/2)
```

```
        a = linspace(xmil+size,x+length,50)
        b = a/length + (i-(x/length))
        plt.plot(a,b,color)

def plot_crossing(x,sigma,N,size=0.05,color='k',length=1):
    '''A function that plot the crossing sigma at
    coordinate x of an N-braid'''
    plot_local_crossing(x,abs(sigma),sigma,size,color,length)
    for i in range(1,N+1):
        if i != abs(sigma) and i!= abs(sigma)+1 :
            plot_one_strand(x,i,color,length)

def plot_half_circle(center,r=0.5,sign=-1,color='k'):
    '''A function that plot half a circle'''
    if sign > 0:
        theta = linspace(-pi/2,pi/2,50)
    elif sign <0:
        theta = linspace(pi/2,3*pi/2,50)
    abscisse = r*cos(theta) + center[0]
    ordonne = r*sin(theta) + center[1]
    plt.plot(abscisse,ordonne,color)


def plot_border(x,y,N,sign,color='k'):
    '''A function that plots the border of a N-braid
    at coordinates (x,y)'''
    center = (x,y)
    for i in range(1,N+1):
        plot_half_circle(center,i,sign,color)

def plot_braid(S,color='k',intercross=0.5,size=0.05,length=1,
               res=300):
    '''A function that plot a braid S'''
    fig = plt.figure(dpi=res)
    N = max(vectorize(abs)(S)) + 1
    #Plotting the crossings
    plot_intercrossing(0,1,N,color,intercross)
    x = intercross
    for sigma in S:
        plot_crossing(x,sigma,N,size,color,length)
        x = x + length
        plot_intercrossing(x,1,N,color,intercross)
        x = x + intercross
    #Plotting the "lower" strands below the center hole
```

```
            plot_intercrossing(0,-N,N,color,len(S)*(intercross+length)
                               +intercross)
            #Plotting the turns
            plot_border(0,0,N,-1,color)
            x = len(S)*(intercross+length)+intercross
            y = 0
            plot_border(x,y,N,1,color)
            #Plotting the singular point
            x = (len(S)*(intercross+length)+intercross)/2
            motif = color + 'o'
            plt.plot(x,0,motif)
            plt.ylim([-N-0.5,N+0.5 ])
            plt.axis('off')
            plt.axis('scaled')
```

Functions to generate BraidData and rot

```
In [ ]: def end_homotopy_class(braid):
            L = Link(B(braid))
            #Getting the last permutation of braid
            n = len(braid)
            p = braid[n-1]
            h = 0
            #Computing the homotopy class of the last crossing
            if abs(p) == 1 :
                if L.braid().permutation().cycle_string() == '(1,2,3,4)':
                    h = 1
                if L.braid().permutation().cycle_string() == '(1,2,4,3)':
                    h = 1
                if L.braid().permutation().cycle_string() == '(1,3,2,4)':
                    h = 2
                if L.braid().permutation().cycle_string() == '(1,3,4,2)':
                    h = 3
                if L.braid().permutation().cycle_string() == '(1,4,2,3)':
                    h = 2
                if L.braid().permutation().cycle_string() == '(1,4,3,2)':
                    h = 3

            if abs(p) == 2 :
                if L.braid().permutation().cycle_string() == '(1,2,3,4)':
                    h = 1
                if L.braid().permutation().cycle_string() == '(1,2,4,3)':
                    h = 2
                if L.braid().permutation().cycle_string() == '(1,3,2,4)':
```

```
            h = 3
        if L.braid().permutation().cycle_string() == '(1,3,4,2)':
            h = 2
        if L.braid().permutation().cycle_string() == '(1,4,2,3)':
            h = 1
        if L.braid().permutation().cycle_string() == '(1,4,3,2)':
            h = 3

    if abs(p) == 3 :
        if L.braid().permutation().cycle_string() == '(1,2,3,4)':
            h = 1
        if L.braid().permutation().cycle_string() == '(1,2,4,3)':
            h = 3
        if L.braid().permutation().cycle_string() == '(1,3,2,4)':
            h = 2
        if L.braid().permutation().cycle_string() == '(1,3,4,2)':
            h = 1
        if L.braid().permutation().cycle_string() == '(1,4,2,3)':
            h = 2
        if L.braid().permutation().cycle_string() == '(1,4,3,2)':
            h = 3

    #If the permutation is negative, just take 4-h
    if sign(p) == -1 :
        h = 4-h

    return h

def any_homotopy_class(braid,i):
    T = deepcopy(braid)
    #Making braid[i] the last permutation
    for j in range(i+1):
        T.append(T[0])
        T.pop(0)
#     print('Tresse',T)
    h = end_homotopy_class(T)
    return h

def gauss_coordinates(GD,i):
    '''A function that computes the coordinates of the arrow number i
    of a torus gauss code'''
    #In : GD = GaussDiagram as a list, i number of the arrow
    #Out : C a list of 2 coordinates
```

```
    n = len(GD)
    C = ['empty','empty']
    #Getting the first coordinate of the arrow i
    for j in range(n):
        if GD[j] == i:
            C[0] = j
    #Getting the second coordinate of the arrow i
        if GD[j] == -i:
            C[1] = j
    return C

def TorusGaussCode_reconstruct(A):
    '''A function that computes a TorusGaussCode from
    a set of TorusGaussArrows A'''
    T = TorusGaussCode()
    #Setting the arrows
    T.set_Arrows(A)
    #Computing the Gauss Code from the arrows coordinates
    l = len(A)
    GD = list(zeros(2*l))
    #For each arrow, we modify the gauss diagram
    for a in A:
        C = a.get_coordinates()
        n = a.get_number()
        #Foot first
        GD[C[0]] = n
        #Then head
        GD[C[1]] = -n
    T.set_GaussDiagram(GD)
    return T

def compute_arrow(braid,i,number):
    '''A function that computes the TorusGaussArrow of
    a braid in position i'''
    A = TorusGaussArrow()
    L = Link(B(braid))
    #Arrow number
    A.set_number(number)
    #Arrow position
    A.set_position(i)
    #Arrow braid element
    A.set_braidelement(braid[i])
    #Arrow homotopy
    A.set_homotopy(any_homotopy_class(braid,i))
```

```
        #Arrow writhe
        A.set_writhe(L.oriented_gauss_code()[1][i])
        GD = L.oriented_gauss_code()[0][0]
        #Arrow coordinates
        A.set_coordinates(gauss_coordinates(GD,number))
        return A

#Function for the RM2
def coordinates_RM2(B,e,i):
    '''A function that returns the coordinates of
    the two arrows created in a RM2 in position i
    with braid element e'''
    #Initialisation
    braid = B.get_BraidClosure()
    A = B.get_TorusGaussCode().get_Arrows()
    m = len(braid) - 1
    E = abs(e)
    #Computing the coordinates
    f1 = 'INIT'
    h1 = 'INIT'
    for j in range(m):
        if abs(braid[i-j-1]) == E:
            X = A[i-j-1]
            Xf = X.get_coordinates()[0]
            Xh = X.get_coordinates()[1]
            if braid[i-j-1] < 0:
                Xf,Xh = Xh,Xf
            if type(f1) == str:
                f1 = (Xh + 1)%(2*m)
                position_element_foot = i-j-1
            if type(h1) == str:
                h1 = (Xf + 1)%(2*m)
                position_element_head = i-j-1
        if abs(braid[i-j-1]) == E-1:
            X = A[i-j-1]
            Xf = X.get_coordinates()[0]
            Xh = X.get_coordinates()[1]
            if braid[i-j-1] < 0:
                Xf,Xh = Xh,Xf
            if type(f1) == str:
                f1 = (Xf + 1)%(2*m)
                position_element_foot = i-j-1
        if abs(braid[i-j-1]) == E+1:
            X = A[i-j-1]
```

```
            Xf = X.get_coordinates()[0]
            Xh = X.get_coordinates()[1]
            if braid[i-j-1] < 0:
                Xf,Xh = Xh,Xf
            if type(h1) == str:
                h1 = (Xh + 1)%(2*m)
                position_element_head = i-j-1

        if type(f1) != str and type(h1) != str:
            break
    #Adjusting for negative first element
    if e < 0 :
        f1,h1 = h1,f1
    #Computing coordinates for the second arrow
    f2 = (f1 + 1)%(2*m)
    h2 = (h1 + 1)%(2*m)

    #Particular Case
    if position_element_foot < 0:
        f1 = (f1 + 2)%(2*m)
        f2 = (f2 + 2)%(2*m)
    elif position_element_head < 0:
        h1 = (h1 + 2)%(2*m)
        h2 = (h2 + 2)%(2*m)
    #Adjusting the two greater coordinates
    if f1 < h1 :
        h1 = h1 + 2
        h2 = h2 + 2
    else:
        f1 = f1 + 2
        f2 = f2 + 2
    return [f1,h1],[f2,h2]

#Functions for the RM3
def find_arrow_coord(coord,A):
    '''A function that finds the arrow in a set A
    that have coordinates coord'''
    arrow = TorusGaussArrow()
    for a in A:
        if a.get_coordinates()[0] == coord
        or a.get_coordinates()[1] == coord :
            arrow = a
    return arrow
```

```
def find_arrow_position(A,i):
    '''A function that returns the arrow that is in position i'''
    ans = TorusGaussArrow()
    for a in A:
        if a.get_position() == i:
            ans = a
    return ans

def find_arrow_number(A,i):
    '''A function that returns the arrow that is in position i'''
    ans = TorusGaussArrow()
    for a in A:
        if a.get_number() == i:
            ans = a
    return ans

def find_switchables(coord,L,numb):
    '''A function that returns all the switchable coordinates
    from a set of coordinates L'''
    Switchables = []
    for c in L:
        if (c-coord)%numb == 1 or (coord-c)%numb == 1:
            Switchables.append(c)
    return Switchables

def is_switchable_once(coord,coord1,L,numb):
    '''A function that determines in a list if a coordinate is
    switchable with only one coordinate (coord1 is
    a forbidden switchable coordinate)'''
    Switchables = find_switchables(coord,L,numb)
    if coord1 in Switchables:
        Switchables.remove(coord1)
    if len(Switchables) == 1:
        return True
    else:
        return False

def switch_coordinates(coord1,Switchables,A):
    '''A function that switches two coordinates in
    a set of arrows'''
    if len(Switchables) != 1:
        print('ERROR : too many coordinates to choose from')
        return False
    else:
```

```
            coord2 = Switchables[0]
            a1 = find_arrow_coord(coord1,A)
            a2 = find_arrow_coord(coord2,A)
            i1 = a1.get_position()
            i2 = a2.get_position()
            C1 = a1.get_coordinates()
            C2 = a2.get_coordinates()
            for i in range(2):
                if C1[i] == coord1:
                    C1[i] = coord2
                if C2[i] == coord2:
                    C2[i] = coord1
            a1.set_coordinates(C1)
            a2.set_coordinates(C2)
            A[i1] = a1
            A[i2] = a2
            return True

    def compute_coordinates(coord1,Switchables,A):
        '''A function that switches two coordinates
        in a set of arrows'''
        if len(Switchables) != 1:
            print('ERROR : too many coordinates to choose from')
            return False
        else:
            coord2 = Switchables[0]
            a1 = find_arrow_coord(coord1,A)
            a2 = find_arrow_coord(coord2,A)
            i1 = a1.get_position()
            i2 = a2.get_position()
            C1 = a1.get_coordinates()
            C2 = a2.get_coordinates()
            coord = min(coord1,coord2)
            for i in range(2):
                if C1[i] == coord1:
                    C1[i] = coord
                if C2[i] == coord2:
                    C2[i] = coord
            a1.set_coordinates(C1)
            a2.set_coordinates(C2)
            A[i1] = a1
            A[i2] = a2
            return True
```

```
def same_foot(A,a):
    '''A function that computes if the arrow a has
    same foot coordinates as another arrow in A'''
    ans = False
    for arrow in A:
        if arrow != a
        and a.get_coordinates()[0] == arrow.get_coordinates()[0]:
            ans = True
    return ans

def same_head(A,a):
    '''A function that computes if the arrow a
    has same head coordinates as another arrow in A'''
    ans = False
    for arrow in A:
        if arrow != a
        and a.get_coordinates()[1] == arrow.get_coordinates()[1]:
            ans = True
    return ans


def name_arrows(a,b,c):
    '''A function that returns
    the name of three singular arrows in a singular RM3 diagram'''
    A = [a,b,c]
    for arrow in A:
        if same_head(A,arrow):
            if same_foot(A,arrow):
                d = arrow
            else:
                hm = arrow
        else:
            ml = arrow
    return d,hm,ml

def local_type(D):
    '''A function that returns the local type
    of a singular BraidData'''
    D1 = deepcopy(D)
    #First, we check that we are in a singular case
    if D1.get_Class() == 'regular':
        print('Erreur : ce n\' est pas un diagramme singulier')
    else:
        clas = D1.get_Class()
```

```
            a,b,c = D1.get_RM3_a(),D1.get_RM3_b(),D1.get_RM3_c()
            ba = a.get_braidelement()
            bb = b.get_braidelement()
            bc = c.get_braidelement()
            rm3 = (ba,bb,bc)
            lt = -1
            n = 4
            for i in range(1,n):
                if rm3 == (i+1,i,i+1) or rm3 == (-(i+1),-i,i+1)
                or rm3 == (i+1,-i,-(i+1)) or rm3 == (-i,i+1,i)
                or rm3 == (i,i+1,-i) or rm3 == (-i,-(i+1),-i):
                    lt = 1
            return lt

    def global_type(D):
        '''A function that returns the global type
        of a singular BraidData'''
        D1 = deepcopy(D)
        #First, we check that we are in a singular case
        if D1.get_Class() == 'regular':
            print('Erreur : ce n\' est pas un diagramme singulier')
        else:
            clas = D1.get_Class()
            a,b,c = D1.get_RM3_a(),D1.get_RM3_b(),D1.get_RM3_c()
            d,hm,ml = name_arrows(a,b,c)
            homd = d.get_homotopy()
            homhm = hm.get_homotopy()
            homml = ml.get_homotopy()
            if homd == homhm + homml:
                gt = -1
            else:
                gt = 1
        return gt
```

Classes for BraidData

```
In [ ]: class TorusGaussArrow:
            def __init__(self):
                self.__number = 'INIT'
                self.__position = 'INIT'
                self.__braidelement = 'INIT'
                self.__homotopy = 0
                self.__writhe = 0
                self.__coordinates = 'empty'
```

```
        self.__class = 'regular'
    def set_number(self,num):
        self.__number = num
    def set_position(self,pos):
        self.__position = pos
    def set_braidelement(self,t):
        self.__braidelement = t
    def set_homotopy(self,h):
        self.__homotopy = h
    def set_writhe(self,w):
        self.__writhe = w
    def set_coordinates(self,c):
        self.__coordinates = c
    def set_class(self,c):
        self.__class = c
    def get_number(self):
        return self.__number
    def get_position(self):
        return self.__position
    def get_braidelement(self):
        return self.__braidelement
    def get_homotopy(self):
        return self.__homotopy
    def get_writhe(self):
        return self.__writhe
    def get_coordinates(self):
        return self.__coordinates
    def get_class(self):
        return self.__class
    def get_infos(self):
        print('Number : ',self.get_number())
        print('Position in braid : ',self.get_position())
        print('Braid element : ',self.get_braidelement())
        print('Writhe : ',self.get_writhe())
        print('Homotopy : ',self.get_homotopy())
        print('Coordinates : ',self.get_coordinates() )
    def __eq__(self,A):
        if A.__class__.__name__ != "TorusGaussArrow" :
            print('Error for the operand == Unsupported type')
            return False
        else:
            if self.get_number() == A.get_number()
                  and self.get_writhe() == A.get_writhe()
                  and self.get_homotopy() == A.get_homotopy()
```

```
                    and self.get_coordinates() == A.get_coordinates() :
                return True
            else:
                return False

class TorusGaussCode :
    "A Gauss Diagram of a braid in the solid torus"
    def __init__(self,braid=0):
        self.__Arrows = []
        if braid == 0:
            self.__GaussDiagram = []
        else:
            L = Link(B(braid))
            self.__GaussDiagram = L.oriented_gauss_code()[0][0]
            for i in range(len(braid)):
                A = compute_arrow(braid,i,i+1)
                self.__Arrows.append(A)
    def set_GaussDiagram(self,G):
        self.__GaussDiagram = G
    def set_Arrows(self,A):
        self.__Arrows = A
    def get_GaussDiagram(self):
        return self.__GaussDiagram
    def get_Arrows(self):
        return self.__Arrows
    def get_infos(self):
        print('Gauss Diagram : ',self.get_GaussDiagram())
    def remove_arrow(self,i):
        #Removing the arrow in position i
        A = self.get_Arrows()
        for arrow in A:
            if arrow.get_position() == i:
                self.get_Arrows().remove(arrow)
                #Removing from the GaussDiagram
                j = arrow.get_number()
                self.get_GaussDiagram().remove(j)
                self.get_GaussDiagram().remove(-j)
    def plot_diagram(self):
        '''A method that plots the Torus Gauss diagram
        associated to the TorusGaussCode'''
        #Plotting the circle
        ax = plt.axes()
        radius = 1
        theta = linspace(0, 2*pi, 100)
```

```
        x1 = radius*cos(theta)
        x2 = radius*sin(theta)
        plt.plot(x1,x2,'black')
#         ax.set_aspect(1)
        plt.axis('scaled')
        plt.axis('off')
        #Plotting the base point
        plt.text(0, 0.92,'*',color='black',size = 20,ha='center')
        #Plotting the arrows
        n = 2*len(self.get_Arrows())+1
        for a in self.get_Arrows():
            if a.get_homotopy() == 1:
                color = 'red'
            elif a.get_homotopy() == 2:
                color = 'blue'
            elif a.get_homotopy() == 3:
                color = 'green'
            else:
                color = 'black'
            number = a.get_number()
            if a.get_writhe() > 0:
                sign = '+'
            else:
                sign = '-'
            c1,c2 = a.get_coordinates()
            fangle = (c1+1)*2*pi/n + pi/2
            hangle = (c2+1)*2*pi/n + pi/2
            plt.annotate('', xy = (cos(hangle), sin(hangle)) ,
                         xytext = (cos(fangle), sin(fangle)),
                            arrowprops = {'color': color,
                                          'arrowstyle' : '->'})
            #Labels
            r = 1.15
            plt.text(r*cos(hangle), r*sin(hangle),sign,color=color)
            plt.text(r*cos(fangle), r*sin(fangle),number,color=color)
    def plot_singular_diagram(self):
        '''A method that plots the Torus Gauss diagram
        associated to the TorusGaussCode'''
        #Plotting the circle
        ax = plt.axes()
        radius = 1
        theta = linspace(0, 2*pi, 100)
        x1 = radius*cos(theta)
        x2 = radius*sin(theta)
```

```
        plt.plot(x1,x2,'black')
#         ax.set_aspect(1)
        plt.axis('scaled')
        plt.axis('off')
        #Plotting the base point
        plt.text(0, 0.92,'*',color='black',size = 20,ha='center')
        #Plotting the arrows
        n = 2*len(self.get_Arrows())-2
        for a in self.get_Arrows():
            if a.get_homotopy() == 1:
                color = 'red'
            elif a.get_homotopy() == 2:
                color = 'blue'
            elif a.get_homotopy() == 3:
                color = 'green'
            else:
                color = 'black'
            number = a.get_number()
            if a.get_writhe() > 0:
                sign = '+'
            else:
                sign = '-'
            c1,c2 = a.get_coordinates()
            fangle = (c1+1)*2*pi/n + pi/2
            hangle = (c2+1)*2*pi/n + pi/2
            #Labels
            r = 1.15
            plt.text(r*cos(hangle), r*sin(hangle),sign,color=color)
            if a.get_class() == 'regular':
                plt.text(r*cos(fangle), r*sin(fangle),number,color=color)
                plt.annotate('', xy = (cos(hangle), sin(hangle)) ,
                             xytext = (cos(fangle), sin(fangle)),
                            arrowprops = {'color': color,
                                          'arrowstyle' : '->'})
            else:
                alpha = (max(fangle,hangle)-min(fangle,hangle))/2
                nangle = pi/2 + hangle - alpha
                m = len(self.get_Arrows())
                nr = 0.05*log(m)
                nx = r*cos(hangle)+nr*cos(nangle)
                ny = r*sin(hangle)+nr*sin(nangle)
                plt.text(nx,ny,number,color=color,weight='bold')
                plt.annotate('', xy = (cos(hangle), sin(hangle)) ,
                             xytext = (cos(fangle), sin(fangle)),
```

```
                                arrowprops = {'color': color,'width' : 0.7,
                                              'headwidth':5,'headlength':5})

    def __eq__(self,G):
        if G.__class__.__name__ != "TorusGaussCode" :
            print('Error : Could not test the operand ==.
                  Unsupported type for the second argument')
            return False
        else:
            if self.get_GaussDiagram() == G.get_GaussDiagram()
                  and self.get_Arrows() == G.get_Arrows():
                return True
            else:
                return False

class BraidData :
    "The datas of a braid, with the braid closure, arrows, TorusGaussCode"
    def __init__(self,braid=0):
        if braid == 0:
            self.BraidClosure = []
            self.ArrowsNumber = []
            self.TorusGaussCode = TorusGaussCode()
            self.length = 0
            #To treat singular diagrams
            self.sign = 1
            self.Class = 'regular'
            self.RM3_a = TorusGaussArrow()
            self.RM3_b = TorusGaussArrow()
            self.RM3_c = TorusGaussArrow()
        else:
            L = Link(B(braid))
            if L.is_knot():
                self.TorusGaussCode = TorusGaussCode(braid)
                self.update_infos()
                self.Class = 'regular'
            else:
                print('Error : the braid is not a knot')
    def update_infos(self):
        '''A method that updates the BraidClosure and Numberlist from the arrows'''
        #Computing the BraidClosure and the Numberlist from the Arrows
        braidclosure = list(range(len(self.get_TorusGaussCode().get_Arrows())))
        numlist = list(range(len(self.get_TorusGaussCode().get_Arrows())))
        for a in self.get_TorusGaussCode().get_Arrows():
            e = a.get_braidelement()
```

```
            n = a.get_number()
            i = a.get_position()
            braidclosure[i] = e
            numlist[i] = n
        self.set_BraidClosure(braidclosure)
        self.set_ArrowsNumber(numlist)
    def update_arrows(self):
        '''A method that updates the arrows from the Numberlist'''
        numlist = self.get_ArrowsNumber()
        for a in self.get_TorusGaussCode().get_Arrows():
            for i in range(len(numlist)):
                if a.get_number() == numlist[i]:
                    a.set_position(i)
    def update_diagram(self):
        '''A method that updates the Gauss Diagram from the set of arrows'''
        A = self.get_TorusGaussCode().get_Arrows()
        GD = TorusGaussCode_reconstruct(A)
        self.set_TorusGaussCode(GD)
    def set_BraidClosure(self,braid):
        self.BraidClosure = braid
        self.length = len(self.BraidClosure)
    def set_sign(self,s):
        self.sign = s
    def set_Class(self,clas):
        self.Class = clas
    def set_RM3_a(self,a):
        self.RM3_a = a
    def set_RM3_b(self,b):
        self.RM3_b = b
    def set_RM3_c(self,c):
        self.RM3_c = c
    def set_ArrowsNumber(self,num):
        self.ArrowsNumber = num
        self.length = len(self.BraidClosure)
    def set_TorusGaussCode(self,T):
        self.TorusGaussCode = T
    def get_BraidClosure(self):
        return self.BraidClosure
    def get_sign(self):
        return self.sign
    def get_Class(self):
        return self.Class
    def get_RM3_a(self):
        return self.RM3_a
```

```
def get_RM3_b(self):
    return self.RM3_b
def get_RM3_c(self):
    return self.RM3_c
def get_ArrowsNumber(self):
    return self.ArrowsNumber
def get_TorusGaussCode(self):
    return self.TorusGaussCode
def get_length(self):
    self.length = len(self.BraidClosure)
    return self.length
def get_infos(self):
    self.update_infos()
    print('Braid length : ',self.get_length())
    print('Braid closure : ',self.get_BraidClosure())
    print('Arrows number : ',self.get_ArrowsNumber())
    self.TorusGaussCode.get_infos()
def remove_arrow(self,i):
    '''A method that removes the arrow in position i'''
    #Deleting arrow in position i
    self.get_TorusGaussCode().remove_arrow(i)
    self.get_BraidClosure().pop(i)
    self.get_ArrowsNumber().pop(i)
    self.update_arrows()
    self.update_infos()
def plot_diagram(self):
    '''A method that plot the Torus Gauss diagram'''
    if self.get_Class() == 'regular':
        self.update_diagram()
        self.get_TorusGaussCode().plot_diagram()
        braid = self.get_BraidClosure()
        braidtitle = 'Braid :  %s'%braid
        plt.text(0,1.5,braidtitle,color='black',ha='center')
        arrowsnum = self.get_ArrowsNumber()
        arrowsnumtitle = 'Arrows : %s'%arrowsnum
        plt.text(0,1.3,arrowsnumtitle,color='black',ha='center')
    else:
        self.update_diagram()
        self.get_TorusGaussCode().plot_singular_diagram()
        move = self.get_Class()
        plt.text(0,1.7,move,color='black',
                 weight='bold',ha='center')
        braid = self.get_BraidClosure()
        braidtitle = 'Braid :  %s'%braid
```

```
            plt.text(0,1.5,braidtitle,color='black',
                     weight='bold',ha='center')
            arrowsnum = self.get_ArrowsNumber()
            arrowsnumtitle = 'Arrows : %s'%arrowsnum
            plt.text(0,1.3,arrowsnumtitle,color='black',
                     weight='bold',ha='center')
    def CommutationMove(self,i):
        ''' A method that commutes the arrows
        in position i and i+1 (Commutation Move)'''
        #Testing if the move is allowed
        A = self.get_TorusGaussCode().get_Arrows()
        a = find_arrow_position(A,i).get_braidelement()
        b = find_arrow_position(A,i+1).get_braidelement()
        if abs(abs(a)-abs(b)) > 1 :
            BC = self.get_BraidClosure()
            BC[i],BC[i+1] = BC[i+1],BC[i]
            self.set_BraidClosure(BC)
            AN = self.get_ArrowsNumber()
            AN[i],AN[i+1] = AN[i+1],AN[i]
            self.set_ArrowsNumber(AN)
            A[i],A[i+1] = A[i+1],A[i]
            self.update_arrows()
            self.update_infos()
        else:
            print('ERROR : the braid elements don\'t correspond to a CM')
            print('Position of arrows : ',i,i+1)
#             print('Arrows : ',a.get_number(),b.get_number())
            print('Braid elements : ',a,b)
    def ReidemeisterMoveII_delete(self,i):
        '''A method that removes the arrows
        in position i and i+1 (Reidemeister Move II)'''
        length = self.get_length()
        j = i+1
        #Checking for particular case
        if i == length-1:
            j = 0
        for a in self.get_TorusGaussCode().get_Arrows():
            n1 = self.get_ArrowsNumber()[i]
            if a.get_position() == i:
                e1 = a.get_braidelement()
                n1 = a.get_number()
                f1,h1 = a.get_coordinates()
            if a.get_position() == j:
                e2 = a.get_braidelement()
```

```
            n2 = a.get_number()
            f2,h2 = a.get_coordinates()
    #Testing if the braid elements are eligible to a RM2
    if e1 == -e2 :
        self.remove_arrow(i)
        if i == length-1:
            self.remove_arrow(0)
        else:
            self.remove_arrow(i)
        #Updating the coordinates for the other arrows
        #Particular case 1
        if min(f1,f2) == 0
              and max(f1,f2) == len(self.get_TorusGaussCode().get_GaussDiagram())
            for a in self.get_TorusGaussCode().get_Arrows():
                C = a.get_coordinates()
                if C[0] < min(h1,h2):
                    C[0] = C[0] - 1
                if C[0] > max(h1,h2):
                    C[0] = C[0] - 3
                if C[1] < min(h1,h2):
                    C[1] = C[1] - 1
                if C[1] > max(h1,h2):
                    C[1] = C[1] - 3
            self.update_arrows()
        #Particular case 2
        elif min(h1,h2) == 0
              and max(h1,h2) == len(self.get_TorusGaussCode().get_GaussDiagram())
            for a in self.get_TorusGaussCode().get_Arrows():
                C = a.get_coordinates()
                if C[0] < min(f1,f2):
                    C[0] = C[0] - 1
                if C[0] > max(f1,f2):
                    C[0] = C[0] - 3
                if C[1] < min(f1,f2):
                    C[1] = C[1] - 1
                if C[1] > max(f1,f2):
                    C[1] = C[1] - 3
            self.update_arrows()
        #Regular case
        else:
            for a in self.get_TorusGaussCode().get_Arrows():
                C = a.get_coordinates()
                if C[0] > min(max(f1,f2),max(h1,h2))
              and C[0] < max(min(f1,f2),min(h1,h2)):
```

```
                    C[0] = C[0] - 2
                if C[1] > min(max(f1,f2),max(h1,h2))
               and C[1] < max(min(f1,f2),min(h1,h2)):
                    C[1] = C[1] - 2
                if C[0] > max(f1,h1,h2,f2):
                    C[0] = C[0] - 4
                if C[1] > max(f1,h1,h2,f2):
                    C[1] = C[1] - 4
            self.update_arrows()
    else:
        print('ERROR : the braid elements don\'t correspond to a RM2')
        print('Position of the arrows')
        print(i,i+1)
        print('Braid elements')
        print(e1,e2)
    return deepcopy(self)
def ReidemeisterMoveII_create(self,e,i):
    '''A method that adds the arrows corresponding to
    the braid elements e,-e in position i and i+1 (Reidemeister Move II)'''
    #Updating the BraidClosure and the ArrowsNumber
    self.get_BraidClosure().insert(i,-e)
    self.get_BraidClosure().insert(i,e)
    m = max(self.get_ArrowsNumber())
    self.get_ArrowsNumber().insert(i,m+2)
    self.get_ArrowsNumber().insert(i,m+1)
    #We can't compute the arrows from the BraidClosure as
    #an RM2 will automatically be cancelled in the Gauss diagram
    #We have to compute them manually
    braid = self.get_BraidClosure()
    s = sign(e)
    h = any_homotopy_class(braid,i)
    #Computations for the coordinates
    C1,C2 = coordinates_RM2(self,e,i)
    f1,h1 = C1
    f2,h2 = C2
    #Arrow a1 corresponding to e, a2 corresponds to -e
    #Position will be updated from the update_arrow method
    a1 = TorusGaussArrow()
    a1.set_number(m+1)
    a1.set_braidelement(e)
    a1.set_writhe(s)
    a1.set_homotopy(h)
    a1.set_coordinates(C1)
    a2 = TorusGaussArrow()
```

```
        a2.set_number(m+2)
        a2.set_braidelement(-e)
        a2.set_writhe(-s)
        a2.set_homotopy(h)
        a2.set_coordinates(C2)
        #Updating the coordinates for the other arrows
        for a in self.get_TorusGaussCode().get_Arrows():
            C = a.get_coordinates()
            if C[0] >= min(f1,h1,f2,h2):
                C[0] = C[0] + 2
            if C[1] >= min(f1,h1,f2,h2):
                C[1] = C[1] + 2
            if C[0] >= min(max(f1,h1),max(f2,h2)):
                C[0] = C[0] + 2
            if C[1] >= min(max(f1,h1),max(f2,h2)):
                C[1] = C[1] + 2
            a.set_coordinates(C)
        self.get_TorusGaussCode().get_Arrows().insert(i,a2)
        self.get_TorusGaussCode().get_Arrows().insert(i,a1)
        self.update_arrows()
        #Updating the Gauss diagram
        self.update_diagram()
        return deepcopy(self)
    def ReidemeisterMoveIII(self,i,j,k):
        '''A method that performs a RM3 for arrows
        in position i,j and k (Reidemeister Move III)'''
        D1 = deepcopy(self)
        #Getting the 3 arrows to switch
        A = self.get_TorusGaussCode().get_Arrows()
        ai = find_arrow_position(A,i)
        aj = find_arrow_position(A,j)
        ak = find_arrow_position(A,k)
        #Getting their coordinates
        fi,hi = ai.get_coordinates()
        fj,hj = aj.get_coordinates()
        fk,hk = ak.get_coordinates()
        #L is the list of coordinates to be switched
        L = [fi,hi,fj,hj,fk,hk]
        #Operating the switch on the braid
        numb = len(self.get_TorusGaussCode().get_GaussDiagram())
        while len(L) > 0 :
            #For each coordinates,
            for coord in L:
                a = find_arrow_coord(coord,A)
```

```
            for c in a.get_coordinates():
                if c != coord:
                    coord2 = c
            #We check if it is switchable with only one other coordinate
            if is_switchable_once(coord,coord2,L,numb):
                Switchables = find_switchables(coord,L,numb)
                for coord1 in Switchables:
                    if coord1 not in L:
                        Switchables.remove(coord1)
                #Once we've switch them, we remove them from the list L
                switch_coordinates(coord,Switchables,A)
                L.remove(coord)
                coord3 = Switchables[0]
                L.remove(coord3)
    #Updating the braid elements in the arrows:
    ei = ai.get_braidelement()
    ej = aj.get_braidelement()
    ek = ak.get_braidelement()
    ei,ej,ek = sign(ei)*abs(ej),sign(ej)*abs(ei),sign(ek)*abs(ej)
    ai.set_braidelement(ei)
    aj.set_braidelement(ej)
    ak.set_braidelement(ek)
    #Updating the position in the arrows
    pi = ai.get_position()
    pk = ak.get_position()
    ai.set_position(k)
    ak.set_position(i)
    A[pi],A[pk] = A[pk],A[pi]
    self.get_TorusGaussCode().set_Arrows(A)
    #Updating the Gauss diagram
    self.update_diagram()
    #Updating the ArrowsNumber
    Numlist = self.get_ArrowsNumber()
    Numlist[i],Numlist[k] = Numlist[k],Numlist[i]
    self.set_ArrowsNumber(Numlist)
    #Updating the BraidClosure
    BC = self.get_BraidClosure()
    BC[i],BC[j],BC[k] = sign(BC[k])*abs(BC[j]),sign(BC[j])*abs(BC[i]),sign(BC[i])
    self.set_BraidClosure(BC)
    self.update_arrows()

    #Generating the singular diagram
    L = [fi,hi,fj,hj,fk,hk]
    A = D1.get_TorusGaussCode().get_Arrows()
```

```
ai = find_arrow_position(A,i)
ai.set_class('singular')
aj = find_arrow_position(A,j)
aj.set_class('singular')
ak = find_arrow_position(A,k)
ak.set_class('singular')
ni = ai.get_number()
nj = aj.get_number()
nk = ak.get_number()
clas = 'RM3 : %d %d %d'%(ni,nj,nk)
D1.set_Class(clas)
D1.set_RM3_a(ai)
D1.set_RM3_b(aj)
D1.set_RM3_c(ak)
while len(L) > 0 :
    #For each coordinates,
    for coord in L:
        a = find_arrow_coord(coord,A)
        for c in a.get_coordinates():
            if c != coord:
                coord2 = c
        #We check if it is switchable with only one other coordinate
        if is_switchable_once(coord,coord2,L,numb):
            Switchables = find_switchables(coord,L,numb)
            for coord1 in Switchables:
                if coord1 not in L:
                    Switchables.remove(coord1)
            #Once we've switch them, we remove them from the list L
            compute_coordinates(coord,Switchables,A)
            L.remove(coord)
            coord3 = Switchables[0]
            L.remove(coord3)
#Updating the coordinates for the other arrows
L = [fi,hi,fj,hj,fk,hk]
L.sort()
for a in D1.get_TorusGaussCode().get_Arrows():
    C = a.get_coordinates()
    if C[0] > L[1] and C[0] <= L[2] :
        C[0] = C[0] - 1
    if C[0] >= L[3] and C[0] <= L[4]:
        C[0] = C[0] - 2
    if C[0] >= L[5]:
        C[0] = C[0] - 3
    if C[1] > L[1] and C[1] <= L[2] :
```

```
                    C[1] = C[1] - 1
                if C[1] >= L[3] and C[1] <= L[4]:
                    C[1] = C[1] - 2
                if C[1] >= L[5]:
                    C[1] = C[1] - 3
                a.set_coordinates(C)
            #Computing the RM3 sign
            lt = local_type(D1)
            gt = global_type(D1)
            D1.set_sign(lt*gt)
            return D1
        def plot_rot(self,GD,figuresize=(6.4, 4.8),figuredpi=100):
            D1 = deepcopy(self)
            #figuresize = (5,5)
            #figuredpi = 100
            roti,rot_sing,Inv = rot(D1,GD)
            for i in range(len(roti)):
                fig = plt.figure(i,figsize=figuresize,dpi=figuredpi,frameon=True)
                roti[i].plot_diagram()
            print('Invariant : {} '.format(Inv))
        def plot_singular_rot(self,GD,figuresize=(6.4, 4.8),figuredpi=100):
            D1 = deepcopy(self)
            #figuresize = (20,20)
            #figuredpi = 100
            roti,rot_sing,Inv = rot(D1,GD)
            for i in range(len(rot_sing)):
                fig = plt.figure(i,figsize=figuresize,dpi=figuredpi,frameon=True)
                rot_sing[i].plot_diagram()
            print('Invariant : {} '.format(Inv))
        def plot_braid_diagram(self,size=0.25,res=200):
            S = self.get_BraidClosure()
            plot_braid(S,size=size,res=res)
```

1.1.2 I.b. Generating all the 1-cocycles

```
In [ ]: def compute_quarter_direction(ai,trigausscode):
            '''A function that compute the quarter for
            the Gauss Arrow ai in trigausscode'''
            fa,ha = ai.get_coordinates()
            #Searching for the foot
            for j in range(1,len(trigausscode)):
                if trigausscode[fa-j] == 'A':
                    Q1 = 0
                    break
```

```
            elif trigausscode[fa-j] == 'B':
                Q1 = 1
                break
            elif trigausscode[fa-j] == 'C':
                Q1 = 2
                break
        #Searching for the head
        for j in range(1,len(trigausscode)):
            if trigausscode[ha-j] == 'A':
                Q2 = 0
                break
            elif trigausscode[ha-j] == 'B':
                Q2 = 1
                break
            elif trigausscode[ha-j] == 'C':
                Q2 = 2
                break
        direction = '+'
        if Q1 == Q2 and fa > ha :
                direction = '-'
        return [Q1,Q2],direction

    def tri_compute_arrows(trigausscode,homotopy):
        '''A function that computes the set of
        Gauss Arrows from a TriGaussCode and a Homotopy vector'''
        arrows = []
        if trigausscode != []:
            for i in range(1,(len(trigausscode)-3)/2 + 1):
                ai = TriGaussArrow()
                for j in range(len(trigausscode)):
                    if trigausscode[j] == i:
                        foot_ai = j
                    elif trigausscode[j] == -i:
                        head_ai = j
                ai.set_coordinates([foot_ai,head_ai])
                ai.set_homotopy(homotopy[i-1])
                ai.set_name(i)
                ai.set_foot(i)
                ai.set_head(-i)
                quarter,direction = compute_quarter_direction(ai,trigausscode)
                ai.set_quarter(quarter)
                ai.set_direction(direction)
                arrows.append(ai)
        return arrows
```

```
def circle(center, radius,color):
    ax = plt.gca()
    theta = linspace(0, 2*pi, 100)
    x1 = center[0]+radius*cos(theta)
    x2 = center[1]+radius*sin(theta)
    plt.plot(x1,x2,color)
#     ax.set_aspect(1)
    plt.axis('scaled')
    return 0

def gauss_diagram(dtype,center,printerr = False):
    ''' A function that plot a triple point of dtype ,
    dtype must be of format ((int,int),'str')'''

    #Example of dtype : dtype = ((1,2),'-')
    n = 4
    x = center[0]
    y = center[1]
    circle((x,y),1,'black')
    #Extracting the data of the type
    sign = dtype[2]

    #Checking for errors
    err = 1
    if sign != '-' and sign != '+' :
        if printerr == True:
            print('The sign must be - or +')
        err = 0
        return 'Error'
    if err != 0 :

        #Computing the homotopy of d
        ml = dtype[0]
        hm = dtype[1]
        if sign == '-' :
            d = ml + hm
        else :
            d = ml + hm - n

        #Computing color for ml
        colorml = 'black'
        if ml == 1 :
            colorml = 'red'
```

```
elif ml == 2 :
    colorml = 'blue'
elif ml == 3 :
    colorml = 'green'
else :
    if printerr == True:
        print('Error : ml doesn\'t have a correct homotopy')

#Computing color for hm
colorhm = 'black'
if hm == 1 :
    colorhm = 'red'
elif hm == 2 :
    colorhm = 'blue'
elif hm == 3 :
    colorhm = 'green'
else :
    if printerr == True:
        print('Error : hm doesn\'t have a correct homotopy')

#Computing color for d
colord = 'black'
if d == 1 :
    colord = 'red'
elif d == 2 :
    colord = 'blue'
elif d == 3 :
    colord = 'green'
else :
    if printerr == True:
        print('Error : Such a type doesn\'t exist')

#Computing colors for the arrows
down = colorml
if sign =='-' :
    left = colorhm
    right = colord
else :
    left = colord
    right = colorhm

#Plotting the arrows
#arrow left
plt.annotate('', xy = (x,y+1) ,
```

```
                    xytext = (x+cos(pi*7/6), y+sin(pi*7/6)),
                    arrowprops = {'color': left,'arrowstyle' : '->'})
        #arrow right
        plt.annotate('', xy = (x,y+1),
                    xytext = (x+cos(pi*11/6), y+sin(pi*11/6)),
                    arrowprops = {'color': right,'arrowstyle' : '->'})
        if sign == '-' :
            #arrow down
            plt.annotate('', xy = (x+cos(pi*7/6), y+sin(pi*7/6)),
                        xytext = (x+cos(pi*11/6), y+sin(pi*11/6)),
                        arrowprops = {'color': down,'arrowstyle' : '->'})
        else :
            #arrow down
            plt.annotate('', xy = (x+cos(pi*11/6), y+sin(pi*11/6)),
                        xytext  = (x+cos(pi*7/6), y+sin(pi*7/6)),
                        arrowprops = {'color': down,'arrowstyle' : '->'})
    return 0

def gauss_diagram_sign(dtype,sign,position,printerr=False) :
    '''A function that plots the gauss diagram of
    a triple point of type dtype and an arrow of homotopy h'''
    #Q1 and Q2 are the concerned quadrant,
    #direct gives the direction of the arrow (default is +)
    #position is where the sign is plot
    #Example : dtype = ((1,1),'-') , Q1 = 1, Q2 = 1, direct = '-', h = 2

    #Plotting the sign
    x = position[0] + 1 - 0.2
    y = position[1] - 0.1
    plt.text(x,y,sign,color='black',size='xx-large')

    center = (x+2+0.2 ,y+0.1)
    gauss_diagram(dtype,center,printerr)
    plt.axis('off')
    plt.axis('scaled')
    return 0

def farrow(hp,fp,h,center,printerr = False):
    '''plot in an already plot circle of given center,
    an arrow of homotopy h in position hp,fp'''

    #hp = head position, fp = foot position, h = homotopy
    #Exemple : 1,2,1,(0,1)
```

```
        #Testing for errors in positions
        err = 1
        if hp == fp :
            if printerr == True:
                print("Head and foot cannot be in the same point")
            err = 0
            return 'Error'
        if hp == 11 or hp == 3 or hp == 7 or fp == 11 or fp == 3 or fp == 7 :
            if printerr == True:
                print("Arrow cannot be added on the singular points")
            err = 0
            return 'Error'
        if err != 0 :
            x = center[0]
            y = center[1]
            #Computing the head and foot position (ha = head angle, fa = foot angle)
            head = (x+cos(hp),y+sin(hp))
            foot = (x+cos(fp),y+sin(fp))
            #Determining the color
            color = 'black'
            if h == 1 :
                color = 'red'
            if h == 2 :
                color = 'blue'
            if h == 3 :
                color = 'green'

            #Plotting the arrow
            plt.annotate('', xy = head, xytext  = foot,
                            arrowprops = {'color': color,'arrowstyle' : '->'})
        return 0

    def compute_ends_for_quarters(trigausscode):
        '''A function that computes for each quarter of
        a Gauss Diagram the number of ends'''
        #Quarter 0
        nb0 = 0
        for i in range(1,len(trigausscode)):
            if trigausscode[i] == 'B':
                break
            nb0 = nb0 + 1
        #Quarter 1
        nb1 = 0
        for i in range(nb0+2,len(trigausscode)):
```

```
        if trigausscode[i] == 'C':
            break
        nb1 = nb1 + 1
    #Quarter 2
    nb2 = len(trigausscode) - 3 - nb0 - nb1
    return nb0,nb1,nb2

def compute_position(arrow,nb0,nb1,nb2):
    '''A function that computes the position of
    Gauss Arrow arrow for plotting'''
    #nb0, nb1 and nb2 are the number of ends in the quarters
    #Getting the coordinates
    fc,hc = arrow.get_coordinates()
    Q1,Q2 = arrow.get_quarter()
    #if foot is in Q0
    if Q1 == 0:
        n0 = nb0 + 1
        fp = 11*pi/6 + fc*2*pi/(3*n0)
    #if foot is in Q1
    elif Q1 == 1:
        n1 = nb1 + 1
        fc = fc - nb0 - 1
        fp = pi/2 + fc*2*pi/(3*n1)
    #if foot is in Q2
    elif Q1 == 2:
        n2 = nb2 + 1
        fc = fc - nb0 - 1 - nb1 - 1
        fp = 7*pi/6 + fc*2*pi/(3*n2)
    #if head is in Q0
    if Q2 == 0:
        n0 = nb0 + 1
        hp = 11*pi/6 + hc*2*pi/(3*n0)
    #if head is in Q1
    elif Q2 == 1:
        n1 = nb1 + 1
        hc = hc - nb0 - 1
        hp = pi/2 + hc*2*pi/(3*n1)
    #if head is in Q2
    elif Q2 == 2:
        n2 = nb2 + 1
        hc = hc - nb0 - 1 - nb1 - 1
        hp = 7*pi/6 + hc*2*pi/(3*n2)
    return fp,hp
```

```
def compute_coordinates_sing(dtype,trigausscode):
    '''A function that computes the coordinates for
    the singular arrows in a TriGaussDiagram'''
    sign = dtype[2]
    #Getting the coordinates for B and C
    A = 0
    for i in range(len(trigausscode)):
        if trigausscode[i] == 'B':
            B = i
        elif trigausscode[i] == 'C':
            C = i
    #Computing the coordinates for the singular arrows
    fd = C
    hd = B
    fhm = A
    hhm = B
    fml = C
    hml = A
    if sign == '-':
        fml,hml = hml,fml
        fhm,fd = fd,fhm
    return [fd,hd],[fhm,hhm],[fml,hml]

def tri_compute_trigausscode(arrows):
    '''A function that computes a trigausscode from a set of arrows'''
    trigausscode =  l = [0] * (2*len(arrows) + 3)
    for a in arrows:
        i = a.get_name()
        fa,ha = a.get_coordinates()
        trigausscode[fa] = i
        trigausscode[ha] = -i
    trigausscode[0] = 'A'
    for i in range(len(trigausscode)):
        if trigausscode[i] == 0:
            trigausscode[i] = 'B'
            break
    for i in range(len(trigausscode)):
        if trigausscode[i] == 0:
            trigausscode[i] = 'C'
            break
    return trigausscode

def tri_compute_trigausscode_braid(arrows):
    '''A function that computes a trigausscode from a set of arrows'''
```

```
            trigausscode =  l = [0] * (2*len(arrows) + 3)
            for a in arrows:
                i = a.get_number()
                fa,ha = a.get_coordinates()
                trigausscode[fa] = i
                trigausscode[ha] = -i
            trigausscode[0] = 'A'
            for i in range(len(trigausscode)):
                if trigausscode[i] == 0:
                    trigausscode[i] = 'B'
                    break
            for i in range(len(trigausscode)):
                if trigausscode[i] == 0:
                    trigausscode[i] = 'C'
                    break
            return trigausscode
```

```
In [ ]: class TriGaussArrow:
            "An arrow in a Gauss diagram of a triple point"
            def __init__(self,coordinates=('fc','hc'),quarter=('INIT','INIT'),
                         direction='INIT',homotopy=0,name='default',foot='INIT',
                         head='INIT',nomenclature='INIT'):
                self.__coordinates = coordinates
                self.__quarter = quarter
                self.__direction = direction
                self.__homotopy = homotopy
                self.__name = name
                self.__foot = foot
                self.__head = head
                self.__nomenclature = nomenclature
                self.update()
            def update(self):
                self.__nomenclature = [self.get_name(),
                                       self.get_coordinates(),self.get_homotopy()]
                if type(self.get_name()) == int:
                    self.__nomenclature.append(self.get_quarter())
                if self.get_quarter()[0] == self.get_quarter()[1]:
                    self.__nomenclature.append(self.get_direction())
            def set_name(self,name):
                self.__name = name
                self.update()
            def set_foot(self,foot):
                self.__foot = foot
            def set_head(self,head):
```

```
        self.__head = head
    def set_coordinates(self,coordinates):
        self.__coordinates = coordinates
        self.update()
    def set_quarter(self,quarter):
        self.__quarter = quarter
        self.update()
    def set_direction(self,direction):
        self.__direction = direction
        self.update()
    def set_homotopy(self,h):
        self.__homotopy = h
        self.update()
    def get_coordinates(self):
        return self.__coordinates
    def get_quarter(self):
        return self.__quarter
    def get_direction(self):
        return self.__direction
    def get_homotopy(self):
        return self.__homotopy
    def get_name(self):
        return self.__name
    def get_foot(self):
        return self.__foot
    def get_head(self):
        return self.__head
    def get_nomenclature(self):
        return self.__nomenclature
    def get_infos(self):
        print('Nomenclature :',self.get_nomenclature())
        print('Name :',self.get_name())
        print('Coordinates :',self.get_coordinates())
        print('Foot : {}, Head : {}'.format(self.get_foot(),self.get_head()))
        print('Homotopy class :',self.get_homotopy())
        print('Quarters :',self.get_quarter())
        if self.get_quarter()[0] == self.get_quarter()[1]:
                print('Direction :',self.get_direction)
    def __eq__(self, A):
        if A.__class__.__name__ != "TriGaussArrow" :
            print('Error for == Unsupported type')
            return False
        else :
            if self.get_coordinates() == A.get_coordinates()
```

```
                and self.get_name() == A.get_name()
                and self.get_homotopy() == A.get_homotopy():
                    return True
                else :
                    return False
        def __str__(self):
            return str(self.get_nomenclature())
        def __repr__(self):
            return str(self)

    class CoeffGaussDiagramTri:
        "A signed Gauss Diagram of a triple point"
        def __init__(self,dtype=[0,0,'INIT'],trigausscode=[],homotopy=[],
                     coeff = 0,d = [1,1,'+','d'],hm = [1,1,'+','hm'],
                     ml = [1,1,'+','ml'],printerrors = False):
            #Checking for errors
            if len(trigausscode) == 2*len(homotopy)+3
            or (trigausscode == [] and homotopy == []):
                self.__type = dtype
                self.__arrows = tri_compute_arrows(trigausscode,homotopy)
                self.__coefficient = coeff
                self.__printerrors = printerrors
                self.__d = d
                self.__hm = hm
                self.__ml = ml
                self.__trigausscode = trigausscode
                self.__homotopy = homotopy
                if (trigausscode != [] or homotopy != []):
                    self.update_type()
                    self.compute_name()
                else:
                    self.__name = 'INIT'
            else:
                print('Error : Gauss Code and Homotopy vector don\'t have same length')
        def compute_name(self):
            if self.get_coefficient() == -1:
                sgn = '-'
            elif self.get_coefficient() == 1:
                sgn = '+'
            else:
                sgn = self.get_coefficient()
            self.__name = '%s (%s,%s,%s)'%(sgn,self.get_type(),
                                           self.get_trigausscode(),self.get_homotopy())
        def update_type(self):
```

```
        self.__d = TriGaussArrow()
        self.__hm = TriGaussArrow()
        self.__ml = TriGaussArrow()
        ml,hm,sign = self.get_type()
        if sign == '-' :
            self.__d.set_homotopy(ml + hm)
            self.__d.set_foot('A')
            self.__d.set_head('B')
            self.__hm.set_foot('C')
            self.__hm.set_head('B')
            self.__ml.set_foot('A')
            self.__ml.set_head('C')
        else :
            self.__d.set_homotopy(ml + hm - Nbraid)
            self.__d.set_foot('C')
            self.__d.set_head('B')
            self.__hm.set_foot('A')
            self.__hm.set_head('B')
            self.__ml.set_foot('C')
            self.__ml.set_head('A')
        coordd,coordhm,coordml = compute_coordinates_sing(self.get_type(),
                                                          self.get_trigausscode())
        self.__d.set_name('d')
        self.__d.set_direction(sign)
        self.__d.set_coordinates(coordd)
        self.__d.set_quarter(['d','d'])
        self.__hm.set_homotopy(hm)
        self.__hm.set_name('hm')
        self.__hm.set_direction(sign)
        self.__hm.set_coordinates(coordhm)
        self.__hm.set_quarter(['hm','hm'])
        self.__ml.set_homotopy(ml)
        self.__ml.set_name('ml')
        self.__ml.set_direction(sign)
        self.__ml.set_coordinates(coordml)
        self.__ml.set_quarter(['ml','ml'])
    def set_type(self,ml,hm,sign):
        self.__type = [ml,hm,sign]
        self.update_type()
        self.compute_name()
    def set_arrows(self,arrows):
        self.__arrows = arrows
        self.compute_name()
    def set_coefficient(self,coeff):
```

```
        self.__coefficient = coeff
        self.compute_name()
    def set_printerrors(self,printerr):
        self.__printerrors = printerr
    def set_trigausscode(self,trigausscode):
        self.__trigausscode = trigausscode
    def set_homotopy(self,homotopy):
        self.__homotopy = homotopy
    def get_type(self):
        return self.__type
    def get_arrows(self):
        return self.__arrows
    def get_coefficient(self):
        return self.__coefficient
    def get_d(self):
        return self.__d
    def get_hm(self):
        return self.__hm
    def get_ml(self):
        return self.__ml
    def get_name(self):
        return self.__name
    def get_printerrors(self):
        return self.__printerrors
    def get_trigausscode(self):
        return self.__trigausscode
    def get_homotopy(self):
        return self.__homotopy
    def get_arrowsinfos(self):
        for a in self.get_arrows():
            a.get_infos()
    def get_infos(self):
        print('Type : ',self.get_type())
        print('Coefficient : ',self.get_coefficient())
        print('d : ',self.get_d())
        print('hm : ',self.get_hm())
        print('ml : ',self.get_ml())
        print('Name : ',self.get_name())
        print('Printerrors : ',self.get_printerrors())
        self.get_arrowsinfos()
    def plot(self,position=(0,0),solo=True,nbfigure='INIT'):
        if self.get_coefficient() != 0:
            if nbfigure != 'INIT':
                fig = plt.figure(nbfigure)
```

```
            #Updating the sign to plot
            if self.get_coefficient() == -1 :
                sign = '-'
            elif self.get_coefficient() == +1:
                sign = '+'
            elif self.get_coefficient() > 0 :
                sign = '+ {}'.format(self.get_coefficient())
            else:
                sign = self.get_coefficient()
            #Plotting the triple point
            dtype = self.get_type()
            printerr = self.get_printerrors()
            gauss_diagram_sign(dtype,sign,position,printerr)
            #Computing number of ends for each quarter
            nb0,nb1,nb2 = compute_ends_for_quarters(self.get_trigausscode())
            #Plotting the arrows
            x = position[0] + 3
            y = position[1]
            for arrow in self.get_arrows():
                #Computing the positions
                fp,hp = compute_position(arrow,nb0,nb1,nb2)
                h = arrow.get_homotopy()
                farrow(hp,fp,h,[x,y],printerr)
            if solo == True:
                ax = plt.gca()
                xlim = plt.xlim(position[0],position[0]+4.1)
                ylim = plt.ylim(position[1]-1.2,position[1]+1.2)
        elif self.get_printerrors():
            print('{} is empty'.format(self))
    def __eq__(self, D):
        if D == 0:
            return self.get_coefficient() == 0
        elif D.__class__.__name__ != "CoeffGaussDiagramTri" :
            print('Error for == Unsupported type')
            return False
        else:
            if self.get_type() == D.get_type()
            and self.get_arrows() == D.get_arrows()
            and self.get_coefficient() == D.get_coefficient():
                return True
            else :
                return False
    def __add__(self,GD):
        LE = LinearCombinationOfTriGaussDiagrams(self)
```

```
            if GD.__class__.__name__ == "CoeffGaussDiagramTri"
            or GD.__class__.__name__ == "LinearCombinationOfTriGaussDiagrams":
                LE = LE + GD
                return LE
            elif GD == 0:
                return LE
            else:
                return 'ERROR not the right type for an addition'
        def __neg__(self):
            GD = deepcopy(self)
            GD.set_coefficient(-1*self.get_coefficient())
            return GD
        def __sub__(self,GD):
            return self + (-GD)
        def __str__(self):
            return str(self.get_name())
        def __repr__(self):
            return str(self)
        def __len__(self):
            return len(self.get_arrows())


    class LinearCombinationOfTriGaussDiagrams:
        "A linear combination of Gauss Diagrams of a triple point containing one arrow"
        def __init__(self,GD=0,printerrors = False,name = 'INIT',printname = False):
            if GD == 0:
                self.__expression = []
            elif GD.__class__.__name__ == "CoeffGaussDiagramTri":
                self.__expression = [GD]
            else:
                self.__expression = GD
            self.__correction = []
            self.__printerrors = printerrors
            self.__name = name
            self.__printname = printname
        def set_expression(self,expression):
            self.__expression = expression
        def set_correction(self,expression):
            self.__correction = expression
        def set_printerrors(self,printerr):
            self.__printerrors = printerr
        def set_name(self,name):
            self.__name = name
        def set_printname(self,printname):
```

```
        self.__printname = printname
    def get_expression(self):
        return self.__expression
    def get_correction(self):
        return self.__correction
    def get_printerrors(self):
        return self.__printerrors
    def get_name(self):
        return self.__name
    def get_printname(self):
        return self.__printname
    def get_infos(self,i=1):
        if self.get_printname() :
            print(self.get_name())
        #Get the infos of the i-th Gauss Diagram (i starting at 1)
        n = len(self)
        i = i - 1
        if i>= n :
            print('Error : there isn\'t a {}-th Gauss diagram'.format(i+1))
        else :
            self[i].get_infos()
        print('Correction',self.get_correction())
    def clean_diagrams(self):
        for GD in self.get_expression():
            if GD.get_coefficient() == 0:
                self.__expression.remove(GD)
    def plot(self,figurenum=0,expression_length=5):
        n = len(self)
        if int(len(self)/expression_length) == 0:
            heigh = 1
        else:
            heigh = int(len(self)/expression_length)
        if n > 0 :
            if not self.get_printname():
                fig = plt.figure(figurenum,figsize=(expression_length*6, 5*heigh))
                i = 0
                j = 0
                for i in range(n) :
                    iline = int(i/expression_length)*(-2.5)
                    self[i].plot((j,iline),False)
                    j = (j+4)%(expression_length*4)
                if self.get_correction() != []:
                    j = 1
                    for i in range(len(self.get_correction())):
```

```
                    cor = self.get_correction()[i]
                    plt.text(4*n+j,-0.1,r'$+$',color='black',size='xx-large')
                    plotcor = r'$ \frac{1}{2}(w(%s)-1)$'%cor
                    plt.text(4*n+j+2,0-0.1,plotcor,color='black',
                             size='xx-large',horizontalalignment='center')
                    j = j + 4
            plt.xlim(0,0.1+4*expression_length)
            plt.ylim(int(len(self)/expression_length)*(-2.5)-1.2,1.2)
            ax = plt.axis('off')
        else:
            fig = plt.figure(figurenum,figsize=(expression_length*6, 5*heigh))
            j = 0
            x = - 0.2
            y = - 0.1
            #plt.title(self.get_name())
            plt.text(x,y,self.get_name(),color='black',size='xx-large')
            plt.text(x+1,y,'=',color='black',size='xx-large')
            for i in range(n) :
                iline = int(i/expression_length)*(-2.5)
                self[i].plot((j+1,iline),False)
                j = (j+4)%(expression_length*4)
            if self.get_correction() != []:
                j = 2
                for i in range(len(self.get_correction())):
                    cor = self.get_correction()[i]
                    plt.text(4*n+j,-0.1,r'$+$',color='black',size='xx-large')
                    plotcor = r'$ \frac{1}{2}(w(%s)-1)$'%cor
                    plt.text(4*n+j+2,0-0.1,plotcor,color='black',
                             size='xx-large',horizontalalignment='center')
                    j = j + 4
            plt.xlim(0,1.1+4*expression_length)
            plt.ylim(int(len(self)/expression_length)*
                     (-2.5)-1.2,int(len(self)/expression_length)+1.2)
            ax = plt.axis('off')
    else :
        if self.get_printerrors() :
            print('Combination is empty' )
def append(self,GD):
    self.__expression.append(GD)
def __len__(self):
    return len(self.get_expression())
def __getitem__(self, key):
    return self.get_expression()[key]
def __setitem__(self, key, value):
```

```
        self.__expression[key] = value
    def __delitem__(self, key):
        self.__expression.pop(key)
    def __missing__(self, key):
        print('%i is not a valid index'% key)
    def __iter__(self):
        return iter(self.get_expression())
    def __reversed__(self):
        return reversed(self.get_expression())
    def __contains__(self, item):
        ans = False
        for GD in self:
            if GD == item:
                ans = True
        return ans
    def __add__(self,GD):
        LE = deepcopy(self)
        if GD.__class__.__name__ == "LinearCombinationOfTriGaussDiagrams":
            for g in GD:
                LE = LE + g
            return LE

        elif GD.__class__.__name__ == "CoeffGaussDiagramTri"
        and GD.get_coefficient() != 0:
            isalready_in = False
            for i in range(len(self.get_expression())):
                g = self.get_expression()[i]
                if g.get_type() == GD.get_type()
                and g.get_arrows() == GD.get_arrows()
                and g.get_trigausscode() == GD.get_trigausscode()
                and g.get_homotopy() == GD.get_homotopy():
                    isalready_in = True
                    k = i
                    break
            if isalready_in :
                coeff1 = self.__expression[k].get_coefficient()
                coeff2 = GD.get_coefficient()
                expression = deepcopy(self.get_expression())
                expression[k].set_coefficient(coeff1+coeff2)
                LE = LinearCombinationOfTriGaussDiagrams(expression)
            else:
                LE.append(GD)
                LE.set_name('%s + %s'%(LE,GD))
            return LE
```

```
            elif GD == 0:
                return LE
            elif GD == LinearCombinationOfTriGaussDiagrams():
                return LE
            elif type(GD) == str:
                LE.__correction.append(GD)
                return LE
            else :
                print('Error : wrong format for',GD)
                return 'ERROR'
        def __neg__(self):
            LE = deepcopy(self)
            for GD in LE:
                GD = -GD
            return LE
        def __sub__(self,GD):
            return self + (-GD)
        def __str__(self):
            pr = ''
            for GD in self:
                pr = pr+GD.get_name()
            return pr
        def __repr__(self):
            return str(self)
```

```
In [ ]: def generate_reduc_gaussdiagram(arrows,trigausscode):
            '''A function that generates
            a reducted Gauss Code that contains only listed arrows'''
            #Generating a reduct GaussDiagram
            copytrigausscode = deepcopy(trigausscode)
            allowedset = []
            for a in arrows:
                namea = a.get_name()
                if type(namea) == int:
                    allowedset.append(namea)
                    allowedset.append(-namea)
                else:
                    fs,hs = a.get_coordinates()
                    allowedset.append(copytrigausscode[fs])
                    allowedset.append(copytrigausscode[hs])
            reductrigausscode = []
            for c in copytrigausscode:
                if c in allowedset:
                    reductrigausscode.append(c)
```

```
    return reductrigausscode

def arrows_in_circle(arrows,trigausscode):
    '''A function that detects if
    a set of TriGaussArrow are in a circle configuration'''
    ans = True
    #Generating a reduct GaussDiagram
    reductrigausscode = generate_reduc_gaussdiagram(arrows,trigausscode)
    #For each arrow, if we start from i or -i,
    #the next one should be -i or i except for singular arrows
    for a in arrows:
        namea = a.get_name()
        fc,hc = a.get_coordinates()
        if type(namea) == int:
            if fc < hc:
                firstend = namea
            else:
                firstend = -namea
            for i in range(len(reductrigausscode)):
                if reductrigausscode[i] == firstend:
                    break
            if i+1 == len(reductrigausscode):
                endafteri = reductrigausscode[0]
            else:
                endafteri = reductrigausscode[i+1]
            endbeforei = reductrigausscode[i-1]
            if endafteri != -firstend and endbeforei != -firstend:
                ans = False
        else:
            for i in range(len(reductrigausscode)):
                if type(reductrigausscode[i]) == str:
                    break
            if i+1 == len(reductrigausscode):
                endafteri = reductrigausscode[0]
            else:
                endafteri = reductrigausscode[i+1]
            endbeforei = reductrigausscode[i-1]
            if type(endafteri) != str and type(endbeforei) != str :
                ans = False
    return ans

def dist(a,b,total=12):
    distance = min(abs((b-a)%total),abs((a-b)%total))
    return distance
```

```
def shortest_distance(a,b,trigausscode):
    '''A function that returns the shortest way
    from a to b (or from b to a) in a TriGaussCode'''
    #Length from a to b
    for i in range(len(trigausscode)):
        if trigausscode[i] == a:
            break
    lenab = 0
    while trigausscode[i] != b:
        i = i+1
        if i == len(trigausscode):
            i = 0
        lenab = lenab + 1
    #Length from b to a
    for i in range(len(trigausscode)):
        if trigausscode[i] == b:
            break
    lenba = 0
    while trigausscode[i] != a:
        i = i+1
        if i == len(trigausscode):
            i = 0
        lenba = lenba + 1
    #Computing shortest way
    if lenab < lenba :
        return a,b
    else:
        return b,a

def border_homotopy_arrow(arrow,reductrigausscode):
    '''A function that returns the border homotopy class of
    an arrow in a circle configuration'''
    h = arrow.get_homotopy()
    #Looking for the shortest way:
    foot = arrow.get_foot()
    head = arrow.get_head()
    if foot == 'INIT' or head == 'INIT':
        print('ERROR : foot or head is not defined for the arrow {}'.format(arrow.get
        return 'Error'
    else:
        startway,endway = shortest_distance(foot,head,reductrigausscode)
        if startway == foot:
            return Nbraid - h
```

```
            else:
                return h

    def arrows_in_circle(arrows,trigausscode,printerr=False):
        '''A function that detects if
        a set of TriGaussArrow are in a circle configuration'''
        ans = True
        #Generating a reduct GaussDiagram
        reductrigausscode = generate_reduc_gaussdiagram(arrows,trigausscode)
        #For each arrow, if we start from i or -i,
        #the next one should be -i or i except for singular arrows
        for a in arrows:
            namea = a.get_name()
            fc,hc = a.get_coordinates()
            #Regular arrows
            if type(namea) == int:
                if fc < hc:
                    firstend = namea
                else:
                    firstend = -namea
                for i in range(len(reductrigausscode)):
                    if reductrigausscode[i] == firstend:
                        break
                if i+1 == len(reductrigausscode):
                    endafteri = reductrigausscode[0]
                else:
                    endafteri = reductrigausscode[i+1]
                endbeforei = reductrigausscode[i-1]
                if endafteri != -firstend and endbeforei != -firstend:
                    ans = False
            #Singular arrows
            else:
                for i in range(len(reductrigausscode)):
                    if type(reductrigausscode[i]) == str:
                        break
                if i+1 == len(reductrigausscode):
                    endafteri = reductrigausscode[0]
                else:
                    endafteri = reductrigausscode[i+1]
                endbeforei = reductrigausscode[i-1]
                if type(endafteri) != str and type(endbeforei) != str :
                    ans = False
        return ans
```

```
def circle_homotopy(arrows,trigausscode):
    '''A function that returns the homotopy of
    the inner loop associated to a set of arrows'''
    reductrigausscode = generate_reduc_gaussdiagram(arrows,trigausscode)
    inner_homotopy = Nbraid
    for arrow in arrows:
        inner_homotopy = inner_homotopy -
        border_homotopy_arrow(arrow,reductrigausscode)
    return inner_homotopy

def test_n_arrows(arrows,trigausscode,printerr=False):
    '''A function that tests if
    a configuration of arrows is allowed in a trigausscode'''
    ans = True
    #Testing if the n arrows are in circle
    if arrows_in_circle(arrows,trigausscode,printerr)
    and arrows != []:
        #Testing that the inner loop homotopy
        if circle_homotopy(arrows,trigausscode) not in range(1,Nbraid):
            ans = False
            if printerr:
                print('      The arrows {} are in circle but inner
                      ' loop homotopy is wrong : {}'.format(arrows,
                                circle_homotopy(arrows,trigausscode)))
    #Testing all n-1 set of arrows
    if printerr:
        print('      Test of subset of n-1 arrows of {}'.format(arrows))
    for a in arrows:
        #Checking a subset without a
        subsetarrows = deepcopy(arrows)
        subsetarrows.remove(a)
        if test_n_arrows(subsetarrows,trigausscode,printerr) == False:
            ans = False
            if printerr:
                print('         The subset of arrows {} is
                      ' not allowed'.format(subsetarrows))
    return ans

def is_diagram_allowed(D,printerr=False):
    '''A function that computes if a CoeffGaussDiagramTri is allowed'''
    trigausscode = D.get_trigausscode()
    d = D.get_d()
    hm = D.get_hm()
    ml = D.get_ml()
```

```
    arrows = D.get_arrows()
    arrowsd = deepcopy(arrows)
    arrowsd.append(d)
    arrowshm = deepcopy(arrows)
    arrowshm.append(hm)
    arrowsml = deepcopy(arrows)
    arrowsml.append(ml)
    if printerr:
        print('Test of diagram',D)
        print('   Test of n arrows with d : ',
              test_n_arrows(arrowsd,trigausscode))
        boold = test_n_arrows(arrowsd,trigausscode,printerr)
        print('   Test of n arrows with hm : ',
              test_n_arrows(arrowshm,trigausscode))
        boolhm = test_n_arrows(arrowshm,trigausscode,printerr)
        print('   Test of n arrows with ml : ',
              test_n_arrows(arrowsml,trigausscode,printerr))
    return (D.get_coefficient() !=0
            and test_n_arrows(arrowsd,trigausscode)
            and test_n_arrows(arrowshm,trigausscode)
            and test_n_arrows(arrowsml,trigausscode))


def all_insert_possible(e,trigausscode):
    '''A function that lists
    all the possible insertions of the element e in a trigausscode'''
    ListPoss = []
    for i in range(1,len(trigausscode)+1):
        newtrigausscode = deepcopy(trigausscode)
        newtrigausscode.insert(i,e)
        ListPoss.append(newtrigausscode)
    return ListPoss

def list_all_possible_trigausscode(nbarrows):
    '''A function that returns
    a list of all possible trigausscodes with nbarrows arrows'''
    InitList = [['A','B','C']]
    for i in range(1,nbarrows+1):
        ListTemp = []
        #Generating all the gausscode for the foot i
        for trigausscode in InitList:
            ListTemp.extend(all_insert_possible(i,trigausscode))
        #Generating all the gausscode for the head -i
        InitList = []
```

```
        for trigausscode in ListTemp:
            InitList.extend(all_insert_possible(-i,trigausscode))
    return InitList

def list_all_possible_CoeffGaussDiagramTri(nbarrows,
                                dtypelist=[1,1,'-'],maxi = 50000):
    '''A function that lists all possible CoeffGaussDiagramTri'''
    ListAll = []
    ListAllPossibleTGC = list_all_possible_trigausscode(nbarrows)
    #Generating for each type
    for dtype in ListType:
        if dtype != dtypelist:
            continue
        #Generating the Homotopy vector for each type of arrows
        pchoice = range(1,Nbraid)
        if nbarrows == 1 :
            choice = []
            for c in pchoice:
                choice.append([c])
            ListHom =  choice
        else:
            homotopy_choice = range(1,Nbraid)
            choice = []
            for c in homotopy_choice:
                choice.append([c])
            n = len(choice)
            L_temp = choice
            for i in range(nbarrows-1):
                ListHom = []
                for k in range(len(L_temp)):
                    l_k = L_temp[k]
                    for c in choice:
                        temp = deepcopy(l_k)
                        temp.extend(c)
                        ListHom.append(temp)
                L_temp = deepcopy(ListHom)
        #Removing "same" homotopy vector
        Notallowed = []
        for Hom in ListHom:
            for i in range(len(Hom)-1):
                if Hom[i] > Hom[i+1]:
                    Notallowed.append(Hom)
        for Hom in Notallowed:
            if Hom in ListHom:
```

```
                ListHom.remove(Hom)
        #For each homotopy vector, we list all possible trigausscodes
        for Hom in ListHom:
            for TGC in ListAllPossibleTGC:
                D = CoeffGaussDiagramTri(dtype,TGC,Hom,1)
                #Appending only allowed diagrams
                if is_diagram_allowed(D):
                    ListAll.append(D)
                if len(ListAll) >= maxi:
                    break
            if len(ListAll) >= maxi:
                break
    return ListAll

def list_all_possible_homogene_CoeffGaussDiagramTri(nbarrows,
                                dtypelist=[1,1,'-'],maxi = 50000):
    '''A function that lists all possible CoeffGaussDiagramTri'''
    ListAll = []
    ListAllPossibleTGC = list_all_possible_trigausscode(nbarrows)
    #Generating for each type
    for dtype in ListType:
        if dtype != dtypelist:
            continue
        #Generating the Homotopy vector for each type of arrows
        pchoice = range(1,Nbraid)
        if nbarrows == 1 :
            choice = []
            for c in pchoice:
                choice.append([c])
            ListHom =  choice
        else:
            homotopy_choice = range(1,Nbraid)
            choice = []
            for c in homotopy_choice:
                choice.append([c])
            n = len(choice)
            L_temp = choice
            for i in range(nbarrows-1):
                ListHom = []
                for k in range(len(L_temp)):
                    l_k = L_temp[k]
                    for c in choice:
                        temp = deepcopy(l_k)
                        temp.extend(c)
```

```
                        ListHom.append(temp)
                L_temp = deepcopy(ListHom)
        #Removing "same" homotopy vector
        Notallowed = []
        for Hom in ListHom:
            for i in range(len(Hom)-1):
                if Hom[i] >= Hom[i+1]:
                    Notallowed.append(Hom)
        for Hom in Notallowed:
            if Hom in ListHom:
                ListHom.remove(Hom)
        #For each homotopy vector, we list all possible trigausscodes
        for Hom in ListHom:
            for TGC in ListAllPossibleTGC:
                D = CoeffGaussDiagramTri(dtype,TGC,Hom,1)
                #Appending only allowed diagrams
                if is_diagram_allowed(D):
                    ListAll.append(D)
                if len(ListAll) >= maxi:
                    break
            if len(ListAll) >= maxi:
                break
    return ListAll

def list_all_allowed_homogene_CoeffGaussDiagramTri(nbarrows,
                                        dtype=[1,1,'-'],maxi=50000):
    '''A function that lists all allowed homogene CoeffGaussDiagramTri'''
    ListAll = list_all_possible_CoeffGaussDiagramTri(nbarrows,
                                                     dtype,maxi)
    #Removing all the non homogene CoeffGaussDiagramTri
    nonAllowedList = []
    for D in ListAll:
        Hom = D.get_homotopy()
        #Checking if two arrows have same homotopy
        deleteHom = False
        for i in range(len(Hom)):
            for j in range(i+1,len(Hom)):
                if Hom[i] == Hom[j]:
                    deleteHom = True
        if deleteHom == True or is_diagram_allowed(D) != True:
            nonAllowedList.append(D)
    #Removing the non allowed SignedGaussDiagramTri
    for D in nonAllowedList:
        ListAll.remove(D)
```

```
        return ListAll

    def list_all_allowed_homogene_CoeffGaussDiagramTri(nbarrows,
                                                       dtype=[1,1,'-'],maxi=50000):
        '''A function that lists all allowed homogene CoeffGaussDiagramTri'''
        ListAll = list_all_possible_homogene_CoeffGaussDiagramTri(nbarrows,
                                                                  dtype,maxi)
        #Removing all the non homogene CoeffGaussDiagramTri
        nonAllowedList = []
        for D in ListAll:
            Hom = D.get_homotopy()
            #Checking if two arrows have same homotopy
            deleteHom = False
            for i in range(len(Hom)):
                for j in range(i+1,len(Hom)):
                    if Hom[i] == Hom[j]:
                        deleteHom = True
            if deleteHom == True or is_diagram_allowed(D) != True:
                nonAllowedList.append(D)
        #Removing the non allowed SignedGaussDiagramTri
        for D in nonAllowedList:
            ListAll.remove(D)
        return ListAll
```

In [ ]:
```
def distance_from_to(a,b,trigausscode):
    '''A function that computes the distance from a to b in a trigausscode'''
    #Looking for a and b
    for i in range(len(trigausscode)):
        if trigausscode[i] == a:
            pa = i
        if trigausscode[i] == b:
            pb = i
    total = len(trigausscode)
    return (pb-pa)%total

def switch_ends(a,b,trigausscode):
    '''A function that switches a and b in a trigausscode'''
    #Finding a and b
    for i in range(len(trigausscode)):
        if trigausscode[i] == a:
            pa = i
        if trigausscode[i] == b:
            pb = i
    newtri = deepcopy(trigausscode)
```

```
    #Switching regular case
    if pa != len(trigausscode)-1 and pb != len(trigausscode)-1
        and pa != 0 and pb != 0:
        newtri[pa],newtri[pb] = newtri[pb],newtri[pa]
    #a is at the end
    elif pa == len(trigausscode)-1:
        newtri.remove(a)
        newtri.insert(1,a)
    #b is at the end
    elif pb == len(trigausscode)-1:
        newtri.remove(b)
        newtri.insert(1,b)
    #a is in first place
    elif pa == 0:
        newtri.remove(b)
        newtri.append(b)
    elif pb == 0:
        newtri.remove(a)
        newtri.append(a)
    return newtri

def positive_slide_move_head_is_possible(D,arrow,printerr=False):
    '''A function that detects if
    an arrow can perform a positive slide move for its head'''
    trigausscode = D.get_trigausscode()
    namea = arrow.get_name()
    #Checking if -namea is close to 'A', 'B' or 'C'
    if distance_from_to(-namea,'A',trigausscode) == 1
    or distance_from_to(-namea,'B',trigausscode) == 1
    or distance_from_to(-namea,'C',trigausscode) == 1:
        return True
    else:
        return False

def slide_move_positive_head(D1,arrow,printerr=False):
    '''A function that performs a positive slide move for the head of an arrow'''
    #Input : D1 a SignedGaussDiagramWithOneArrow
    #Output : D2 a SignedGaussDiagramWithOneArrow obtained from performing
    #a slide move with the head of the arrow on D1 or 0 if not possible

    #Sliding if possible
    if positive_slide_move_head_is_possible(D1,arrow,printerr):
        trigausscode = deepcopy(D1.get_trigausscode())
        dtype = D1.get_type()
```

```
            homotopy = D1.get_homotopy()
            coeff = D1.get_coefficient()
            #Finding the singular point to switch
            fc,hc = arrow.get_coordinates()
            maxi = len(trigausscode)
            singpt = (hc+1)%maxi
            trigausscode[hc],trigausscode[singpt] = trigausscode[singpt],trigausscode[hc]
            while trigausscode[0] != 'A':
                trigausscode = trigausscode[1:] + trigausscode[:1]
            D2 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
        else:
            if printerr:
                print('Error : cannot perform a positive head slide move')
            D2 = CoeffGaussDiagramTri()
            D2.set_coefficient(0)
        return D2

    def negative_slide_move_head_is_possible(D,arrow,printerr=False):
        '''A function that detects if an arrow can perform a positive slide move for its
        trigausscode = D.get_trigausscode()
        namea = arrow.get_name()
        #Checking if -namea is close to 'A', 'B' or 'C'
        if distance_from_to('A',-namea,trigausscode) == 1
        or distance_from_to('B',-namea,trigausscode) == 1
        or distance_from_to('C',-namea,trigausscode) == 1:
            return True
        else:
            return False

    def slide_move_negative_head(D1,arrow,printerr=False):
        '''A function that performs a positive slide move for the head of an arrow'''
        #Input : D1 a SignedGaussDiagramWithOneArrow
        #Output : D2 a SignedGaussDiagramWithOneArrow obtained from performing
        #a slide move with the head of the arrow on D1 or 0 if not possible

        #Sliding if possible
        if negative_slide_move_head_is_possible(D1,arrow,printerr):
            trigausscode = deepcopy(D1.get_trigausscode())
            dtype = D1.get_type()
            homotopy = D1.get_homotopy()
            coeff = D1.get_coefficient()
            #Finding the singular point to switch
            fc,hc = arrow.get_coordinates()
            maxi = len(trigausscode)
```

```
            singpt = (hc-1)%maxi
            trigausscode[hc],trigausscode[singpt] = trigausscode[singpt],trigausscode[hc]
            while trigausscode[0] != 'A':
                trigausscode = trigausscode[1:] + trigausscode[:1]
            D2 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
        else:
            if printerr:
                print('Error : cannot perform a negative head slide move')
            D2 = CoeffGaussDiagramTri()
            D2.set_coefficient(0)
        return D2

    def positive_slide_move_foot_is_possible(D,arrow,printerr=False):
        '''A function that detects if an arrow can perform a positive slide move for its
        trigausscode = D.get_trigausscode()
        namea = arrow.get_name()
        #Checking if -namea is close to 'A', 'B' or 'C'
        if distance_from_to(namea,'A',trigausscode) == 1
        or distance_from_to(namea,'B',trigausscode) == 1
        or distance_from_to(namea,'C',trigausscode) == 1:
            return True
        else:
            return False

    def slide_move_positive_foot(D1,arrow,printerr=False):
        '''A function that performs a positive slide move for the head of an arrow'''
        #Input : D1 a SignedGaussDiagramWithOneArrow
        #Output : D2 a SignedGaussDiagramWithOneArrow obtained from performing
        #a slide move with the head of the arrow on D1 or 0 if not possible
        #Sliding if possible
        if positive_slide_move_foot_is_possible(D1,arrow,printerr):
            trigausscode = deepcopy(D1.get_trigausscode())
            dtype = D1.get_type()
            homotopy = D1.get_homotopy()
            coeff = D1.get_coefficient()
            #Finding the singular point to switch
            fc,hc = arrow.get_coordinates()
            maxi = len(trigausscode)
            singpt = (fc+1)%maxi
            trigausscode[fc],trigausscode[singpt] = trigausscode[singpt],trigausscode[fc]
            while trigausscode[0] != 'A':
                trigausscode = trigausscode[1:] + trigausscode[:1]
            D2 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
        else:
```

```
            if printerr:
                print('Error : cannot perform a positive foot slide move')
            D2 = CoeffGaussDiagramTri()
            D2.set_coefficient(0)
        return D2

    def negative_slide_move_foot_is_possible(D,arrow,printerr=False):
        '''A function that detects if
        an arrow can perform a positive slide move for its head'''
        trigausscode = D.get_trigausscode()
        namea = arrow.get_name()
        #Checking if -namea is close to 'A', 'B' or 'C'
        if distance_from_to('A',namea,trigausscode) == 1
        or distance_from_to('B',namea,trigausscode) == 1
        or distance_from_to('C',namea,trigausscode) == 1:
            return True
        else:
            return False

    def slide_move_negative_foot(D1,arrow,printerr=False):
        '''A function that performs a positive slide move for the head of an arrow'''
        #Input : D1 a CoeffGaussDiagramWithOneArrow
        #Output : D2 a CoeffGaussDiagramWithOneArrow obtained from performing
        #a slide move with the head of the arrow on D1 or 0 if not possible

        #Sliding if possible
        if negative_slide_move_foot_is_possible(D1,arrow,printerr):
            trigausscode = deepcopy(D1.get_trigausscode())
            dtype = D1.get_type()
            homotopy = D1.get_homotopy()
            coeff = D1.get_coefficient()
            #Finding the singular point to switch
            fc,hc = arrow.get_coordinates()
            maxi = len(trigausscode)
            singpt = (fc-1)%maxi
            trigausscode[fc],trigausscode[singpt] = trigausscode[singpt],trigausscode[fc]
            while trigausscode[0] != 'A':
                trigausscode = trigausscode[1:] + trigausscode[:1]
            D2 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
        else:
            if printerr:
                print('Error : cannot perform a negative foot slide move')
            D2 = CoeffGaussDiagramTri()
            D2.set_coefficient(0)
```

```
        return D2

    def exists_in(E,D):
        '''A function that detects if
        a CoeffGaussDiagramTri D is in a LinearCombinationOfTriGaussDiagrams E'''
        #E = LinearCombinationOfGaussDiagrams, D = SignedGaussDiagramWithOneArrow
        exists = False
        for GD in E:
            if GD == D :
                exists = True
        return exists

    def all_slide_move(D,printerr=False):
        '''A function that computes all slide moves from a diagram D'''
        E = LinearCombinationOfTriGaussDiagrams()
        if is_diagram_allowed(D):
            D1 = deepcopy(D)
            for arrow in D.get_arrows():
                D1 = deepcopy(D)
                if is_diagram_allowed(slide_move_positive_head(D1,arrow,printerr)) :
                    D1 = slide_move_positive_head(D1,arrow,printerr)
                    if not exists_in(E,D1):
                        E = E + D1
                D1 = deepcopy(D)
                if is_diagram_allowed(slide_move_positive_foot(D1,arrow,printerr)) :
                    D1 = slide_move_positive_foot(D1,arrow,printerr)
                    if not exists_in(E,D1):
                        E = E + D1
                D1 = deepcopy(D)
                if is_diagram_allowed(slide_move_negative_head(D1,arrow,printerr)) :
                    D1 = slide_move_negative_head(D1,arrow,printerr)
                    if not exists_in(E,D1):
                        E = E + D1
                D1 = deepcopy(D)
                if is_diagram_allowed(slide_move_negative_foot(D1,arrow,printerr)) :
                    D1 = slide_move_negative_foot(D1,arrow,printerr)
                    if not exists_in(E,D1):
                        E = E + D1
        else:
            if printerr :
                print('Not an allowed diagram')
        return E
```

Regular exchange move

```
In [ ]: def exchange_move_is_possible(D,singular_arrow,regular_arrow):
            '''A function that detects if an arrow can perform
            a positive slide move for its head'''
            length = len(D.get_trigausscode())
            fa,ha = regular_arrow.get_coordinates()
            fs,hs = singular_arrow.get_coordinates()
            homa = regular_arrow.get_homotopy()
            homs = singular_arrow.get_homotopy()
            if dist(fa,fs,length) == 1 and dist(ha,hs,length) == 1
            and homa == homs:
                #Almost every cases
                if (ha < hs and fa < fs) or (ha > hs and fa > fs):
                    return True
                #Particular Case - Positive exchange with A as an endpoint
                elif (ha == length-1 and hs == 0 and fa < fs)
                or (fa == length-1 and fs == 0 and ha < hs):
                    return True
                else:
                    return False
            else:
                return False

        def exchange_move(D,singular_arrow,regular_arrow,printerr=False):
            '''A function that operates an exchange_move between
            a singular_arrow and a regular_arrow in a CoeffTriGaussDiagram'''
            if exchange_move_is_possible(D,singular_arrow,regular_arrow):
                dtype = D.get_type()
                homotopy = D.get_homotopy()
                coeff = -D.get_coefficient()
                arrows = deepcopy(D.get_arrows())
                fa,ha = regular_arrow.get_coordinates()
                fs,hs = singular_arrow.get_coordinates()
                rarrow = deepcopy(regular_arrow)
                rname = rarrow.get_name()
                trigausscode = deepcopy(D.get_trigausscode())
                trigausscode[fa],trigausscode[fs] = trigausscode[fs],trigausscode[fa]
                trigausscode[ha],trigausscode[hs] = trigausscode[hs],trigausscode[ha]
                while trigausscode[0] != 'A':
                    trigausscode = trigausscode[1:] + trigausscode[:1]
                D1 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
                return D1
            else:
                if printerr:
                    print('Error : cannot perform an exchange move')
```

```
            return 0


    def all_exchange_move(D,printerr=False):
        '''A function that computes all exchange moves from a diagram D'''
        E = LinearCombinationOfTriGaussDiagrams()
        d = D.get_d()
        hm = D.get_hm()
        ml = D.get_ml()
        arrows = D.get_arrows()
        if is_diagram_allowed(D):
            D1 = deepcopy(D)
            for a in arrows:
                E = E + exchange_move(D,d,a)
                E = E + exchange_move(D,hm,a)
                E = E + exchange_move(D,ml,a)
        else:
            if printerr :
                print('Not an allowed diagram')
        return E
```

Refined exchange move

```
In [ ]: def exchange_move_is_possible(D,singular_arrow,regular_arrow):
            '''A function that detects if an arrow can perform
            a positive slide move for its head'''
            length = len(D.get_trigausscode())
            fa,ha = regular_arrow.get_coordinates()
            fs,hs = singular_arrow.get_coordinates()
            homa = regular_arrow.get_homotopy()
            homs = singular_arrow.get_homotopy()
            if dist(fa,fs,length) == 1 and dist(ha,hs,length) == 1
            and homa == homs:
                #Almost every cases
                if (ha < hs and fa < fs) or (ha > hs and fa > fs):
                    return True
                #Particular Case - Positive exchange with A as an endpoint
                elif (ha == length-1 and hs == 0 and fa < fs)
                or (fa == length-1 and fs == 0 and ha < hs):
                    return True
                else:
                    return False
            else:
                return False
```

```
        def exchange_move(D,singular_arrow,regular_arrow,printerr=False):
            '''A function that operates an exchange_move between
            a singular_arrow and a regular_arrow in a CoeffTriGaussDiagram'''
            if exchange_move_is_possible(D,singular_arrow,regular_arrow):
                return singular_arrow.get_name()
            else:
                if printerr:
                    print('Error : cannot perform an exchange move')
                return 0


        def all_exchange_move(D,printerr=False):
            '''A function that computes all exchange moves from a diagram D'''
            E = LinearCombinationOfTriGaussDiagrams()
            d = D.get_d()
            hm = D.get_hm()
            ml = D.get_ml()
            arrows = D.get_arrows()
            res = []
            if is_diagram_allowed(D):
                D1 = deepcopy(D)
                for a in arrows:
                    res.append(exchange_move(D,d,a))
                    res.append(exchange_move(D,hm,a))
                    res.append(exchange_move(D,ml,a))
            else:
                if printerr :
                    print('Not an allowed diagram')
            return res
```

In [ ]:
```
        def ends_are_switchable(end1,end2):
            '''A function that detects if end1 and end2 are switchable'''
            return abs(end1-end2) == 1

        def do_cross(arrow1,arrow2):
            '''A function that computes if two arrows cross'''
            (f1,h1) = arrow1.get_coordinates()
            (f2,h2) = arrow2.get_coordinates()
            a = min(f1,h1)
            b = max(f1,h1)
            c = min(f2,h2)
            d = max(f2,h2)
            if (a < c < d < b) or ( c < d < a < b) or (c < a < b < d)
```

```
    or (a < b < c < d):
        return False
    else :
        return True

def insert_arrow(D,fa,ha,homa):
    '''A function that insert an arrow in a CoeffTriGaussDiagram'''
    #Modifying the trigausscode
    trigausscode = deepcopy(D.get_trigausscode())
    namea = len(D.get_arrows())+1
    if ha < fa :
        trigausscode.insert(ha,-namea)
        trigausscode.insert(fa,namea)
    else:
        trigausscode.insert(fa,namea)
        trigausscode.insert(ha,-namea)
    #Rolling the trigausscode until 'A' is in position 0
    while trigausscode[0] != 'A':
        trigausscode = trigausscode[1:] + trigausscode[:1]
    homotopy = deepcopy(D.get_homotopy())
    homotopy.append(homa)
    dtype = D.get_type()
    coeff = D.get_coefficient()
    return CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)

def compute_homotopy_for_3rd_slide_move_arrow(D,arrow1,arrow2):
    '''A function that computes the homotopy class of the 3rd arrow
    in a slide move for two arrows not in the triangle'''
    trigausscode = D.get_trigausscode()
    hom1 = border_homotopy_arrow(arrow1,trigausscode)
    hom2 = border_homotopy_arrow(arrow2,trigausscode)
    return Nbraid - (hom1 + hom2)

def left_right_slide_move_for_two_arrows(D,arrow1,arrow2,end1,end2):
    '''A functiont that returns the left and the right arrow
    of a slide move for two arrows not in the triangle'''
    #Going to the next endpoint of the two arrows from the singular point
    singpt = max(end1,end2)
    trigausscode = D.get_trigausscode()
    name1 = arrow1.get_name()
    name2 = arrow2.get_name()
    singpt = (singpt+1)%len(trigausscode)
    found = False
    i = singpt
```

```
        while found == False:
        #for i in range(singpt,len(trigausscode)):
            if trigausscode[i] == -name1 or trigausscode[i] == name1:
                left = arrow1
                right = arrow2
                found = True
                break
            if trigausscode[i] == -name2 or trigausscode[i] == name2:
                left = arrow2
                right = arrow1
                found = True
                break
            i = (i+1)%len(trigausscode)
        return left,right

    def slide_move_for_two_arrows_is_doable_between(D,arrow1,arrow2,
                                                    end1,end2,printerr=False):
        '''A function that detects if a slide move is doable between
        two arrows not in the triangle'''
        #Checking for error
        f1,h1 = arrow1.get_coordinates()
        f2,h2 = arrow2.get_coordinates()
        if (end1 != f1 and end1 != h1) or (end2 != f2 and end2 != h2):
            if printerr:
                print('Error : the ends do not correspond to
                      ' the arrows :',arrow1.get_name(),arrow2.get_name())
            return False
        #If the ends are switchable, we compute the hypothetical RM3
        elif ends_are_switchable(end1,end2):
            #We compute the third arrow
            #Determining left and right
            leftarrow,rightarrow = left_right_slide_move_for_two_arrows(D,
                                                        arrow1,arrow2,end1,end2)
            if printerr:
                print('Test du triangle pour Flèche de gauche : {},
                      'Flèche de droite : {}'.format(leftarrow.get_name(),
                                                     rightarrow.get_name()))
            fl,hl = leftarrow.get_coordinates()
            fr,hr = rightarrow.get_coordinates()
            List_coords = [coord for coord in
                           (leftarrow.get_coordinates()+rightarrow.get_coordinates())
                           if (coord != end1 and coord != end2)]
            #Computing the coordinates of the third arrow
            #The head is the left arrow end
```

```
            if fl in List_coords:
                head = fl
            else:
                head = hl
            #The foot is the right arrow end
            if fr in List_coords:
                foot = fr
            else:
                foot = hr
            #Computing the coordinates depending on if the arrows cross
            #If the two arrows cross
            if do_cross(arrow1,arrow2):
                if head < foot :
                    f3,h3 = foot + 1, head + 1
                else:
                    f3,h3 = foot, head + 2
            else:
                if foot < head :
                    f3,h3 = foot + 1, head + 1
                else:
                    f3,h3 = foot + 2, head
            #If the two arrows crosses
#           if do_cross(arrow1,arrow2):
#               f3,h3 = foot+1,head+1
#           else:
#               if head == min(foot,head):
#                   foot = foot + 2
#               else:
#                   head = head + 2
#               f3,h3 = foot,head
            hom3 = compute_homotopy_for_3rd_slide_move_arrow(D,arrow1,arrow2)
            D2 = insert_arrow(D,f3,h3,hom3)
            if printerr:
                D2.plot()
                print('Plot de {} dans le cadre d\'un
                      'test de slide tri'.format(D2.get_trigausscode()))
            if is_diagram_allowed(D2,printerr):
                return True
            else:
                return False

    def all_end_slide_move(D,arrow1,arrow2,printerr=False):
        (f1,h1) = arrow1.get_coordinates()
        (f2,h2) = arrow2.get_coordinates()
```

```
        dtype = D.get_type()
        homotopy = D.get_homotopy()
        coeff = D.get_coefficient()
        arrows = deepcopy(D.get_arrows())
        res = LinearCombinationOfTriGaussDiagrams()
        if slide_move_for_two_arrows_is_doable_between(D,arrow1,arrow2,f1,f2,printerr):
            pt1 = f1
            pt2 = f2
            #sliding
            trigausscode = deepcopy(D.get_trigausscode())
            trigausscode[pt1],trigausscode[pt2] = trigausscode[pt2],trigausscode[pt1]
            D2 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
            res = res + D2
        elif slide_move_for_two_arrows_is_doable_between(D,arrow1,arrow2,f1,h2,printerr):
            pt1 = f1
            pt2 = h2
            #sliding
            trigausscode = deepcopy(D.get_trigausscode())
            trigausscode[pt1],trigausscode[pt2] = trigausscode[pt2],trigausscode[pt1]
            D2 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
            res = res + D2
        elif slide_move_for_two_arrows_is_doable_between(D,arrow1,arrow2,h1,f2,printerr):
            pt1 = h1
            pt2 = f2
            #sliding
            trigausscode = deepcopy(D.get_trigausscode())
            trigausscode[pt1],trigausscode[pt2] = trigausscode[pt2],trigausscode[pt1]
            D2 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
            res = res + D2
        elif slide_move_for_two_arrows_is_doable_between(D,arrow1,arrow2,h1,h2,printerr):
            pt1 = h1
            pt2 = h2
            #sliding
            trigausscode = deepcopy(D.get_trigausscode())
            trigausscode[pt1],trigausscode[pt2] = trigausscode[pt2],trigausscode[pt1]
            D2 = CoeffGaussDiagramTri(dtype,trigausscode,homotopy,coeff)
            res = res + D2
        else:
            if printerr:
                print('Error : cannot perform a tri slide move')
        return res

    def all_slide_move_for_two_arrows_not_in_triangle(D,printerr=False):
        '''A function that computes all slide moves for
```

```
        two arrows not in the triangle from a diagram D'''
        E = LinearCombinationOfTriGaussDiagrams()
        arrows = D.get_arrows()
        if is_diagram_allowed(D):
            D2 = deepcopy(D)
            for i in range(len(arrows)):
                for j in range(i+1,len(arrows)):
                    arrow1 = arrows[i]
                    arrow2 = arrows[j]
                    E = E + all_end_slide_move(D,arrow1,arrow2,printerr)
        else:
            if printerr :
                print('Not an allowed diagram')
        return E
```

In [ ]:
```
def exists_in(E,D):
    '''A function that detects if
    a CoeffGaussDiagramTri D is in a LinearCombinationOfTriGaussDiagrams E'''
    #E = LinearCombinationOfGaussDiagrams, D = SignedGaussDiagramWithOneArrow
    exists = False
    if type(D) == str:
        return False
    else:
        for GD in E:
            if GD == D :
                exists = True
        return exists

def generate_invariant(D,printerr=False):
    '''A function that generates
    the invariant that contains a CoeffGaussDiagramTri D'''
    #D  = SignedGaussDiagramWithOneArrow
    E = LinearCombinationOfTriGaussDiagrams()
    if is_diagram_allowed(D,printerr):
        E = E + D
        i = 0
        n = len(E)
        #i represents the diagram for which we compute all the possible moves
        while i<n :
            D = E[i]
            for Dj in all_slide_move(D,printerr):
                if not exists_in(E,Dj):
                    E = E + Dj
            for cor in all_exchange_move(D,printerr):
```

```
                        if not exists_in(E,cor):
                            E = E + cor
                    for Dj in all_slide_move_for_two_arrows_not_in_triangle(D,printerr):
                        if not exists_in(E,Dj):
                            E = E + Dj
                    i = i+1
                    n = len(E)
                return E

            else:
                if printerr :
                    print('Error : The diagram is not allowed.
                          'Will not compute invariant')
                return 'Error'
```

In [ ]:
```
def is_in_list(Liste,D):
    '''A function that detects if a CoeffGaussDiagramTri is in a List'''
    presence = False
    for G in Liste:
        if G.get_type() == D.get_type()
        and G.get_arrows() == D.get_arrows()
        and G.get_homotopy() == D.get_homotopy():
            presence = True
    return presence

def list_all_invariants(nbarrows,dtype=[1,1,'-'],
                        maxi=50000,printerr=False):
    '''A function that list all invariants'''
    T = []
    #On commence par générer tous les diagrammes possibles :
    L = list_all_allowed_homogene_CoeffGaussDiagramTri(nbarrows,
                                                       dtype,maxi)

    #Pour chaque diagramme dans la liste,on calcule son invariant
    while (len(L) > 0):
        D = L[0]
        E = generate_invariant(D,printerr)
        T.append(E)
        for G in E.get_expression():
            if is_in_list(L,G):
                H = deepcopy(G)
                H.set_coefficient(1)
                L.remove(H)
    return T
```

```
        def list_all_type_invariants(nbarrows,maxi=50000,printerr=False):
            '''A function that list all invariants of all types'''
            T = []
            for dtype in ListType:
                T = T + list_all_invariants(nbarrows,dtype,maxi,printerr)
            return T
```

Generating the refined invariants

```
In [ ]: T1 = list_all_type_invariants(1)
        # print('Nombre d\'invariants : ',len(T1))
        for i in range(len(T1)):
            name = '$W_{%d}^{1,aff}$'%i
            T1[i].set_name(name)
            T1[i].set_printname(True)
            T1[i].plot(i)
```

1.1.3 I.c. Functions to compute the invariants

```
In [ ]: def trigausscode_reconstruct_singular(arrows,size_tri,sign='+'):
            '''A function that computes
            a trigausscode from a set of arrows, the first arrows are d,hm,ml'''
            trigausscode =  l = [0] * (size_tri)
            for k in range(3,len(arrows)):
                a = arrows[k]
                if a.__class__.__name__ == "TriGaussArrow":
                    i = a.get_name()
                elif a.__class__.__name__ == "TorusGaussArrow":
                    i = a.get_number()
                else:
                    return 'ERROR : Wrong class for the arrows'
                fa,ha = a.get_coordinates()
                #print(i,fa,ha)
                trigausscode[fa] = i
                trigausscode[ha] = -i
            #Generating A,B,C
            fd,hd = arrows[0].get_coordinates()
            fhm,hhm = arrows[1].get_coordinates()
            right = fhm
            left = fd
            if sign == '-':
                right,left = left,right
            trigausscode[right] = 'A'
```

```
            trigausscode[hd] = 'B'
            trigausscode[left] = 'C'
            #Removing the remaining 0
            final_trigausscode = [el for el in trigausscode if el != 0]
            #Placing A first
            while final_trigausscode[0] != 'A':
                final_trigausscode = final_trigausscode[1:] +
                final_trigausscode[:1]
            return final_trigausscode

        def compute_temp_arrow(name,trigausscode):
            '''A function that computes
            the temporary coordinates for an arrow in a trigausscode'''
            for i in range(len(trigausscode)):
                if trigausscode[i] == name:
                    fa = i
                elif trigausscode[i] == -name:
                    ha = i
            return fa,ha
```

```
In [ ]: def filtered_list(arrows,model_diagram,size_model_tri,
                           diagram,size_tri,model_arrow,
                           dictionary,sign,debug=False):
            '''A function that returns the arrows of diagram
            that are like model_arrow in model_diagram using a dictionary'''
            if debug:
                print('   DEBUT FILTRE')
                print('   Flèche modèle testée',model_arrow.get_name())
            if len(model_diagram) <= 0:
                return arrows
            else:
                #Initialization
                result_list = []
                #Constructing the model_trigausscode
                model_trigausscode = trigausscode_reconstruct_singular(model_diagram+
                                                    [model_arrow],size_model_tri,sign)
                if debug:
                    print('   Construction du modèle',model_trigausscode)
                model_f,model_h = compute_temp_arrow(model_arrow.get_name(),
                                                     model_trigausscode)
                model_hom = model_arrow.get_homotopy()
                model_f_after = (model_f + 1)%len(model_trigausscode)
                model_f_before = (model_f - 1)%len(model_trigausscode)
                model_h_after = (model_h + 1)%len(model_trigausscode)
```

```
model_h_before = (model_h - 1)%len(model_trigausscode)
#Testing each arrow
if debug:
    print('   Test des flèches par rapport à ',
          model_arrow.get_name())
for arrow in arrows:
    if debug:
        print('        Test de la flèche',arrow.get_number())
    #Constructing a temporary dictionary
    temp_dictionary = deepcopy(dictionary)
    temp_dictionary[model_arrow.get_name()] = arrow.get_number()
    temp_dictionary[-model_arrow.get_name()] = -arrow.get_number()
    if debug:
        print('          ',temp_dictionary)
    #Constructing the trigausscode to be tested
    tested_trigausscode = trigausscode_reconstruct_singular(diagram+
                                                [arrow],size_tri,sign)
    if debug:
        print('          Construction du diagramme à tester',
              tested_trigausscode)
    fa,ha = compute_temp_arrow(arrow.get_number(),tested_trigausscode)
    #Testing position of the foot
    f_after = (fa+1)%len(tested_trigausscode)
    f_before = (fa-1)%len(tested_trigausscode)
    test_arrow = tested_trigausscode[f_before] ==
        temp_dictionary[model_trigausscode[model_f_before]]
    test_arrow = test_arrow
    and (tested_trigausscode[f_after] ==
                        temp_dictionary[model_trigausscode[model_f_after]])
    #Testing position of the head
    h_after = (ha+1)%len(tested_trigausscode)
    h_before = (ha-1)%len(tested_trigausscode)
    test_arrow = test_arrow
    and (tested_trigausscode[h_before] ==
         temp_dictionary[model_trigausscode[model_h_before]])
    test_arrow = test_arrow
    and (tested_trigausscode[h_after] ==
         temp_dictionary[model_trigausscode[model_h_after]])
    #Testing the homotopy
    homa = arrow.get_homotopy()
    test_arrow = test_arrow and (homa == model_hom)
    if test_arrow:
        result_list.append(arrow)
    if debug:
```

```
                    print('          ',arrow.get_number(),test_arrow)
            if debug:
                print('   FIN FILTRE')
            return result_list
```

```
In [ ]: def test_type(D,GD):
            '''A function that returns the condition : the BraidData is of type GD'''
            if GD != 0:
                #Initialization
                D1 = deepcopy(D)
                #Getting the number of the three singular arrows
                clas = D1.get_Class()
                a,b,c = D1.get_RM3_a(),D1.get_RM3_b(),D1.get_RM3_c()
                d,hm,ml = name_arrows(a,b,c)
                #Testing if we are in the appropriate RM3 type
                homd = GD.get_d().get_homotopy()
                homhm = GD.get_hm().get_homotopy()
                homml = GD.get_ml().get_homotopy()
                return homd == d.get_homotopy()
            and homhm == hm.get_homotopy() and homml == ml.get_homotopy()
            else:
                return False

        def contribution(I0,G,debug=False):
            '''A function that returns the contribution of
            a BraidData G with respect to a CoefficientGaussDiagramTri I0'''
            #Testing the type
            if test_type(G,I0):
                #Creating a dictionary that will translate
                #the TriGaussArrows of I0 into TorusGaussArrows for G
                current_dictionary = {}
                arrow_d,arrow_hm,arrow_ml = name_arrows(G.get_RM3_a(),
                                                        G.get_RM3_b(),G.get_RM3_c())
                d = arrow_d.get_number()
                hm = arrow_hm.get_number()
                ml = arrow_ml.get_number()
                dtype = I0.get_type()
                sign = dtype[2]
                A = [-ml,hm]
                B = [-hm,-d]
                C = [d,ml]
                if sign == '-':
                    B,C = C,B
                current_dictionary["A"] = "A"
```

```
current_dictionary["B"] = "B"
current_dictionary["C"] = "C"
#Initialisation of the TriGaussDiagram to test
current_model_diagram = [I0.get_d(),I0.get_hm(),I0.get_ml()]
size_model_tri = len(I0.get_trigausscode())
#Initialisation of the Arrows that are tested in G
current_diagram = [arrow_d,arrow_hm,arrow_ml]
size_tri = len(G.get_TorusGaussCode().get_GaussDiagram())
#Initialisation of the index of the model arrow in I0
current_arrow_index = 0
#Getting the number of arrows in I0
n = len(I0)
#Initialisation of the list of corresponding set of arrows
corresponding_arrows_set_list = []
#Defining a function that will recursively compute the arrows that correspond
def recursive_test(current_model_diagram,current_diagram,
                   current_arrow_index,current_dictionary):
    if current_arrow_index == n:
        corresponding_arrows_set_list.append(current_diagram)
    else:
        model_arrow = I0.get_arrows()[current_arrow_index]
        if debug:
            print('Test pour ',model_arrow.get_name())
        remaining_arrows = [arrow for arrow in G.get_TorusGaussCode().get_Arr
                            if arrow not in current_diagram]
        if debug:
            print('Remaining arrows',[arrow.get_number()
                                      for arrow in remaining_arrows])
        #For each arrow that is like model_arrow,
        #we test the other arrows of model_diagram
        for arrow in filtered_list(remaining_arrows,current_model_diagram,
                                   size_model_tri,current_diagram,size_tri,mo
                                   current_dictionary,sign,debug):
            #Updating to test the other arrows of I0
            new_model_diagram = current_model_diagram + [model_arrow]
            new_diagram = current_diagram + [arrow]
            new_arrow_index = current_arrow_index + 1
            new_dictionary = deepcopy(current_dictionary)
            new_dictionary[model_arrow.get_name()] = arrow.get_number()
            new_dictionary[-model_arrow.get_name()] = -arrow.get_number()
            recursive_test(new_model_diagram,new_diagram,
                           new_arrow_index,new_dictionary)
recursive_test(current_model_diagram,current_diagram,current_arrow_index,
               current_dictionary)
```

```
            #Now corresponding_arrows_set_list contains
            #all the set of arrows that are like the model_diagram I0
            res = 0
            coeff = I0.get_coefficient()
            if debug:
                print('Coefficient',coeff)
            if debug and len(corresponding_arrows_set_list) > 0 :
                print('Set of corresponding arrows : ')
                for index in range(len(corresponding_arrows_set_list)):
                    print('   Set ',index,[arrow.get_number()
                                           for arrow in corresponding_arrows_set_list[ind
            #We get the contribution of each set
            for arrows_set in corresponding_arrows_set_list:
                temp_res = 1
                for i in range(3,len(arrows_set)):
                    temp_res = temp_res*arrows_set[i].get_writhe()
                res = res + temp_res
            return res
        else:
            return 0

    def invariant_compute(D,List_Expression,debug=False):
        '''A function that computes the invariants
        given by List_Expression for D'''
        D1 = deepcopy(D)
        List_Expressions = deepcopy(List_Expression)
        res = []
        for LC in List_Expressions:
            I = 0
            if LC != 0 :
                #Part with Gauss Diagram
                for GD in LC.get_expression():
                    coeff = GD.get_coefficient()
                    cont = contribution(GD,D1)
                    if debug and cont != 0:
                        contribution(GD,D1,debug)
                        print('Contribution : ',coeff*cont)
                        plt.figure()
                        GD.plot()
                    I = I + coeff*cont
                #Part with correction term
                if LC.get_correction() != []:
                    a,b,c = D1.get_RM3_a(),D1.get_RM3_b(),D1.get_RM3_c()
                    d,hm,ml = name_arrows(a,b,c)
```

```
                        wd = d.get_writhe()
                        whm = hm.get_writhe()
                        wml = ml.get_writhe()
                        for cor in LC.get_correction():
                            if cor == 'd':
                                I = I + (1/2)*(wd-1)
                            if cor == 'hm':
                                I = I + (1/2)*(whm-1)
                            if cor == 'ml':
                                I = I + (1/2)*(wml-1)
                    res.append(I)
                return array(res)
```

```
In [ ]: R.<x0,x1,x2,x3,x4,x5> = LaurentPolynomialRing(QQ)
        x = [x0,x1,x2,x3,x4,x5]
```

T1raf is the variable that have all the weights

```
In [ ]: T1raf = [[T1[i]] for i in range(len(T1))]
```

T1mult is the variable that have the weights for the multivariable invariants

```
In [ ]: T1mult = [[T1[index] for index in range(j,6+j)] for j in range(6)]
```

Pushing $Delta^2$ in order to apply *rot*

```
In [ ]: def push_delta(D,List_invariants,i,debug=False):
            '''A function that pushes Delta through a generator
            and computes the contribution with
            respect to a list of invariants'''
            #D is a BraidData, i is the position of the generator
            #Getting a copy of D
            D1 = deepcopy(D)
            diag = []
            singular = []
            cont = [0 for index in range(len(List_invariants))]
            GD = [List_invariants[index][0][0] for index
                  in range(len(List_invariants))]
            #Getting the braid element in position i
            ei = D.get_BraidClosure()[i]
            #Pushing through 1 : 1 123121
            if ei == 1 :
                #Commutation move : 1123121 = 1121321
                D1.CommutationMove(i+3)
                diag.append(deepcopy(D1))
```

```
#Reidemeister move III : 1121321 --> 1212321
d = D1.ReidemeisterMoveIII(i+1,i+2,i+3)
I = [0 for index in range(len(List_invariants))]
for index in range(len(I)):
    I[index] = invariant_compute(d,List_invariants[index])
s = d.get_sign()
for index in range(len(List_invariants)):
    if test_type(d,GD[index]):
        tempcont = 1
        for jindex in range(len(List_invariants[index])):
            tempcont = tempcont*x[jindex]**I[index][jindex]
        cont[index] = cont[index] + s*tempcont
        if debug:
            print(d.get_Class())
            plt.figure()
            d.plot_diagram()
            print(s*tempcont)
            print(invariant_compute(d,
                                    List_invariants[index],debug))
diag.append(d)
diag.append(deepcopy(D1))
singular.append(d)
#Reidemeister move III : 1212321 --> 1213231
d = D1.ReidemeisterMoveIII(i+3,i+4,i+5)
I = [0 for index in range(len(List_invariants))]
for index in range(len(I)):
    I[index] = invariant_compute(d,List_invariants[index])
s = d.get_sign()
for index in range(len(List_invariants)):
    if test_type(d,GD[index]):
        tempcont = 1
        for jindex in range(len(List_invariants[index])):
            tempcont = tempcont*x[jindex]**I[index][jindex]
        cont[index] = cont[index] + s*tempcont
        if debug:
            print(d.get_Class())
            plt.figure()
            d.plot_diagram()
            print(s*tempcont)
            print(invariant_compute(d,
                                    List_invariants[index],debug))
diag.append(d)
diag.append(deepcopy(D1))
singular.append(d)
```

```
        #Commutation move : 1213231 = 1231231
        D1.CommutationMove(i+2)
        diag.append(deepcopy(D1))
        #Commutation move : 1231231 = 1231213 ====> 123121 3
        D1.CommutationMove(i+5)
        diag.append(deepcopy(D1))

    #Pushing through -1 : -1 123121
    elif ei == -1 :
        #Reidemeister move II : -1123121 --> 23121
        D1.ReidemeisterMoveII_delete(i)
        diag.append(deepcopy(D1))
        #Reidemeister move II : 23121 --> 1-123121
        D1.ReidemeisterMoveII_create(1,i)
        diag.append(deepcopy(D1))
        #Commutation move : 1-123121 = 1-121321
        D1.CommutationMove(i+3)
        diag.append(deepcopy(D1))
        #Reidemeister move III : 1-121321 -->121-2321
        d = D1.ReidemeisterMoveIII(i+1,i+2,i+3)
        I = [0 for index in range(len(List_invariants))]
        for index in range(len(I)):
            I[index] = invariant_compute(d,List_invariants[index])
        s = d.get_sign()
        for index in range(len(List_invariants)):
            if test_type(d,GD[index]):
                tempcont = 1
                for jindex in range(len(List_invariants[index])):
                    tempcont = tempcont*x[jindex]**I[index][jindex]
                cont[index] = cont[index] + s*tempcont
                if debug:
                    print(d.get_Class())
                    plt.figure()
                    d.plot_diagram()
                    print(s*tempcont)
                    print(invariant_compute(d,
                                    List_invariants[index],debug))
        diag.append(d)
        diag.append(deepcopy(D1))
        singular.append(d)
        #Reidemeister move III : 121-2321 --> 12132-31
        d = D1.ReidemeisterMoveIII(i+3,i+4,i+5)
        I = [0 for index in range(len(List_invariants))]
        for index in range(len(I)):
```

```
            I[index] = invariant_compute(d,List_invariants[index])
        s = d.get_sign()
        for index in range(len(List_invariants)):
            if test_type(d,GD[index]):
                tempcont = 1
                for jindex in range(len(List_invariants[index])):
                    tempcont = tempcont*x[jindex]**I[index][jindex]
                cont[index] = cont[index] + s*tempcont
                if debug:
                    print(d.get_Class())
                    plt.figure()
                    d.plot_diagram()
                    print(s*tempcont)
                    print(invariant_compute(d,
                                    List_invariants[index],debug))
        diag.append(d)
        diag.append(deepcopy(D1))
        singular.append(d)
        #Commutation move : 12132-31 = 12312-31
        D1.CommutationMove(i+2)
        diag.append(deepcopy(D1))
        #Commutation move : 12312-31 = 123121-3 ====> 123121 -3
        D1.CommutationMove(i+5)
        diag.append(deepcopy(D1))

    #Pushing through 2 : 2 123121
    elif ei == 2 :
        #Reidemeister move III : 2123121 --> 1213121
        d = D1.ReidemeisterMoveIII(i,i+1,i+2)
        I = [0 for index in range(len(List_invariants))]
        for index in range(len(I)):
            I[index] = invariant_compute(d,List_invariants[index])
        s = d.get_sign()
        for index in range(len(List_invariants)):
            if test_type(d,GD[index]):
                tempcont = 1
                for jindex in range(len(List_invariants[index])):
                    tempcont = tempcont*x[jindex]**I[index][jindex]
                cont[index] = cont[index] + s*tempcont
                if debug:
                    print(d.get_Class())
                    plt.figure()
                    d.plot_diagram()
                    print(s*tempcont)
```

```
                print(invariant_compute(d,
                                        List_invariants[index],debug))
    diag.append(d)
    diag.append(deepcopy(D1))
    singular.append(d)
    #Commutation move : 1213121 = 1231121
    D1.CommutationMove(i+2)
    diag.append(deepcopy(D1))
    #Reidemeister move III : 1231121 --> 1231212 ====> 123121 2
    d = D1.ReidemeisterMoveIII(i+4,i+5,i+6)
    I = [0 for index in range(len(List_invariants))]
    for index in range(len(I)):
        I[index] = invariant_compute(d,List_invariants[index])
    s = d.get_sign()
    for index in range(len(List_invariants)):
        if test_type(d,GD[index]):
            tempcont = 1
            for jindex in range(len(List_invariants[index])):
                tempcont = tempcont*x[jindex]**I[index][jindex]
            cont[index] = cont[index] + s*tempcont
            if debug:
                print(d.get_Class())
                plt.figure()
                d.plot_diagram()
                print(s*tempcont)
                print(invariant_compute(d,
                                        List_invariants[index],debug))
    diag.append(d)
    diag.append(deepcopy(D1))
    singular.append(d)

#Pushing through -2 : -2 123121
elif ei == -2 :
    #Reidemeister move III : -2123121 --> 12-13121
    d = D1.ReidemeisterMoveIII(i,i+1,i+2)
    I = [0 for index in range(len(List_invariants))]
    for index in range(len(I)):
        I[index] = invariant_compute(d,List_invariants[index])
    s = d.get_sign()
    for index in range(len(List_invariants)):
        if test_type(d,GD[index]):
            tempcont = 1
            for jindex in range(len(List_invariants[index])):
                tempcont = tempcont*x[jindex]**I[index][jindex]
```

```
                cont[index] = cont[index] + s*tempcont
                if debug:
                    print(d.get_Class())
                    plt.figure()
                    d.plot_diagram()
                    print(s*tempcont)
                    print(invariant_compute(d,
                                List_invariants[index],debug))
        diag.append(d)
        diag.append(deepcopy(D1))
        singular.append(d)
        #Commutation move : 12-13121 = 123-1121
        D1.CommutationMove(i+2)
        diag.append(deepcopy(D1))
        #Reidemeister move II : 123-1121 --> 12321
        D1.ReidemeisterMoveII_delete(i+3)
        diag.append(deepcopy(D1))
        #Reidemeister move II : 12321 --> 1231-121
        D1.ReidemeisterMoveII_create(1,i+3)
        diag.append(deepcopy(D1))
        #Reidemeister move III : 1231-121 --> 123121-2 ====> 123121 -2
        d = D1.ReidemeisterMoveIII(i+4,i+5,i+6)
        I = [0 for index in range(len(List_invariants))]
        for index in range(len(I)):
            I[index] = invariant_compute(d,List_invariants[index])
        s = d.get_sign()
        for index in range(len(List_invariants)):
            if test_type(d,GD[index]):
                tempcont = 1
                for jindex in range(len(List_invariants[index])):
                    tempcont = tempcont*x[jindex]**I[index][jindex]
                cont[index] = cont[index] + s*tempcont
                if debug:
                    print(d.get_Class())
                    plt.figure()
                    d.plot_diagram()
                    print(s*tempcont)
                    print(invariant_compute(d,
                                List_invariants[index],debug))
        diag.append(d)
        diag.append(deepcopy(D1))
        singular.append(d)

#Pushing through 3 : 3 123121
```

```
elif ei == 3 :
    #Commutation move : 3123121 = 1323121
    D1.CommutationMove(i)
    diag.append(deepcopy(D1))
    #Reidemeister move III : 1323121 --> 1232121
    d = D1.ReidemeisterMoveIII(i+1,i+2,i+3)
    I = [0 for index in range(len(List_invariants))]
    for index in range(len(I)):
        I[index] = invariant_compute(d,List_invariants[index])
    s = d.get_sign()
    for index in range(len(List_invariants)):
        if test_type(d,GD[index]):
            tempcont = 1
            for jindex in range(len(List_invariants[index])):
                tempcont = tempcont*x[jindex]**I[index][jindex]
            cont[index] = cont[index] + s*tempcont
            if debug:
                print(d.get_Class())
                plt.figure()
                d.plot_diagram()
                print(s*tempcont)
                print(invariant_compute(d,
                                        List_invariants[index],debug))
    diag.append(d)
    diag.append(deepcopy(D1))
    singular.append(d)
    #Reidemeister move III : 1232121 --> 1231211 ====> 123121 1
    d = D1.ReidemeisterMoveIII(i+3,i+4,i+5)
    I = [0 for index in range(len(List_invariants))]
    for index in range(len(I)):
        I[index] = invariant_compute(d,List_invariants[index])
    s = d.get_sign()
    for index in range(len(List_invariants)):
        if test_type(d,GD[index]):
            tempcont = 1
            for jindex in range(len(List_invariants[index])):
                tempcont = tempcont*x[jindex]**I[index][jindex]
            cont[index] = cont[index] + s*tempcont
            if debug:
                print(d.get_Class())
                plt.figure()
                d.plot_diagram()
                print(s*tempcont)
                print(invariant_compute(d,
```

```
                                    List_invariants[index],debug))
        diag.append(d)
        diag.append(deepcopy(D1))
        singular.append(d)

    #Pushing through -3 : -3 123121
    elif ei == -3 :
        #Commutation move : -3123121 = 1-323121
        D1.CommutationMove(i)
        diag.append(deepcopy(D1))
        #Reidemeister move III : 1-323121 --> 123-2121
        d = D1.ReidemeisterMoveIII(i+1,i+2,i+3)
        I = [0 for index in range(len(List_invariants))]
        for index in range(len(I)):
            I[index] = invariant_compute(d,List_invariants[index])
        s = d.get_sign()
        for index in range(len(List_invariants)):
            if test_type(d,GD[index]):
                tempcont = 1
                for jindex in range(len(List_invariants[index])):
                    tempcont = tempcont*x[jindex]**I[index][jindex]
                cont[index] = cont[index] + s*tempcont
                if debug:
                    print(d.get_Class())
                    plt.figure()
                    d.plot_diagram()
                    print(s*tempcont)
                    print(invariant_compute(d,
                                    List_invariants[index],debug))
        diag.append(d)
        diag.append(deepcopy(D1))
        singular.append(d)
        #Reidemeister move III : 123-2121 --> 12312-11
        d = D1.ReidemeisterMoveIII(i+3,i+4,i+5)
        I = [0 for index in range(len(List_invariants))]
        for index in range(len(I)):
            I[index] = invariant_compute(d,List_invariants[index])
        s = d.get_sign()
        for index in range(len(List_invariants)):
            if test_type(d,GD[index]):
                tempcont = 1
                for jindex in range(len(List_invariants[index])):
                    tempcont = tempcont*x[jindex]**I[index][jindex]
                cont[index] = cont[index] + s*tempcont
```

```
                if debug:
                    print(d.get_Class())
                    plt.figure()
                    d.plot_diagram()
                    print(s*tempcont)
                    print(invariant_compute(d,
                                    List_invariants[index],debug))
        diag.append(d)
        diag.append(deepcopy(D1))
        singular.append(d)
        #Reidemeister move II : 12312-11 --> 12312
        D1.ReidemeisterMoveII_delete(i+5)
        diag.append(deepcopy(D1))
        #Reidemeister move II : 12312 --> 123121-1 ====> 123121 -1
        D1.ReidemeisterMoveII_create(1,i+5)
        diag.append(deepcopy(D1))
    return D1,diag,singular,array(cont)

def add_delta(D,i):
    '''A function that adds Delta and Delta^-1
    to a braid closure at position i'''
    #In : G,S torus gauss code, braid closure
    #Out : H,R torus gauss code, braid closure
    D1 = deepcopy(D)
    diag = []
    #Adding Delta and Delta^-1
    d = D1.ReidemeisterMoveII_create(1,i)
    diag.append(d)
    d = D1.ReidemeisterMoveII_create(2,i+1)
    diag.append(d)
    d = D1.ReidemeisterMoveII_create(3,i+2)
    diag.append(d)
    d = D1.ReidemeisterMoveII_create(1,i+3)
    diag.append(d)
    d = D1.ReidemeisterMoveII_create(2,i+4)
    diag.append(d)
    d = D1.ReidemeisterMoveII_create(1,i+5)
    diag.append(d)

    return D1,diag

def rot(D,List_invariants,debug=False):
    '''A function that operates the loop rot(S) on a braid'''
    #In : S braid closure
```

```
#Out : H,R torus gauss code, braid closure
D1 = deepcopy(D)
n = D1.get_length()
Inv = [int(0) for index in range(len(List_invariants))]
#Case S = S'DD
R = D1.get_BraidClosure()
rot = [deepcopy(D1)]
rot_sing = []
if n >= 12:
    if (R[n-12],R[n-11],R[n-10],R[n-9],R[n-8],R[n-7],
        R[n-6],R[n-5],R[n-4],R[n-3],R[n-2],R[n-1]) ==
                            (1,2,3,1,2,1,1,2,3,1,2,1):
        #m is the size of S'
        m = n-12
        #Applying Delta once :
        for i in range(m):
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m-i-1,debug)
            Inv = Inv + cont
            rot.extend(diag)
            rot_sing.extend(sing)
        #Applying Delta a second time :
        for i in range(m):
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m+5-i,debug)
            Inv = Inv + cont
            rot.extend(diag)
            rot_sing.extend(sing)
    elif (R[n-6],R[n-5],R[n-4],R[n-3],R[n-2],R[n-1]) ==
                                (1,2,3,1,2,1):
        #Size of S'
        m = n-6
        #Adding Delta once
        D1,diag = add_delta(D1,n)
        rot.extend(diag)
        #Applying Delta once :
        for i in range(m):
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m-i-1,debug)
            Inv = Inv + cont
            rot.extend(diag)
            rot_sing.extend(sing)
        #Applying Delta a second time :
        for i in range(m):
```

```
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m+5-i,debug)
            Inv = Inv + cont
            rot.extend(diag)
            rot_sing.extend(sing)
    else:
        m = n
        #Adding Delta twice
        D1,diag = add_delta(D1,n)
        rot.extend(diag)
        D1,diag = add_delta(D1,n+6)
        rot.extend(diag)
        #Applying Delta once :
        for i in range(m):
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m-i-1,debug)
            Inv = Inv + cont
            rot.extend(diag)
            rot_sing.extend(sing)
        #Applying Delta a second time :
        for i in range(m):
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m+5-i,debug)
            Inv = Inv + cont
            rot.extend(diag)
            rot_sing.extend(sing)

elif n>= 6:
    if (R[n-6],R[n-5],R[n-4],R[n-3],R[n-2],R[n-1]) ==
                            (1,2,3,1,2,1):
        #Size of S'
        m = n-6
        #Adding Delta once
        D1,diag = add_delta(D1,n)
        rot.extend(diag)
        #Applying Delta once :
        for i in range(m):
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m-i-1,debug)
            Inv = Inv + cont
            rot.extend(diag)
            rot_sing.extend(sing)
        #Applying Delta a second time :
        for i in range(m):
```

```
                D1,diag,sing,cont = push_delta(D1,
                                    List_invariants,m+5-i,debug)
                Inv = Inv + cont
                rot.extend(diag)
                rot_sing.extend(sing)
        else:
            m = n
            #Adding Delta twice
            D1,diag = add_delta(D1,n)
            rot.extend(diag)
            D1,diag = add_delta(D1,n+6)
            rot.extend(diag)
            #Applying Delta once :
            for i in range(m):
                D1,diag,sing,cont = push_delta(D1,
                                    List_invariants,m-i-1,debug)
                Inv = Inv + cont
                rot.extend(diag)
                rot_sing.extend(sing)
            #Applying Delta a second time :
            for i in range(m):
                D1,diag,sing,cont = push_delta(D1,
                                    List_invariants,m+5-i,debug)
                Inv = Inv + cont
                rot.extend(diag)
                rot_sing.extend(sing)
    else:
        m = n
        #Adding Delta twice
        D1,diag = add_delta(D1,n)
        rot.extend(diag)
        D1,diag = add_delta(D1,n+6)
        rot.extend(diag)
        #Applying Delta once :
        for i in range(m):
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m-i-1,debug)
            Inv = Inv + cont
            rot.extend(diag)
            rot_sing.extend(sing)
        #Applying Delta a second time :
        for i in range(m):
            D1,diag,sing,cont = push_delta(D1,
                                List_invariants,m+5-i,debug)
```

```
                Inv = Inv + cont
                rot.extend(diag)
                rot_sing.extend(sing)
        return rot,rot_sing,Inv

    def List_sigmas(S,List_invariants,debug=False):
        D = BraidData(S)
        roti,rot_sing,Inv = rot(D,List_invariants,debug)
        return Inv
```

1.2 II - Examples and results

```
In [ ]: S = [1,2,3]
        print('Multivariable invariants : {}'
              .format(list(List_sigmas(S,T1mult))))
        print('Refined invariants : {}'
              .format(list(List_sigmas(S,T1raf))))
```

```
In [ ]: S = [2, -1, 2, -3, 2, 3, 2, 3, 3]
        print('Multivariable invariants : {}'
              .format(list(List_sigmas(S,T1mult))))
        print('Refined invariants : {}'
              .format(list(List_sigmas(S,T1raf))))
```

```
In [ ]: S = [2, 3, -1, -2, 3, 3, 2, 3, 3]
        print('Multivariable invariants : {}'
              .format(list(List_sigmas(S,T1mult))))
        print('Refined invariants : {}'
              .format(list(List_sigmas(S,T1raf))))
```

1.2.1 II.a. Functions to generate random braids

```
In [ ]: import random

        def random_braid(size=3):
            '''A function that generates
            a random 4-braid that is a knot'''
            isknot = False
            is4braid = False
            #We generate a random 4-braid until it is a knot
            while isknot == False or is4braid == False:
                elem = random.choice([-1,1,-2,2,-3,3])
                braid = [elem]
                #Generating a random braid
                while len(braid) < size:
```

```
                elem = random.choice([-1,1,-2,2,-3,3])
                if elem != -braid[-1]:
                    braid.append(elem)
            is4braid = min(abs(array(braid))) == 1
                and max(abs(array(braid))) == 3
            L = Link(B(braid))
            isknot = L.is_knot()
        return braid

    def random_positive_braid(size=3):
        '''A function that generates
        a random 4-braid that is a knot'''
        isknot = False
        is4braid = False
        #We generate a random 4-braid until it is a knot
        while isknot == False or is4braid == False:
            elem = random.choice([1,2,3])
            braid = [elem]
            #Generating a random braid
            while len(braid) < size:
                elem = random.choice([1,2,3])
                if elem != -braid[-1]:
                    braid.append(elem)
            is4braid = min(abs(array(braid))) == 1
                and max(abs(array(braid))) == 3
            L = Link(B(braid))
            isknot = L.is_knot()
        return braid
```

1.2.2 II.b. Looking for non invertible braids

```
In [ ]: def reverse_braid(S):
            '''A function that returns the reverse braid of S'''
            return [s for s in reversed(S)]

        def find_non_invertible_braid(braidlength=21,nbattempts=30):
            '''A function that find a non invertible braid'''
            attempt = 1
            found = False
            while found == False and attempt <= nbattempts:
                S = random_braid(braidlength)
        #         print(S)
        #         S = clear_all_negative_crossings(S)
        #         print(S)
```

```
#          L = Link(B(S))
#          print(L.is_knot())
            InvS = List_sigmas(S,T1raf)
            revS = reverse_braid(S)
            InvrevS = List_sigmas(revS,T1raf)
            print('Attempt n°{} : {},{}'
                  .format(attempt,S,list(InvS)==list(InvrevS)))
            print('   {}'.format(InvS))
            if list(InvS) != list(InvrevS) :
                found = True
            if attempt%10 == 0:
                print('{} tries'.format(attempt))
            attempt = attempt + 1
        if found:
            print('Found one !!!!!!!!!!')
            print('Braid : ',S)
            print(InvS)
            print('Inverted braid : ',revS)
            print(InvrevS)
        return 'OK'
```

```
In [ ]: find_non_invertible_braid(11,50)
```

1.2.3 II.c. Invariants dimension

```
In [ ]: def estimation_dimension(repeat=20,maxi=15,print_test=False):
            dim = 0
            if print_test:
                print('Start of the computations')
            #Repeating the process repeat times
            for k in range(repeat):
                if print_test:
                    print('   Process number {}'.format(k+1))
                    print('      Generating the matrix')
                M = []
                braid = []
                #Generating as much braids as index in T)
                for j in range(len(T1raf)):
#                   print(j)
                    #Generating a random size
                    size = 0
                    n2 = size%2
                    while (n2 == 0):
                        size = int(round(random.random()*maxi))
```

```
                    n2 = size%2
#                 print(size)
                #Generating a random braid of size size
                S = random_braid(size)
                if print_test:
                    print('         Braid : {}'.format(S))
                #Computing the different Invariants for S
                InvS = List_sigmas(S,T1raf)
                Lis = []
                for i in range(len(T1raf)):
                    Lis.append(InvS[i])
                if print_test:
                    print('         Invariants : {}'.format(Lis))
                #Storing the different datas
                braid.append(S)
                M.append(Lis)
                if print_test:
                    print('         Updating the matrices')
                    print('         Braids matrix : {}'.format(braid))
                    print('         Invariants matrix : {}'.format(M))
            #Computing the matrix rank
            M = array(M)
            dimM = linalg.matrix_rank(M)
            dim = max(dim,dimM)
            if print_test:
                print('      Final invariants matrix : {}'.format(M))
                print('      Matrix rank : {}'.format(dimM))
                print('      Dimension : {}'.format(dim))
        return dim
```

```
In [ ]: estimation_dimension(11,5,True)
```

1.2.4 II.d. Functions to compare the invariants with finite type invariants defined by Gauss formulas

```
In [ ]: def circle(center, radius,color):
            ax = plt.axes()
            theta = linspace(0, 2*pi, 100)
            x1 = center[0]+radius*cos(theta)
            x2 = center[1]+radius*sin(theta)
            plt.plot(x1,x2,color)
            #plt.xlim(-10,10)
            #plt.ylim(-10,10)
            ax.set_aspect(1)
```

```
    return 0

def compute_color(homotopy):
    if homotopy == 1:
        return 'red'
    elif homotopy == 2:
        return 'blue'
    elif homotopy == 3:
        return 'green'
    else:
        return 'black'

class GaussArrow:
    "An arrow in a Gauss diagram "
    def __init__(self,foot=-1,head=-1,homotopy=0):
        self.__foot = foot
        self.__head = head
        self.__homotopy = homotopy
        self.__color = compute_color(homotopy)
    def set_foot(self,f):
        self.__foot = f
    def set_head(self,h):
        self.__head = h
    def set_homotopy(self,h):
        self.__homotopy = h
        self.__color = compute_color(h)
    def get_foot(self):
        return self.__foot
    def get_head(self):
        return self.__head
    def get_homotopy(self):
        return self.__homotopy
    def get_color(self):
        return self.__color
    def get_infos(self):
        print( 'Foot coordinate :',self.__foot)
        print( 'Head coordinate :',self.__head)
        print( 'Homotopy class:',self.__homotopy)
        print( 'Color :',self.__color)
    def plot_arrow(self,center=[0,0],nbarrows=1):
        #Computing the arrow coordinates in the circle
        x = center[0]
        y = center[1]
        fangle = self.get_foot()*(2*pi/(2*nbarrows)) + pi/(2*nbarrows)
```

```
            hangle = self.get_head()*(2*pi/(2*nbarrows)) + pi/(2*nbarrows)
            foot = (x + cos(fangle),y + sin(fangle))
            head = (x + cos(hangle),y + sin(hangle))
            color = self.get_color()
            #Plotting the arrow
            plt.annotate('', xy = head, xytext  = foot,
                             arrowprops = {'color': color,'arrowstyle' : '->'})
        def __eq__(self,A):
            if A.__class__.__name__ == 'GaussArrow':
                if self.get_foot() == A.get_foot()
                and self.get_head() == A.get_head()
                and self.get_homotopy() == A.get_homotopy():
                    return True
                else:
                    return False
            else:
                return False
    class GaussDiagram:
        "A circle together with marked arrows"
        def __init__(self,arrows=[]):
            self.__arrowslist = arrows
        def set_arrowslist(self,arrows):
            self.__arrowslist = arrows
        def get_arrowslist(self):
            return self.__arrowslist
        def add_arrow(self,a):
            self.__arrowslist.append(a)
        def del_arrow(self,i):
            '''A method that the arrow that
            is in position i in the arrowslist'''
            self.__arrowslist.pop(i)
        def plot_diagram(self,center=[0,0]):
            #Plotting the circle
            x = center[0]
            y = center[1]
            circle((x,y),1,'black')
            plt.axis('off')
            #Plotting the arrows
            nbarrows = len(self.get_arrowslist())
            for arrow in self.get_arrowslist():
                arrow.plot_arrow(center,nbarrows)
        def __eq__(self,G):
            if G.__class__.__name__ == 'GaussDiagram':
                if self.get_arrowslist() == G.get_arrowslist():
```

```
                return True
            else:
                return False
        else:
            return False

class GaussDiagramWithCoefficient(GaussDiagram):
    "A Gauss Diagram with Coefficient"
    def __init__(self,arrows=[],c=1):
        GaussDiagram.__init__(self,arrows)
        self.__coefficient = c
    def set_coefficient(self,coeff):
        self.__coefficient = coeff
    def get_coefficient(self):
        return self.__coefficient
    def __mul__(self,n):
        G = deepcopy(self)
        G.set_coefficient(self.get_coefficient()*n)
        return G
    def __rmul__(self,n):
        return self*n
    def __eq__(self,G):
        if G.__class__.__name__ == 'GaussDiagramWithCoefficient':
            if G != 0 and self.get_arrowslist() ==
                G.get_arrowslist():
                return True
            elif G == 0 and self.get_coefficient() == 0:
                return True
            else:
                return False
        else:
            return False
    def plot_diagram(self,center=[0,0]):
        #Plotting the circle
        x = center[0]
        y = center[1]
        circle((x,y),1,'black')
        plt.axis('off')
        #Plotting the arrows
        nbarrows = len(self.get_arrowslist())
        for arrow in self.get_arrowslist():
            arrow.plot_arrow(center,nbarrows)
        #Plotting the coefficient
        x = center[0] - 2 - 0.2
```

```
        y = center[1] - 0.1
        coeff = self.get_coefficient()
        if coeff > 1:
            coeff = '+ ' + str(coeff)
        elif coeff == 1:
            coeff = '+'
        elif coeff == -1:
            coeff = '-'
#         coeff = ''
        plt.text(x,y,coeff,color='black',size='xx-large')

class LinearCombinationOfGaussDiagrams:
    "A linear combination of Gauss diagrams"
    def __init__(self,GD=[]):
        self.__name = 'INIT'
        if GD.__class__.__name__ ==
            'GaussDiagramWithCoefficient' :
            self.__expression = [GD]
        elif GD != []:
            D = GaussDiagramWithCoefficient(GD)
            self.__expression = [D]
        else:
            self.__expression = []
    def set_name(self,value):
        self.__name = value
    def get_expression(self):
        return self.__expression
    def get_name(self):
        return self.__name
    def update(self):
        Lis = []
        for GD in self:
            if GD.get_coefficient() == 0:
                Lis.append(GD)
        for GD in Lis:
            self.get_expression().remove(GD)
    def __len__(self):
        return len(self.__expression)
    def __getitem__(self, key):
        return self.get_expression()[key]
    def __setitem__(self, key, value):
        self.__expression[key] = value
    def __delitem__(self, key):
        self.__expression.pop(key)
```

```
def __missing__(self, key):
    print( '%i is not a valid index'% key)
def __iter__(self):
    return iter(self.__expression)
def __reversed__(self):
    return reversed(self.__expression)
def __contains__(self, item):
    ans = False
    for GD in self:
        if GD == item:
            ans = True
    return ans
def __mul__(self,n):
    E = deepcopy(self)
    for i in range(len(self)):
        E[i] = E[i]*n
    #E.update()
    return E
def __rmul__(self,n):
    return self*n
def __add__(self, T):
    E = deepcopy(self)
    if T.__class__.__name__ ==
        'LinearCombinationOfGaussDiagrams':
        for GD in T:
            boolin = False
            for i in range(len(self)):
                if E[i] == GD:
                    E[i].set_coefficient(
                        E[i].get_coefficient() +
                        GD.get_coefficient())
                    boolin = True
            if boolin == False:
                E.get_expression().append(GD)
    elif T.__class__.__name__ ==
        'GaussDiagramWithCoefficient':
        boolin = False
        for i in range(len(self)):
            if E[i] == T:
                E[i].set_coefficient(
                    E[i].get_coefficient() +
                    T.get_coefficient())
                boolin = True
        if boolin == False:
```

```
            E.get_expression().append(T)
    #E.update()
    return E
def __neg__(self):
    return (-1)*self
def __sub__(self,GD):
    return self + (-GD)
def plot_expression(self,m=0):
    fig = plt.figure(m, figsize=(6.4*len(self), 7))
    #Plotting the name
    if self.get_name() != 'INIT':
        x = - 0.2 - 4
        y = - 0.1
        #plt.title(self.get_name())
        plt.text(x,y,self.get_name(),color='black',size='xx-large')
        plt.text(x+1,y,'=',color='black',size='xx-large')
    if len(self) <= 5 :
        plt.xlim(-2,len(self)*4)
        plt.ylim(-1.2,1.2)
        center=array([-4,0])
        for GD in self:
            center = center + array([4,0])
            GD.plot_diagram(center)
    elif len(self) <= 10:
        plt.xlim(-2,20)
        plt.ylim(-3.7,1.2)
        center=array([-4,0])
        for i in range(5):
            GD = self[i]
            center = center + array([4,0])
            GD.plot_diagram(center)
        center=array([-4,-2.5])
        for i in range(5,len(self)):
            GD = self[i]
            center = center + array([4,0])
            GD.plot_diagram(center)
    else:
        plt.xlim(-2,20)
        plt.ylim(-6.2,1.2)
        center=array([-4,0])
        for i in range(5):
            GD = self[i]
            center = center + array([4,0])
            GD.plot_diagram(center)
```

```
            center=array([-4,-2.5])
            for i in range(5,10):
                GD = self[i]
                center = center + array([4,0])
                GD.plot_diagram(center)
            center=array([-4,-5])
            for i in range(10,len(self)):
                GD = self[i]
                center = center + array([4,0])
                GD.plot_diagram(center)

#1
a = GaussArrow(2,1,1)
b = GaussArrow(3,0,2)
D1 = LinearCombinationOfGaussDiagrams([a,b])
#2
c = GaussArrow(3,0,3)
D2 = LinearCombinationOfGaussDiagrams([a,c])
#3
d = GaussArrow(2,1,2)
D3 = LinearCombinationOfGaussDiagrams([c,d])
#4
e = GaussArrow(2,0,1)
f = GaussArrow(3,1,3)
D4 = LinearCombinationOfGaussDiagrams([e,f])
#5
g = GaussArrow(3,1,2)
D5 = LinearCombinationOfGaussDiagrams([e,g])
#6
h = GaussArrow(3,1,1)
k = GaussArrow(2,0,2)
D6 = LinearCombinationOfGaussDiagrams([h,k])
#7
D7 = LinearCombinationOfGaussDiagrams([f,k])
#8
l = GaussArrow(2,0,3)
D8 = LinearCombinationOfGaussDiagrams([l,h])
#9
D9 = LinearCombinationOfGaussDiagrams([g,l])
#10
m = GaussArrow(0,3,2)
D10 = LinearCombinationOfGaussDiagrams([a,m])
#11
o = GaussArrow(0,3,1)
```

```
D11 = LinearCombinationOfGaussDiagrams([a,o])
#12
p = GaussArrow(1,2,3)
D12 = LinearCombinationOfGaussDiagrams([p,b])
#13
D13 = LinearCombinationOfGaussDiagrams([p,c])

def plot_invariant(l2,l3,l4,l5,l6,l8,l10,l12,name,m=0):
    l1 = -l3+l4+l12-2*l2+l8+l10
    l7 = l6+l3+l2-l8-l10
    l9 = l3-l4+l2-l10+l5
    l11 = -l6-2*l3+2*l4+2*l12-4*l2+2*l8+2*l10-l5
    l13 = -l6+l4-2*l2+l8+2*l10-l5
    E = l1*D1+l2*D2+l3*D3+l4*D4+l5*D5+l6*D6+l7*D7+
    l8*D8+l9*D9+l10*D10+l11*D11+l12*D12+l13*D13
    E.update()
    E.set_name(name)
    E.plot_expression(m)

def generate_finite_invariant(l2,l3,l4,l5,l6,l8,l10,l12,name):
    l1 = -l3+l4+l12-2*l2+l8+l10
    l7 = l6+l3+l2-l8-l10
    l9 = l3-l4+l2-l10+l5
    l11 = -l6-2*l3+2*l4+2*l12-4*l2+2*l8+2*l10-l5
    l13 = -l6+l4-2*l2+l8+2*l10-l5
    E = l1*D1+l2*D2+l3*D3+l4*D4+l5*D5+l6*D6+l7*D7+
    l8*D8+l9*D9+l10*D10+l11*D11+l12*D12+l13*D13
    E.update()
    E.set_name(name)
    return E

Tf = []
Tf.append(generate_finite_invariant(1,0,0,0,0,0,0,0,
                                    '$I_{%d}$'%0))
Tf.append(generate_finite_invariant(0,1,0,0,0,0,0,0,
                                    '$I_{%d}$'%1))
Tf.append(generate_finite_invariant(0,0,1,0,0,0,0,0,
                                    '$I_{%d}$'%2))
Tf.append(generate_finite_invariant(0,0,0,1,0,0,0,0,
                                    '$I_{%d}$'%3))
Tf.append(generate_finite_invariant(0,0,0,0,1,0,0,0,
                                    '$I_{%d}$'%4))
Tf.append(generate_finite_invariant(0,0,0,0,0,1,0,0,
                                    '$I_{%d}$'%5))
```

```
Tf.append(generate_finite_invariant(0,0,0,0,0,0,1,0,
                                    '$I_{%d}$'%6))
Tf.append(generate_finite_invariant(0,0,0,0,0,0,0,1,
                                    '$I_{%d}$'%7))

def same_configuration(hi,hj,fi,fj,h1,h2,f1,f2):
    '''A function that tests if arrow i,j,1
    and 2 are in same configuration'''
    return (hi < hj) == (h1 < h2) and (hi < fi) == (h1 < f1)
    and (hi < fj) == (h1 < f2) and (hj < fi) == (h2 < f1)
    and (hj < fj) == (h2 < f2) and (fi < fj) == (f1 < f2)

def configuration_contribution(GD,TC):
    '''A function that computes all subconfigurations
    of a TorusGaussCode TC that satifies the condition of
    a GaussDiagramWithCoefficient GD that have two arrows'''
    #Getting the GD condition
    arrow1 = GD.get_arrowslist()[0]
    arrow2 = GD.get_arrowslist()[1]
    hom1 = arrow1.get_homotopy()
    hom2 = arrow2.get_homotopy()
    h1 = arrow1.get_head()
    f1 = arrow1.get_foot()
    h2 = arrow2.get_head()
    f2 = arrow2.get_foot()
    Cont = 0

    #Scanning all arrows of TC
    for i in range(len(TC.get_Arrows())):
        arrowi = TC.get_Arrows()[i]
        homi = arrowi.get_homotopy()
        wi = arrowi.get_writhe()
        fi,hi = arrowi.get_coordinates()
        #Scanning all arrows after arrow i
        for j in range(i+1,len(TC.get_Arrows())):
            arrowj = TC.get_Arrows()[j]
            homj = arrowj.get_homotopy()
            wj = arrowj.get_writhe()
            fj,hj = arrowj.get_coordinates()
            #Checking if the couple satisfies the GD condition
            if homi == hom1 and homj == hom2:
                #Checking for the 4 possible configurations of GD
                for k in range(4):
                    if same_configuration(hi,hj,fi,fj,h1,h2,f1,f2):
```

```
                            Cont = Cont + (wi*wj)
                        h1 = (h1 - 1)%4
                        f1 = (f1 - 1)%4
                        h2 = (h2 - 1)%4
                        f2 = (f2 - 1)%4
                elif homj == hom1 and homi == hom2:
                    #Checking for the 4 possible configurations of GD
                    for k in range(4):
                        if same_configuration(hj,hi,fj,fi,h1,h2,f1,f2):
                            Cont = Cont + (wi*wj)
                        h1 = (h1 - 1)%4
                        f1 = (f1 - 1)%4
                        h2 = (h2 - 1)%4
                        f2 = (f2 - 1)%4
        return Cont

    def compute_finite_invariant(I,S):
        '''A function that compute the value
        of a Gauss diagram formulae I on a braid data D '''
        D = BraidData(S)
        TC = D.get_TorusGaussCode()
        Inv = 0
        for GD in I:
            Inv = Inv + configuration_contribution(GD,TC)*GD.get_coefficient()
        return Inv

    def print_all_finite_invariants(S):
        for i in range(len(Tf)):
            print( 'I%d = '%i, compute_finite_invariant(Tf[i],S))
```

List of the different Gauss formulas

```
In [ ]: plot_invariant(1,0,0,0,0,0,0,0,'$C_{%d}$'%0,0)
        plot_invariant(0,1,0,0,0,0,0,0,'$C_{%d}$'%1,1)
        plot_invariant(0,0,1,0,0,0,0,0,'$C_{%d}$'%2,2)
        plot_invariant(0,0,0,1,0,0,0,0,'$C_{%d}$'%3,3)
        plot_invariant(0,0,0,0,1,0,0,0,'$C_{%d}$'%4,4)
        plot_invariant(0,0,0,0,0,1,0,0,'$C_{%d}$'%5,5)
        plot_invariant(0,0,0,0,0,0,1,0,'$C_{%d}$'%6,6)
        plot_invariant(0,0,0,0,0,0,0,1,'$C_{%d}$'%7,7)
```

Looking for a useful example of two braids that don't have the same invariants

```
In [ ]: def generate_all_finite_invariants(S):
            Lis = []
```

```
    for i in range(len(Tf)):
        Lis.append(compute_finite_invariant(Tf[i],S))
    return Lis

def compute_invariants_cocycle_and_finite_type(S):
    FInvS = generate_all_finite_invariants(S)
    InvCocS = List_sigmas(S,T1raf)
    InvDerivS = []
    for invariant in InvCocS:
        if invariant != 0 :
            InvDerivS.append(invariant.derivative(x0)(1,0,0,0,0,0))
        else:
            InvDerivS.append(0)
    return [FInvS,InvDerivS]

def recherche_exemple_pertinent(maxi=11,attempt=10,
                                perattempt=20,test=False):
    i = 1
    found = False
    if test:
        print('DEBUT DE LA RECHERCHE')
    while i <= attempt and found == False:
        # Tirage tresse 1
        if test:
            print('   Tentative numero {}'.format(i))
        size = 0
        while size%2 == 0:
            size = int(round(random.random()*maxi))
        S1 = random_braid(size)
        if test:
            print('      Tirage de la tresse 1 : {}'
                  .format(S1))
        [FInvS1,InvDerivS1] =
            compute_invariants_cocycle_and_finite_type(S1)
        if test:
            print('         Invariants de type fini : {}'
                  .format(FInvS1))
            print('         Invariants de cocycle : {}'
                  .format(InvDerivS1))
            print('      Tirage de la tresse 2 :')
        #Tirage tresse 2
        j = 1
        size = 0
        while size%2 == 0:
```

```
            size = int(round(random.random()*maxi))
        S2 = random_braid(size)
        [FInvS2,InvDerivS2] =
            compute_invariants_cocycle_and_finite_type(S2)
        if test:
            print('         {}'.format(S2))
            print('            {}'.format(FInvS1==FInvS2))
        while j <= perattempt and FInvS1 != FInvS2:
            size = 0
            while size%2 == 0:
                size = int(round(random.random()*maxi))
            S2 = random_braid(size)
            if test:
                print('         {}'.format(S2))
            [FInvS2,InvDerivS2] =
                compute_invariants_cocycle_and_finite_type(S2)
            if test:
                print('            {}'.format(FInvS1==FInvS2))
            j = j + 1
        #Verification de si on l'a trouvée ou pas
        if FInvS1 == FInvS2 and InvDerivS1 != InvDerivS2:
            found = True
            if test:
                print('EXEMPLE TROUVE : {} et {}'.format(S1,S2))
        i = i + 1
    return S1,S2,FInvS1,FInvS2,InvDerivS1,InvDerivS2
```

```
In [ ]: recherche_exemple_pertinent(11,30,10,True)
```

Example :

```
In [ ]: S1 = [3, -2, 1, 3, -2, 3, 1]
        S2 = [1, -3, 2]
        [FInvS1,InvDerivS1] =
            compute_invariants_cocycle_and_finite_type(S1)
        [FInvS2,InvDerivS2] =
            compute_invariants_cocycle_and_finite_type(S2)
        print('S1 : {}'.format(S1))
        print('S2 : {}'.format(S2))
        print('Invariants de type 2 : ')
        print('   Pour S1 : {}'.format(FInvS1))
        print('   Pour S2 : {}'.format(FInvS2))
        print('Invariants de cocycles combinatoires : ')
        print('   Pour S1 : {}'.format(InvDerivS1))
        print('   Pour S2 : {}'.format(InvDerivS2))
```

1.2.5 II.e. Behavior of the invariants with Δ^{2n}

```
In [ ]: def print_results_delta2n(sizemax=7,nmax=4,tries=1):
            Delta = [1,2,3,1,2,1]
            Delta2 = Delta+Delta
            for i in range(tries):
                size = 0
                while size%2 == 0 or size < 3:
                    size = int(round(random.random()*sizemax))
                S = random_braid(size)
                while S[-1] == -1:
                    S = random_braid(size)
                n = 0
                print('For the braid {}'.format(S))
                Inv = list(List_sigmas(S,T1mult))
                print('   Multivariable invariants for n = {} : '
                      .format(n))
                print('      (1,1,-) : {}'.format(Inv[0]))
                print('      (1,2,-) : {}'.format(Inv[1]))
                print('      (2,1,-) : {}'.format(Inv[2]))
                print('      (2,3,+) : {}'.format(Inv[3]))
                print('      (3,2,+) : {}'.format(Inv[4]))
                print('      (2,3,+) : {}'.format(Inv[5]))
                while n < nmax :
                    n = n + 1
                    S = S + Delta2
                    Inv = list(List_sigmas(S,T1mult))
                    print('   Multivariable invariants for n = {} : '
                          .format(n))
                    print('      (1,1,-) : {}'.format(Inv[0]))
                    print('      (1,2,-) : {}'.format(Inv[1]))
                    print('      (2,1,-) : {}'.format(Inv[2]))
                    print('      (2,3,+) : {}'.format(Inv[3]))
                    print('      (3,2,+) : {}'.format(Inv[4]))
                    print('      (2,3,+) : {}'.format(Inv[5]))
```

```
In [ ]: print_results_delta2n(tries=10)
```

1.3 III - Clearing all negative crossings (for versions 8.8 and above)

This program doesn't work for Sagemath 8.8 and above if the braid has negative crossings. So we get around that problem using Δ^2.

```
In [ ]: def is_positive_braid(S):
            is_positive = True
```

```
    i = 0
    while i < len(S) and is_positive == True :
        s = S[i]
        if s < 0:
            is_positive = False
        i = i + 1
    return is_positive

def push_delta2_positive(S,i):
    '''a function that pushes Delta2 through S[i]'''
    #Suppose we have S of the form S'Delta2S''
    s = S[i]
    newS = [S[j] for j in range(i)]
    Delta2 = [1,2,3,1,2,1,1,2,3,1,2,1]
    newS.extend(Delta2)
    newS.append(s)
    newS.extend([S[j] for j in range(i+13,len(S))])
    return newS

def clear_negative_crossing(S,i):
    '''a function that clears a negative crossing S[i]
    assuming it is followed by Delta2'''
    s = S[i]
    newS = [S[j] for j in range(i)]
    #If s == -1, replace -1123121123121 by 23121123121
    if s == -1:
        newS.extend([2,3,1,2,1,1,2,3,1,2,1])
        newS.extend([S[j] for j in range(i+13,len(S))])
    elif s == -2:
        newS.extend([1,2,3,2,1,1,2,3,1,2,1])
        newS.extend([S[j] for j in range(i+13,len(S))])
    elif s == -3:
        newS.extend([1,2,3,1,2,1,2,3,1,2,1])
        newS.extend([S[j] for j in range(i+13,len(S))])
    return newS

def clear_all_negative_crossings(S):
    '''a function that clears all negative crossings from a braid S'''
    newS = deepcopy(S)
    Delta2 = [1,2,3,1,2,1,1,2,3,1,2,1]
    while is_positive_braid(newS) == False:
        i = len(newS)-1
        newS.extend(Delta2)
        #On pousse Delta2 jusqu'au prochain croisement négatif
```

```
            while newS[i] > 0 :
                newS = push_delta2_positive(newS,i)
                i = i - 1
            #On élimine le croisement négatif
            newS = clear_negative_crossing(newS,i)
        return newS
```

Example

```
In [ ]: S = [-1,2,3,1,2,3,-1,2,-3,2,-1,1,2,1]
        print(clear_all_negative_crossings(S))
```

```
In [ ]: S = [1,2,3]
        print(clear_all_negative_crossings(S))
```

2 ----------

2.1 "User-friendly" interface

Just put your braid after $S = $. The braid has to be of the form $[i]$ where i stands for the crossing σ_i. Then, execute all the next cells :

```
In [ ]: S =
```

The knot diagram of the braid in the solid torus :

```
In [ ]: D = BraidData(S)
        D.plot_braid_diagram()
```

The invariants :

```
In [ ]: # Uncomment for n=4 to get the multivariable invariants
        # print('Multivariable invariants : {}'.format(list(List_sigmas(S,T1mult))))
        print('Refined invariants : {}'.format(list(List_sigmas(S,T1raf))))
```

Example :

```
In [ ]: S = [1,2,3,1,2,3,1,2,1,1,2,3,1,2,1]
```

```
In [ ]: D = BraidData(S)
        D.plot_braid_diagram()
```

```
In [ ]: # Uncomment for n=4 to get the multivariable invariants
        # print('Multivariable invariants : {}'.format(list(List_sigmas(S,T1mult))))
        print('Refined invariants : {}'.format(list(List_sigmas(S,T1raf))))
```

```
In [ ]:
```

Bibliographie

[1] J. W. Alexander. “A Lemma on Systems of Knotted Curves”. In : *Proceedings of the National Academy of Sciences* 9.3 (1923), p. 93-95. doi : 10.1073/pnas.9.3.93. url : https://doi.org/10.1073/pnas.9.3.93.

[2] J. W. Alexander et G. B. Briggs. “On Types of Knotted Curves”. In : *Annals of Mathematics* 28.1/4 (1926). Full publication date : 1926 - 1927, p. 562-586. issn : 0003486X. doi : 10.2307/1968399. url : https://doi.org/10.2307/1968399.

[3] J.W. Alexander. “Topological invariants of knots and links”. In : *Transactions of the American Mathematical Society* 30 (1928). doi : 10.1090/S0002-9947-1928-1501429-1. url : https://doi.org/10.1090/S0002-9947-1928-1501429-1.

[4] E. Artin. “Theory of Braids”. In : *The Annals of Mathematics* 48.1 (1947), p. 101. doi : 10.2307/1969218. url : https://doi.org/10.2307/1969218.

[5] Emil Artin. “Theorie der Zöpfe”. In : *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 4.1 (1925), p. 47-72. issn : 1865-8784. doi : 10.1007/BF02950718. url : https://doi.org/10.1007/BF02950718.

[6] Dror Bar-Natan. “On the Vassiliev knot invariants”. In : *Topology* 34.2 (1995), p. 423-472. issn : 0040-9383. doi : https://doi.org/10.1016/0040-9383(95)93237-2. url : https://www.sciencedirect.com/science/article/pii/0040938395932372.

[7] Joan Birman et William Menasco. “On Markov’s Theorem”. In : (fév. 2002).

[8] Sergei Chmutov et Michael Polyak. “Elementary Combinatorics of the HOMFLYPT Polynomial”. In : *International Mathematics Research Notices* 2010.3 (2009), p. 480-495. issn : 1073-7928. doi : 10.1093/imrn/rnp137. url : http://dx.doi.org/10.1093/imrn/rnp137.

[9] T. Fiedler. *Polynomial One-cocycles for Knots and Closed Braids*. Sept. 2019. isbn : 978-981-12-1029-7. doi : 10.1142/11551.

[10] Thomas Fiedler. *Knot polynomials from 1-cocycles*. 2019. arXiv : 1709.10332 [math.GT].

[11] Thomas Fiedler. *One-cocycle invariants for closed braids*. 2018. arXiv : 1804.03549 [math.GT].

[12] Thomas FIEDLER et Vitaliy KURLIN. “A 1-parameter approach to links in a solid torus”. In : *J. Math. Soc. Japan* 62.1 (jan. 2010), p. 167-211. doi : 10.2969/jmsj/06210167. url : https://doi.org/10.2969/jmsj/06210167.

[13] Peter Freyd et al. “A new polynomial invariant of knots and links”. In : *Bull. Amer. Math. Soc.* 12 (avr. 1985). doi : 10.1090/S0273-0979-1985-15361-3.

[14] Guillaume Gandolfi. “Résultats sur les extensions singulières des groupes d’Artin et de tresses virtuelles”. Theses. Normandie Université, 2020. url : https://tel.archives-ouvertes.fr/tel-03103318.

[15] C.F. Gauss et al. *Carl Friedrich Gauss Werke*. Carl Friedrich Gauss Werke. Gedruckt in der Dieterichschen Universitätsdruckerei (W.F. Kaestner), 1903.

[16] C.McA. Gordon et J. Luecke. "Knots are determined by their complements". In : *Journal of the American Mathematical Society* 2 (1989). doi : 10.1090/S0894-0347-1989-0965210-7. url : https://doi.org/10.1090/S0894-0347-1989-0965210-7.

[17] André Gramain. "Sur le groupe fondamental de l'espace des noeuds". fr. In : *Annales de l'Institut Fourier* 27.3 (1977), p. 29-44. doi : 10.5802/aif.660. url : http://www.numdam.org/articles/10.5802/aif.660/.

[18] M. N. Gusarov. "Then-equivalence of knots and invariants of finite degree". In : *Journal of Mathematical Sciences* 81.2 (1996), p. 2549-2561. issn : 1573-8795. doi : 10.1007/BF02362425. url : https://doi.org/10.1007/BF02362425.

[19] V. F. R. Jones. "Hecke Algebra Representations of Braid Groups and Link Polynomials". In : *Annals of Mathematics* 126.2 (1987), p. 335-388. issn : 0003486X. url : http://www.jstor.org/stable/1971403.

[20] C. Kassel, O. Dodane et V. Turaev. *Braid Groups*. Graduate Texts in Mathematics. Springer New York, 2008. isbn : 9780387685489.

[21] L.H. Kauffman. *On Knots*. Annals of mathematics studies. Princeton University Press, 1987. isbn : 9780691084350.

[22] Louis H. Kauffman. *Knots and Physics*. T. Volume 1. Series on Knots and Everything. Volume 1. WORLD SCIENTIFIC, 1991, p. 552. doi : 10.1142/1116. url : https://doi.org/10.1142/1116.

[23] M. Kontsevich. "Vassiliev's knot invariants". In : *I. M. Gelfand Seminar, Partie 2* 16 (1993), p. 137-150.

[24] W. Magnus. "Über Automorphismen von Fundamentalgruppen berandeter Flächen". In : *Mathematische Annalen* 109 (1934), p. 617-646. url : http://eudml.org/doc/159699.

[25] a. Markov. "Über die freie Äquivalenz geschlossener Zöpfe." rus. In : *Matematiceskij sbornik* 43.1 (), p. 73-78. url : http://eudml.org/doc/64806.

[26] H.R. Morton. "Infinitely many fibred knots having the same Alexander polynomial". In : *Topology* 17.1 (1978), p. 101-104. issn : 0040-9383. doi : https://doi.org/10.1016/0040-9383(78)90016-2. url : https://www.sciencedirect.com/science/article/pii/0040938378900162.

[27] Luis Paris. *Braid groups and Artin groups*. 2007. arXiv : 0711.2372 [math.GR].

[28] Kurt Reidemeister. "Elementare Begründung der Knotentheorie". In : *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 5.1 (1927), p. 24-32. issn : 1865-8784. doi : 10.1007/BF02952507. url : https://doi.org/10.1007/BF02952507.

[29] Dale Rolfsen. *Knots and Links*. American Mathematical Society, 1976. isbn : 0821834363.

[30] E. Stein et al. *Braids, Links, and Mapping Class Groups*. Annals of mathematics studies. Princeton University Press, 1974. isbn : 9780691081496.

[31] William Thomson. "On Vortex Atoms". In : *Proceedings of the Royal Society of Edinburgh* 6 (1869), p. 94-105. doi : 10.1017/S0370164600045430.

[32] V. Vassiliev. "Cohomology of knot spaces". In : déc. 1990, p. 23-70. isbn : 9780821841006. doi : 10.1090/advsov/001/03.

1-cocycles pour les n-tresses fermées dans le tore solide qui sont des nœuds et algorithmes de calculs

Résumé

Ce manuscrit est un travail qui s'inscrit dans le cadre de la topologie, de l'algèbre, de la combinatoire et de la programmation. Plus précisément, c'est une thèse en théorie des noeuds. L'objectif de ce travail est de fournir une famille d'invariants permettant de distinguer les 4-tresses qui sont des noeuds (une famille particulière de noeuds) dans le tore solide $S^1 \times D^2$. La construction et le calcul de ces invariants utilise des notions élémentaires de la théorie des noeuds mais la preuve du théorème principal d'invariance nécessite des connaissances plus poussées en théorie des singularités. La compréhension du programme de calcul qui implémente ces invariants en Sagemath implique d'avoir des bases en programmation Python et en algorithmique (Programmation Orientée Objet, fonctions récursives, dictionnaires, etc...).

Mots-clés : noeuds, tresses, invariants, 1-cocycles, tore solide, diagrammes de Gauss.

1-cocycles for closed n-braids that are knots in the solid torus and computational algorithms

Abstract

This manuscript is a work within the scope of topology, algebra, combinatorics and programming. More precisely, it is a thesis in knot theory. The main goal of this manuscript is to provide a family of invariants that can distinguish 4-braids that are knots (a particular family of knots) in the solid torus $S^1 \times D^2$. The construction and the computation of these invariants use knot theory basics but the proof of the main invariance theorem requires more advanced knowledge in singularity theory. The understanding of the computational program that implements these invariants in Sagemath requires basic knowledge of Python programming and algorithmics (Oriented-Object Programming, recursive function theory, dictionaries, etc...).

Keywords : knots, braids, invariants, 1-cocycles, solid torus, Gauss diagrams.