Teoría de nudos y aplicaciones

Jorge Tejerina García

Trabajo de fin de grado en Matemáticas
Universidad de Zaragoza

Directores del trabajo:
José Ignacio Cogolludo Agustín
María Teresa Lozano Imízcoz

16 de Septiembre de 2016

Resumen

A lo largo de prácticamente todo el desarrollo tecnológico de la humanidad, los nudos han tenido un papel relevante debido a su singular geometría: desde las primeras herramientas ideadas por el hombre hasta la ingeniería de nuestros días.

Su particular estructura les hace habituales en fenómenos naturales e indispensables en el progreso del ser humano. Debido a todo esto, se ha ido generando un fuerte estudio matemático de lo denominado como nudo matemático (clásico): rudimentariamente, el resultado de anudar una cuerda y unir los extremos. Este objeto recoge en cierta medida la esencia de lo que nosotros entendemos como nudo y su estudio es fundamental en topología y geometría en particular, y en matemáticas en general, resultando útil también en otras disciplinas científicas como la bioquímica o la física.

Desarrollando las ideas de [1] (y usando [2] como apoyo), el presente trabajo consiste en una breve introducción a la teoría de nudos, con especial hincapié en la teoría de nudos virtuales (una joven rama de este campo con prometedoras perspectivas de futuro que estudia una generalización del concepto de nudo clásico) y su conexión (siendo ésta de carácter topológico) con unos elementos de naturaleza algebraica.

La memoria está dividida en tres partes fundamentales: preliminares (apéndice A), códigos de Gauss (capítulo 1) y planaridad (capítulo 2).

La lectura debe comenzar con el apéndice A: en éste se expone lo necesario para entender el desarrollo de los capítulos 1 y 2. Se introducen conceptos básicos de teoría de nudos y teoría de nudos virtuales y, debido a la fuerte analogía entre nudos y grafos, también de teoría de grafos. Esta parte presenta una visión general para posicionarse en el contexto de lo estudiado, proporcionando las herramientas para afrontar cuestiones más particulares.

En el capítulo 1 entraremos en algo más concreto: la capacidad de unas secuencias de números, letras y signos para representar los nudos virtuales, los llamados códigos de Gauss. Desarrollaremos de forma natural y detallada el proceso para demostrar que estos códigos verdaderamente sirven para representar nudos virtuales y nos familiarizaremos mejor con el entorno sobre el que estamos trabajando.

El último y segundo capítulo servirá para demostrar la utilidad de este nuevo enfoque para estudiar los nudos virtuales: el hecho de trabajar con códigos de Gauss nos permitirá obtener nuevos resultados y además, ser capaces de desarrollar algoritmos para el estudio de nudos virtuales.

En definitiva, se trata de una breve inmersión en una línea de investigación actual, la teoría de nudos virtuales, mostrando su motivación y (parte de) la utilidad de la misma. Como anotación final, cabe destacar que tanto el algoritmo que concluye el trabajo como los resultados que justifican la representación de nudos mediante códigos de Gauss están realizados por nosotros, basándonos en las ideas de [1], además de las figuras no referenciadas.

Abstract

From the most daily things, like tying our shoes or getting water from a well, to the most abstract, like mathematics or biochemistry, from the most ancient hunting weapons to nowadays fishing devices or from art to physics, knots has served as a powerful tool for the development of the human being.

It is its unique strucure what make them that special and necessary for certain tasks. Due to this, there is a strong line of investigation in mathematics entirely devoted to the understanding of their geometry called *Knot Theory*, or being more exact, to the study of the so-called *(classical) mathematical knot*: the resulting of knotting a rope and joining its endpoints (forming though an "intertwined circunference" in R^3).

The following paper consists in a clarification and extension of the ideas exposed in [1] (with the help of [2]):

Appendix A is a brief introduction to Knot Theory and the basics of it. In addtion, it will also be exposed *Virtual Knot Theory*, the study of *virtual knots*: a generalization of the concept of classical knot. In the following chapters we will see that, although they can´t (always) be seen in R^3, it is a very natural generalization in the context of topology that contributes to the understanding of classical knots: in the same way that prime numbers inherit what known of natural numbers, virtual knots help us with the study of classical knots. Moreover, some necessary concepts of Graph Theory will be reviewed due to the conexion between graphs and knots.

In some sense, although knots are defined from a topological/geometrical persepective, they have kind of discrete interpretation: they are determined by the number and types of intertwinings they have. This idea is captured by the algebraic concept of Gauss code, a sequence of numbers, letters and signs, which are capable to represent virtual knots. This new way of seeing virtual knots is what will be developed in chapter 1.

Chapter 2 is devoted to proving that this new prespective is able to provide us new results, in particular in identifying when a virtual knot is classical as well, using techniques based in algorithms and computable processes.

In conclusion, we will get involved in a current line of investigation in mathematcis, a field with promising future prospectives that contributes to the development of Knot Theory, fundamental in topology and geometry and with applications to a wide variety of disciplines.

Contents

Chapter 1

Gauss Codes

An interesting motivation for virtual knot theory is due to the so-called *Gauss codes*. The basic idea of this chapter is that, associating to each diagram a Gauss code, which consits in a specific sequence of numbers, signs and letters (that won´t be unique), we will get a new way to represent knots as classes of equivalence of these codes instead of as classes of equivalence of diagrams.

The algebraic nature of these codes is allow us uncover certain results hidden before, mainly about understanding when a virtual link is also classical or not, and will provide us a more computable approach to virtual knot theory.

1.1 Introduction

Gauss codes were introduced by C.F. Gauss in the XIX century. Although they haven't been much useful, an enriched version of them turns out to be interesting in virtual knot theory. Let's see what consists these codes in:

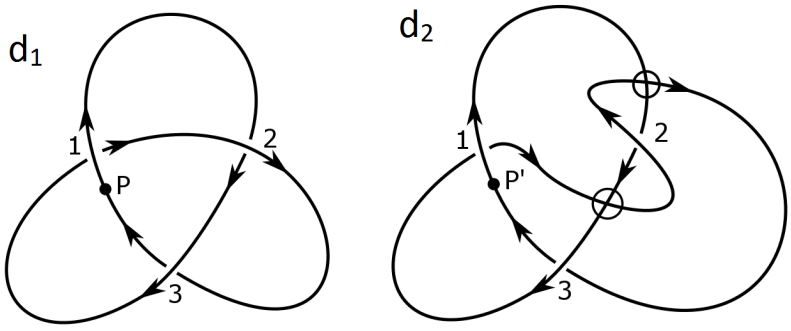
Definition 1.1.1. We say that a diagram d, with $n \in \mathbb{N}$ (non-virtual) crossing is a *labelled diagram* if every (non-virtual) crossing $i_j \in u$, with $j \in \{1,2,...,n\}$, is labelled as j.

Definition 1.1.2. Let K be a knot, d a labelled diagram of K in which we have fixed as + and - the two different possible orientations, with $n \in \mathbb{N}$ (non-virtual) crossings and a point $P \in d$. We call *Gauss code of d based in P with orientation k*, $g_P^k(d)$, to the sequence $g_P^k(d) = a_1b_1a_2b_2...a_{2n}b_{2n}$, where $a_i \in \mathbb{N}$ and $b_i \in \{O,U\}$ $\forall i \in \{1,2,...,2n\}$, obtained in the following way:

- Starting from P, we travel through d following orientation k until we reach P again. Every time we reach a (non-virtual) crossing (in j^{th} position, with $j \in \{1,2,...,2n\}$) we conclude:
  - $a_j = r$, where $r \in \{1,2,...,n\}$ is the label of the (non-virtual) crossing reached in j^{th} position.
  - $b_j = O$ if we overcross r in j^{th} position and $b_j = U$ if we undercross it.

Since we know that diagrams are 4-valent graphs with an extra structure in the vertices, then the number of appearances of every label in $gu_P^k(u)$ must be two.

In order to clarify this concept, let's check an example in Figure 1.1 (b): d_2 is a diagram of a knot K, similar to d_1 in Figure 1.1 (a) except for crossing 2. We start in P' following the given orientation (which we have denoted as + and - its opposite): We first face crossing 1 and start with 1O, then we have a virtual crossing so we add nothing (is like is not really there) and keep on checking. The following is crossing 2, so we have 1O2U. We get 1O2U3O in crossing 3 after a virtual crossing, and the next one is crossing 1, where we get 1O2U3O1U. To finish, we face crossing 2 and 3 before coming back to P', concluding that $g_P^+(d_2) = 1O2U3O1U2O3U$. Following the same scheme in the d_1 in Figure 1.1 (b)



(a) $g_P^+(d_1) = 1O2U3O1U2O3U$ (b) $g_P^+(d_2) = 1O2U3O1U2O3U$

Figure 1.1: Gauss codes examples.

(and denoting as + to the given orientation) , we get that $g_P^+(d_1) = 1O2U3O1U2O3U$.

Note that the nature of the diagram, and then of the knot, is captured in a Gauss code: it tells you how you have to "travel" in the plane to get the diagram you're dealing with, it gives the instructions you have to follow to draw it.

Definition 1.1.3. In general, we call *Gauss code* g to a sequence $g = a_1b_1a_2b_2...a_{2n}b_{2n}$ with $a_i \in \{1,2,...,n\}$ and $b_i \in \{O,U\}$ $\forall i \in \{1,2,...,2n\}$. The *length of* g is 2n and the set of Gauss codes will be denoted as GC.

Since our aim is to forget about diagramas and work with Gauss codes in an abstract sense, we will deal more with this last definition, but it is necessary to understand the motivation of the set we working are with. Let's connect both perspectives:

Definition 1.1.4. Given $g \in GC$, a labelled diagram d and orientations + and - fixed in it, we say that g satisfy d (or d satisfy g) if $\exists P \in d \ni g_P^k(d) = g$ for some $k \in \{+,-\}$.

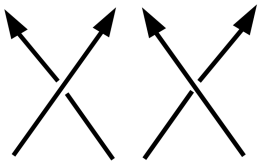
1.2 Adjustments in GC

Recall that our intention is to represent knots as classes of equivalence of Gauss codes. Nevertheless, we have the same Gauss code, 1O2U3O1U2O3U, for two non-equivalent diagrams, d_1 and d_2 from Figure 1.1 (in fact, one is classical the other is not ([1] page 668)). This makes us wonder if the information given in the Gauss code is not enough.

Unfortunately, this is not the only problem we will encounter. Since we need to define an equivalence relation in GC that behaves analogously to the given in D in a way that the natural representation (that is, every Gauss code represents the diagrams it satisfies) makes sense, we have the following problems to care about:

- (1): A Gauss code can represent two non-equivalent diagrams.
- (2): There may be Gauss codes that does not satisfy any diagram.
- (3): The same diagram satisfies more than one Gauss code. In a more general sense, two equivalent diagrams may satisfy two different Gauss codes (so, we will have to relate them in some way).

Let's check first problem (1): coming back to the Gauss codes in Figure 1.1, it is clear that they do not contain enough information to differenciate d_1 from d_2. As said before, d_1 and d_2 are equal except for crossing 2 and you can get d_2 by applying move in figure 1.3 (a) to crossing 2 in d_1, which doesn't preserve diagram equivalence in general.



(a) + (b) -

Figure 1.2: Sign associated to a crossing.

This change is not detected by the Gauss code. Nevertheless, the orientation given to the diagrams allows us to capture what happened in crossing 2, it has the form of Figure 1.2 (a) in Figure 1.1 (a) and of Figure 1.2 (b) in Figure 1.1 (b). These two types of crossings are essentially different: if we have the arc that overcrosses as our reference (with the arrow getting out from the crossing), in Figure 1.1 (a) we have that the arrow of the arc that undercrosses (with the arrow getting out from the crossing) is on the left of our reference and in Figure 1.1 (b) on the right. Every (non-virtual) crossing (in an oriented diagram) can be classified as one of these two types and, if the crossing is as in Figure 1.2 (a), we associate it sign + and - if it is as in Figure 1.2 (b).

In addition, although an orientation must be given to obtain the sign of a crossing, it does not depend on the orientation given: if we change the orientation of the diagram the sign of every crossing is preserved.

Coming back to Figure 1.1, what has actually happened is that the sign of crossing 2 has been changed: this is a curious phenomenon that arises when we admit virtual crossings in our diagrams. The appearance of virtual crossings permits, using moves in Figure 1.3, to change the sign of any crossing and keep any other information given in its Gauss codes invariant, in other words, we can get diagrams such that they have the same Gauss code but are essensially different.

Thus, we have to care about the sign information too:

Definition 1.2.1. Let K be a virtual knot, d a labelled diagram of K with $n \in \mathbb{N}$ (non-virtual) crossings in which we have fixed as + and - the two different possible orientations, and a point $P \in d$. We call *signed Gauss code of d based in P with orientation k*, $gs_P^k(d)$, to the sequence $gs_P^k(d) = a_1b_1c_1a_2b_2c_2...a_{2n}b_{2n}c_{2n}$ with $k \in \{+,-\}$, $n \in \mathbb{N}$, $a_i \in \mathbb{N}$, $b_i \in \{O,U\}$ and $c_i \in \{+,-\}$ $\forall i \in \{1,...,2n\}$, where a_i and b_i are obtained as allways and c_i is the sign of the crossing reached in i^{th} position. The sequence $s(gs_P^k) = c_1c_2...c_{2n}$ that contains the sign information is called the *sign sequence of the signed Gauss code*.

Definition 1.2.2. Given $n \in \mathbb{N}$, in general, a *signed Gauss code gs* is just a sequence $gs = a_1b_1c_1a_2b_2c_2...a_{2n}b_{2n}c_{2n}$ with $a_i \in \{1,2,...,n\}$, $b_i \in \{O,U\}$ and $c_i \in \{+,-\}$ $\forall i \in \{1,...,n\}$ and the sequence $s(gs) = c_1c_2...c_n$ that contains the sign information is called the sign sequence of gs. The *length of gs* is 2n and the set of all signed Gauss codes will be denoted as SGC.

As before, we will deal more with this last definition due to its abstract sense.

If we now come back again to Figure 1.1 (a) and Figure 1.1 (b) we can see that, if we add the sign sequence to the Gauss codes obtained before, then they are different, 1O+2U+3O+1U+2O+3U+ and 1O+2U+3O-1U+2O+3U- respectively. This is not casuality:

Proposition 1.2.1. Given d_1, d_2 two diagrams, $P_1 \in d_1$, $P_2 \in d_2$ and $k_1, k_2 \in \{+,-\}$. If $gs^{k_1}_{P_1}(d_1) = gs^{k_2}_{P_2}(d_2) \Rightarrow d_1 \sim d_2$.

Which gives an alternative to problem (1): signed Gauss codes distinguish between non-equivalent diagrams. Then, from now on we will be working in *SGC*.

On the other hand, not all signed Gauss codes can satisfy a diagram (problem(2)). For example, 3O+3O+ have no proper realization, since crossing 3 must be overcrossed twice and never undercrossed, which has no sense. All these codes will be a problem for us, they won´t represent any diagram and should be eliminated in order to adapt the set we are working with according to our purposes: we will clean SGC in a way that all the elements represent at least, one diagram:

Definition 1.2.3. Given $gs \in SGC$, gs is said to be *valid* if there exists a diagram satisfying it. We will denote the set of all valid signed Gauss codes as $VSGC$.

These codes can be easily characterized:

Proposition 1.2.2. $gs \in SGC$ is valid $\iff$ gs has exactly two appearances of each label, both O and U are associated to exactly one appearance of each label and the signs of the two appearances of each label coincide.

Proof. $\Rightarrow$): The negation of one of the three conditions of the hypothesis trivially implies that gs is not valid.
$\Leftarrow$): Let $gs = a_1b_1c_1a_2b_2c_2...a_{2n}b_{2n}c_{2n}$ be a code satisfying the hypothesis, with $n \in \mathbb{N}$. Let's construct a diagram satisfying it: in the plane, start from a point P, drawing from it an oriented straight line and put the labels in the order they appear until one label is repeated, then we go back with the line to this label by using no interection except from the needed when we reach the label. We continue drawing a straight line with labels until we reach one repeated and do as before. We will finish when we get to the last label. Notice that we can need extra intersections during the process, so we just consider these intersections as virtual crossings in order to generate no more crossings. When the last label is reached, join the end and the begining with a line putting virtual crossings in the intersections generated if necessary, generating thus a shadow u.

Now we travel again through u, starting from P and using the orientation fixed before and start giving the (O/U) structure of the code (with no ambiguity due to the (O/U) hypothesis). We have generated a diagram d.

Travel again through d as before and every time you reach a label:

- If the sign coincides with the one of the code, then keep on travelling.
- If the sign does not coincide we do the corresponding move in figure 1.3 to change it (with no extra consequences for the code but in the involved sign).

Notice that there will be no ambiguity in the sign of a label, since by hypothesis are equal for the two appearances in the code. This new diagram d clearly satisfies gs. □

(a) : + to -. (b) : - to +.

Figure 1.3: Change of sign moves.

From now, when we say *code* we will refer to a valid signed Gauss code.

This proposition gives us the oportunity to stop and think about an important remark in this paper: when we have a valid signed Gauss code what we actually have is an infinite amount of diagrams (all of them equivalent) represented by the code, and when we have a diagram, a finite number of codes. For every valid signed Gauss code there exists a diagram satisfying it and vice versa.

This seems to warn us that the working line we are following is not the correct one, that is not a good idea to connect diagramas and codes: how can we make a proper identification between them if, not only each diagram corresponds to more than one code, but also the same happens in the other direction?

The answer to the first lays in joining together all the codes that satisfy the same diagram:

Definition 1.2.4. Given a diagram d of a knot K, we denote by $C(d)$ the set of all the codes that satisfy d, which will be called *the set of representantives of* d.

In the context of codes, we call *move* any variation over a code. The set above can be characterized in terms of moves:

Definition 1.2.5. Given $n \in \mathbb{N}$ and $gs, gs^{'} \in VSGC$, we call *strong equivalence moves number 1,2 and 3* to the moves of the form

$$gs = a_1 b_1 c_1 a_2 b_2 c_2 ... a_{2n} b_{2n} c_{2n} \longrightarrow gs^{'} = a_1^{'} b_1^{'} c_1^{'} a_2^{'} b_2^{'} c_2^{'} ... a_{2n}^{'} b_{2n}^{'} c_{2n}^{'}$$

where:

- (1): $a_i^{'} = a_j^{'} \Longleftrightarrow a_i = a_j$, $b_i^{'} = b_i$ and $c_i^{'} = c_i \ \forall i, j \in \{1, 2, ..2n\}$.
- (2): $a_i^{'} = a_{l(i)}$, $b_i^{'} = b_{l(i)}$ and $c_i^{'} = c_{l(i)}$, where $l(i) = i - k \ (mod \ 2n)$ for a fixed $k \in \mathbb{N}$ and $\forall i, j \in \{1, 2, ..2n\}$.
- (3): $a_i^{'} = a_{2n-i+1}$, $b_i^{'} = b_{2n-i+1}$ and $c_i^{'} = c_{2n-i+1} \ \forall i \in \{1, 2, ..2n\}$.

Note that the length of the code is always preserved. Is easy to check that the geometrical interpretation of these moves is the following:

- (1): Re-labelling of the crossings.
- (2): Change of the point P of start.
- (3): Change of orientation.

So, these moves and the finite compositions of them capture all the possible variations in the parameters involved in obtaining a code from a certain diagram: the labelling, the point of start and the orientation choosen:

Definition 1.2.6. Given $gs_1, gs_2 \in VSGC$, we say gs_1 is *strongly equivalent* to gs_2, $gs_1 \simeq_o gs_2$, if gs_2 can be obtained by applying to gs_1 a finite number of strong equivalence moves.

Strong equivalence is clearly an equivalence relation in $VSGC$ and we will denote the corresponding classes as $[\cdot]_o$. The sets of representantives can be identified as these classes and finally characterized:

Proposition 1.2.3. Given a diagram d of a knot K and $gs \in C(d) \Longrightarrow C(d) = [gs]_o$.

Thus, we can now represent every diagram $d \in D$ with $C(d) \in VSGC$. And what is more, these sets are classes of equivalence: we have generated partition in VSGC such that every component of it represents an infinite amount of diagramas that are not represented in any other component. From a

more rigurous point of view, at this point, the natural representation we have been adjusting through this chapter is:

$$\begin{array}{rccc} \mu: & D & \longrightarrow & VSGC/\simeq_o \\ & d & \longmapsto & C(d) \end{array}$$

Which is clearly surjective but not injective: an infinite amount of diagrams goes to every element of $VSGC/\simeq_o$. However, this won´t be a problem: From proposition 1.2.3, $C(d_1)=C(d_2) \iff \exists gs_1 \in C(d_1), gs_2 \in C(d_2) \ni gs_1 = gs_2$, which let us extend and clarify proposition 1.2.1:

Corollary 1.2.1. Given diagrams d_1, d_2, $C(d_1)=C(d_2) \Rightarrow d_1 \sim d_2$.

Thus, there can be two different diagrams $d_1, d_2 \in D \ni C(d_1)=C(d_2)$, but they will always be equivalent.

To sum up, we are working in $VSGC$, where we have defined an equivalence relation to represent diagrams to begin to see this space as something that will represent knots. Then, our aim is to make a partition in $VSGC$ such that every component of it consists in all the codes that satisfy a certain knot (that is, satisfy some diagram of this knot). It is quite clear that the equivalence relation defined so far is not enough, we have equivalent diagrams whose set of representants are different, as for example two equivalent diagrams that differ somewhere in a single classical Reidemeister move (i), since they will have different number of crossings.

This shouldn't be a surprise since we haven't taken into account generalised Reidemeister moves, which contain the essence of diagram equivalence. We haven't studied how to translate (if possible) these moves in terms of codes, what are the consequences in its corresponding codes when we do a Reidemeister move to a certain diagram.

Although what we want is to relate and put together all the codes that satisfy a certain knot, we will work with a more intuitive approach, we will relate the set of representants of diagrams as we relate diagrams in terms of Reidemeister moves:

Not all generalised Reidemeister moves leave invariant signed Gauss codes (in fact, classical ones does not). If we look at moves in Figure A.7 (b) and (c) is difficult to check that keep invariant signed Gauss codes, nevertheless, all moves in Figure A.7 (a) does not.

Concretely, two different codes like $gs_1 = 1O+2O+2U+1U+$ and $gs_2 = 1O+1U+$ belong to different classes (they have a different number of crossings) but all their realizations are equivalent, in fact we can construct two diagrams d_1 and d_2 from gs_1 and gs_2 respectively using the proof of proposition 1.2.2 and see that d_2 is obtained by aplying to d_1 classical Reidemeister (i). Since every diagram obtained from gs_1 and gs_2 is equivalent to d_1 and d_2 respectively, by proposition 1.2.1 and the fact that this last two are equivalent, every two realization of gs_1 and gs_2 are equivalent.

More rigurously, what we need is to define an equivalence relation in $VSGC/\simeq_o$ such that μ preserves the equivalence relation and the converse also holds, that if two elements in $VSGC/\simeq_o$ are equivalent, then all its anti-images are equivalent.

We will use the same work line used in $(D,\sim)$ and try to define the equivalence relation in terms of moves, relating the sets of representants as do its diagrams in $(D,\sim)$. In particular, it seems natural try to traduce generalised Reidemeister moves to the language of codes and see if they work for our our purposes:

Definition 1.2.7. In the context of codes, we call *generalised Reidemeister move (from $gs_1 \in VGSC$ to $gs_2 \in VSGC$)* any move from gs_1 to $gs_2 \ni \exists\, d_1, d_2$ diagrams satisfying gs_1 and gs_2 respectively $\ni d_1$ is obtained from d_2 by applying a generalised Reidemeister move.

Is important to pay attention in which context we are in every moment (diagrams or codes) to be able to distinguish to what we refer when we say generalised Reidemeister move, since there is an abuse of notation here. Let's consider:

Definition 1.2.8. The following moves are called *basic generalised Reidemeister moves (i),(ii) and (iii)*:

(i):

- $gs_1 = C_1 i a_1 s_1 i(-a_1) s_1 C_1 \longrightarrow gs_2 = C_1 C_2$

(ii):

- (a) $gs_1 = C_1 i a_1 s_1 j a_1 (-s_1) C_2 i(-a_1) s_1 j(-a_1)(-s_1) C_3 \longrightarrow gs_2 = C_1 C_2 C_3$
- (b) $gs_1 = C_1 i a_1 s_1 j a_1 (-s_1) C_2 j(-a_1)(-s_1) i(-a_1) s_1 C_3 \longrightarrow gs_2 = C_1 C_2 C_3$

(iii):

- a) $gs_1 = C_1 i a_1 s_1 j a_1 s_2 C_2 i(-a_1) s_1 k a_2 s_3 C_3 j(-a_1) s_2 k(-a_2) s_3 C_4 \longrightarrow$
  $gs_2 = C_1 j a_1 s_1 i a_1 s_2 C_2 k a_2 s_3 i(-a_1) s_1 C_3 k(-a_2) s_3 j(-a_1) s_2 C_4$
- b) $gs_1 = C_1 i a_1 s_1 j a_1 s_2 C_2 j(-a_1) s_2 k(-a_2) s_3 C_3 i(-a_1) s_1 k a_2 s_3 C_4 \longrightarrow$
  $gs_2 = C_1 j a_1 s_1 i a_1 s_2 C_2 k(-a_2) s_3 j(-a_1) s_2 C_3 k a_2 s_3 i(-a_1) s_1 C_4$

Where i,j,k$\in \mathbb{N}$ and $i \neq j \neq k$, $a_l \in \{O, U\}$, $s_m \in \{O, U\}$,with $l \in \{1,2\}$ and $m \in \{1,2,3\}$ and $C_r \in SGC$ in a way that $gs_1, gs_2 \in VSGC$, for $r \in \{1,2,3,4\}$. Moreover, $O = -U$ and $U = -O$.

In particular, we will say that σ is a basic generalised Reidemeister (i),(ii) or (iii) if it is of the form in (i),(ii) or (iii).

Proposition 1.2.4. In the context of codes, any generalised Reidemeister move is of the form $\rho^{-1} \circ \sigma \circ \rho$, where ρ is a finite composition of strong equivalence moves and σ is a basic Reidemeister move.

Definition 1.2.9. In the context of codes, we say that γ is a *generalised Reidemeister move (i), (ii) or (iii)* if the σ above is a basic Reidemeister move (i), (ii) or (iii).

To sum up, we have characterized all the possible consequences of doing a Reidemeister move to a diagram in terms codes, defining them with the support of basic Reidemeister moves in order to make the notation the most compact as possible, since there a lot of possible code moves. However, this characterization won´t be very helpful: as we will see after, the equivalence relation in $VSGC/\simeq_o$ we are looking for to make our representation work is going to be defined in terms of these moves. However, finding when a code gs_1 can be obtained by applying a finite number of generalised Reidemeister moves to other code gs_2 is going to be, at least, as hard as finding when a diagram d_1 can be obtained by applying a finite number of Reidemeister moves to other diagram d_2, since there exists no other method than brute force.

Nevertheless, now we know that these moves can be characterized and we can understand better the representation we are constructing.

It is simple (but ponderous and technical) to check that, effectively, (code) generalised Reidemeister moves are all the possible consequences in a code when the diagram that it represents is affected by a generalised Reidemeister move. Therefore, we will begin to relate the sets of representants of diagrams under these moves:

Definition 1.2.10. Given d_1,$d_2 \in D$, we say $C(d_1)$ and $C(d_2)$ are equivalent, $C(d_1) \sim C(d_2)$, if $\exists\ gs_1 \in C(d_1)$, $gs_2 \in C(d_2)$ such that gs_2 can be obtained from gs_1 by using a finite number of generalised Reidemeister moves. It is an equivalence relation and the equivalence classes will be denoted by $[\cdot]$.

Note that it can be defined in the same way in terms of elements of $VSGC/\simeq_o$:

Definition 1.2.11. Given gs_1,$gs_2 \in VSGC$, we say $[gs_1]_o$ and $[gs_2]_o$ are equivalent, $[gs_1]_o \sim [gs_2]_o$, if $\exists$ $gs_1^{'} \in [gs_1]$, $gs_2^{'} \in [gs_2]$ such that $gs_2^{'}$ can be obtained from $gs_1^{'}$ by using a finite number of generalised Reidemeister moves. It is clearly an equivalence relation and the equivalence classes will be denoted by $[\cdot]$.

We now have to study if this relation make sense, if $\forall d_1, d_2 \in D$, $C(d_1) \sim C(d_2)$ means that $d_1 \sim d_2$ and vice versa:

First of all, If $d_1 \sim d_2$ we can get d_2 from d_1 by using just Reidemeister moves, what, if we fix $gs_1 \in C(d_1)$, is traduced as obtaining some $gs_2 \in C(d_2)$ by using just (code) Reidemeister moves, and then $C(d_1) \sim C(d_2)$. As an immediate consequence of this one has the following.

Proposition 1.2.5. Given $d_1, d_2 \in D$, $d_1 \sim d_2 \Longrightarrow C(d_1) \sim C(d_2)$.

Proof. □

However, the converse is not that easy, is not trivial that $C(d_1) \sim C(d_2)$ implies $d_1 \sim d_2$, or equivalently, that every diagram move that has as a consequence in its codes a (code) generalised Reidemeister move is a (diagram) generalised Reidemeiser move (note that corollary 1.2.1 provide us a not so trivial necessary condition for the implication to hold). We have to care about if there exist moves in diagrams that are not generalised Reidemeister moves whose consequence in codes is a Reidemeister move. In fact, the answer is yes: diagram move in figure 1.4 is not a generalised Reidemeister move and is traduced in its codes as a Reidemeister move number 1 (2U-2O-1O+1U+$\longrightarrow$1O+1U+ if we start from the arrow).

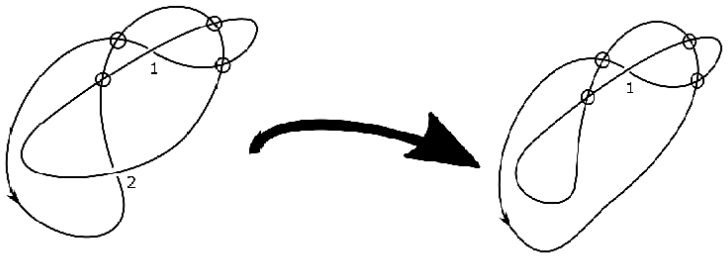


Figure 1.4: Example of non-Reidemeister move with a (code) Reidemeister move as consequence.

Nevertheless, this won´t be a problem:

Lemma 1.2.1. Any (diagram) move whose consequence in a previously fixed code of the starting diagram is a generalised Reidemeister move is composition of a finite number of (diagram) generalised Reidemeister moves.

This means that when we do a generalised Reidemeister move to a code we do not get out from the class of equivalence in its corresponding diagrams, every two realizations of these codes are equivalent:

Proposition 1.2.6. Given $d_1, d_2 \in D$, $C(d_1) \sim C(d_2) \Longrightarrow d_1 \sim d_2$.

Which, together with the fact that the converse also holds (see Proposition 1.2.5), implies:

Proposition 1.2.7. Given d_1, d_2 diagrams, $C(d_1) \sim C(d_2) \Longleftrightarrow d_1 \sim d_2$.

Then, we have finally reached our purpose, we have the representation μ desired: Now every diagram d can be represented by $C(d)$ (or by $[gs]_o$ for some $gs \in C(d)$ in a more abstract sense) which are all related in terms of generalised Reidemeister moves in the same way as their correspoding diagrams. This two notations, even though they are defined from different perspectives, basically represent the same, but the second one will let us talk fully in terms of codes while the first still shows the connections with the world of diagrams.

We can now talk about knots in terms of codes: we have traduced the geometrical definition of knots in the world of diagrams into the algebraic and more computable world of codes. To conclude, we will join together all the codes that satisfy the same knot and make a partition in $VSGC$ such that every component uniquely represents a knot and every knot is represented by some component:

Definition 1.2.12. Given a knot K, we call set of representatives of K, $C(K) \subseteq VSGC$, to $C(K) := \{gs \in VSGC : gs \in \mu(d)$ for some $d \in D(K)\}$, where $D(K)$ is the set of all diagrams of K.

This is the set of all the codes that are satisfied by K. Analogously to diagrams, these sets are classes of equivalence (under the relation of "being satisfied by the same knot") and the relation can be characterized in the following way:

Definition 1.2.13. Given $gs_1, gs_2 \in VSGC$, we say gs_1 is equivalent to gs_2, $gs_1 \sim gs_2$, if gs_2 can be obtained from gs_1 by applying a finite number of generalised Reidemeister moves and/or strong equivalence moves. It is an equivalence relation and the equivalence classes will be denoted by $[\,\cdot\,]$.

Proposition 1.2.8. Given a knot K and $gs \in C(k)$, $C(K) = [gs]$.

As before, the proposition above assures that we can use both notations, the second if we want to talk fully in terms of codes and the first if we want the connection with the world of diagrams to remain. This last concept of set of representatives of a knot conforms the conclusion of this chapter: codes can represent knots. Moreover, the following chapter will be the responsible to show that this representation becomes useful in virtual knot theory.

Chapter 2

Planarity

In the context of virtual knot theory, it seems reasonable to wonder how we can identify when a given virtual knot is classical or purely virtual. Unfortunately, this problem, which we call *the problem of the nature of knots*, exceeds the objectives of this paper.

Nevertheless, if we work with diagrams, it is trivial that no virtual crossings implies its corresponding knot is classical, what is an advantage for this approach. This property is lost when we work with codes: it is not trivial in general to identify the nature of a knot just by watching one of its codes. However, there is a way to identify whether a code is representing a classical knot in certain situations.

In this chapter, making an independent analysis to the different types of information given in a valid signed Gauss code, we will give answer to the so-called *problem of planarity of codes*: determine whether or not a given code has a realization with no virtual crossings. This checking is made via an algorithm and will also give such a diagram should it exist.

Nevertheless, note that we won't be able to identify the nature of the knot from one of its codes, just obtain if, among the diagrams that satistisfy the code there exists at least one planar. If it exists, the knot represented is classical, if not, the knot won't necessarily be non-classical, since there can exist other code equivalent to ours with a planar realization. Then, in comparison with diagrams representation, the fact of working with codes will let us check the planarity of more than one diagram at the same time.

In conclusion, we will close this paper by seeing that, not only valid signed Gauss codes can represent knots, but also they inherit and extend some planarity properties of diagrams working in a more abstract sense: their algebraic nature let us operate with computable algorithms.

2.1 Introduction

Recall that a knot is classical if it can be embedded in $\mathbb{R}^3$, or equivalently, when there exists at least one diagram of the knot that has no virtual crossings. Thus, in order to determine whether the knot represented by a given diagram is classical, we have to find an equivalent planar diagram or prove that there exist no such diagrams. In this paper, the only tool we have to do so is generalised Reidemeister moves, which is not a good idea: in general, finding the way to obtain one diagram from another in terms of these moves turns out to be intractable from a not so big number of crossings.

We know from appendix A that the classification of knots is one of the main problems in this theory. Although is easily proposed, its answer does not seem that easy. The same happens with identifying the nature of a knot: at this point, the problem is reduced to use generalised Reidemeister moves in both cases. However, other techniques have been succesfully developed to give partial solutions to these problems, more advanced techniques that won't be studied in this paper. Gauss codes will let us obtain

an answer in some cases:

Definition 2.1.1. Given $gs \in VSGC$, we will say gs is *planar* if there exists a planar diagram satisfying gs.

In the following sections, we will develop a new approach to the study of codes in order to give a method to determine whether or not a given code is planar. The concept of shadows will come back, simpler objects than diagrams where our results (mainly an algorithm) can be clearerly exposed in a way that they can be extended to the universe of diagrams later. Meanwhile, $gs = 1O+2U+3O+1U+2O+3U+$ will be the example followed to clarify all the exposed.

2.2 Shadows

As said before, we will make an independent analysis to the different types of information given in a code: labels, (O/U) structure and signs. This lets us face the problem of planarity of codes by parts, which simplifies considerably the magnitude of the problem.

First we will study the information provided by the distribution of the labels:

Definition 2.2.1. Given $gs = a_1b_1c_1a_2b_2c_2...a_{2n}b_{2n}c_{2n} \in VSGC$, where $n \in \mathbb{N}$, we call *underlying code of gs*, $u(gs)$, the sequence $u(gs) = a_1a_2...a_{2n}$.

In a more abstract sense:

Definition 2.2.2. We call *underlying code* any sequence $gu = a_1a_2...a_{2n}$ where $n \in \mathbb{N}$ and $a_i \in \{1,2,...,n\}$. We define the *length of the code* as n and the set of all underlying codes is denoted as UGC.

Shadows will play an essential role here. Recall from appendix A that a shadow is just a diagram that have lost the over/under structure of their crossings, are diagrams with intersections instead of crossings. In order to clarify this, some examples are available: in Figure A.5 (a) we have a diagram and the shadow obtained from the diagram (every diagram d has associated with it exactly one shadow by transforming its crossings into intersections denoted by $w(d)$) and more other shadows in Figure A.6 (a) and Figure A.8 (a). Therefore, every shadow can be seen as a diagram without the crossing structure and every diagram as a shadow with a certain crossing (O/U) structure.

Every shadow has a finite number of associated diagrams, depending on the (O/U) structure given to the shadow and every diagram just one shadow. Thus, it is faster to get the shadows of all the planar diagrams satisfying a given code instead of all the planar diagrams that satisfy a code.

On the other hand, underlying codes will be to shadows what signed Gauss codes to diagrams. If we treat shadows like diagrams, when we try to get a signed Gauss code from them, we realize that, although the sequence of labels can be obtained as allways, the (O/U) structure and consequently, the sign sequence, make no sense here:

Definition 2.2.3. We say that a shadow u, with $n \in \mathbb{N}$ (non-virtual) intersections is a *labelled shadow* if every (non-virtual) intersection $i_j \in u$, with $j \in \{1,2,...,n\}$, is labelled as j.

Definition 2.2.4. Let K be a knot, u a labelled shadow of K in which we have fixed as + and - the two different possible orientations, with $n \in \mathbb{N}$ (non-virtual) intersections and a point $P \in u$. We call *underlying code of u based in P with orientation k*, $gu_P^k(u)$, to the sequence $gu_P^k(u) = a_1a_2...a_{2n}$, where $a_i \in \mathbb{N}\ \forall i \in \{1,2,...,2n\}$, obtained in the following way:

- Starting from P, we travel through u following orientation k until we reach P again. Every time we reach a (non-virtual) intersection (in j^{th} position, with $j \in \{1,2,...,2n\}$) we conclude:

* $a_j = r$, where $r \in \{1,2,..,n\}$ is the label of the (non-virtual) intersection reached in j^{th} position.

Since shadows are 4-valent graphs, then the number of appearances of every label in $gu^k_P(u)$ must be two.

In order to clarify this concept, we have an example in Figure 2.1 (a) of an underlying Gauss code of the trefoil shadow. Starting from P, we first face intersection 1 and later intersection 2, obtaining 12. Later on, we encounter intersection 3, and again intersection 1 and 2. The last one is 3, which concludes that $gu^k_P(u) = 123123$.

Then, if we are working with a diagram d and we want to study its corresponing shadow $w(d)$, the underlying codes of $w(d)$ capture its information as do the codes of d with d and, moreover, the underlying code of a code obtained from a diagram is the underlying code of its shadow, read starting in the same point, using the same labelling and with the same orientation used in the diagram reading:

Proposition 2.2.1. Given d a diagram and $w(d)$ its corresponding shadow, $P \in d$ point and k orientation given to d, then $u(gs^k_P(d)) = gu^k_P(w(d))$.

This is why we make this independent analysis: if we first study the distribution of labels of a code (its underlying Gauss code) we will get information of the shadows of the diagrams of the code. As it is a necessary condition for the diagram to be planar that its shadow is planar, we will first obtain the planar shadows satisfying the underlying Gauss code (if exists), give them the (O/U) structure and check if they verify the sign sequence. Therefore, let's study more about underlying codes. Most of the things seen for codes have their analogous for underlying codes, but just some will be required here:

Definition 2.2.5. Given $gu \in UGC$, we say gu is valid if $\exists$ u a shadow satisfying gu. The space of all valid signed Gauss codes is denoted by $VUGC$.

Corollary 2.2.1. Given $gu \in UGC$, gu is valid $\Longleftrightarrow$ every label appears exactly twice.

Proof. Analogous to Proposition 1.2.2. □

Definition 2.2.6. Given $gu \in VUGC$, we say gu is planar if $\exists$ u a shadow satisfying gu with no virtual crossings.

With the above, we have all the necessary preliminaries to expose our study of underlying codes' planarity. The algorithm we will first develop, called *underlying codes' planarity algorithm*, not only tells us if the underlying code we are dealing with is planar, but also gives us a planar realization of it and its understanding will serve to characterize all the possible planar realizations (recall that there are infinite planar realizations of a given planar underlying code, however, they can be (completely) classified in a finite list of shadows under strong equivalence, in other words, there is a finite list of representatives where every realization of the planar underlying code is strongly equivalent to just one of them).

The idea is the following: we will suppose that such a planar realization exists and work with it in an abstract sense. We will perform some deformations that reorganize this realization and bring it to a especial type of shadow that is strongly equivalent to it (and thus, satisfies the same underlying code). All these geometrical performances can be captured in terms of sequences in a way that this special shadow can be reconstructed from them. This reconstruction will be our planar embedding in case that it is planar and, otherwise, the code is not planar.

Let's begin with the algorithm. Note that the underlying code we are dealing with must be valid and that we have given an easy criterion to check so. We will first describe the algorithm itself and after, give its geometrical interpretation. It is a simple method, but the necessity of making a rigurous

description so as to avoid ambiguity can make the lecture difficult. In order to fix so, it will be applied to our standard example $gs = 10+2U+3O+1U+2O+3U+$ while being exposed:

UNDERLYING CODE PLANARITY ALGORITHM:

Input: $gu = a_1a_2...a_{2n} \in VUGC$, with $n \in \mathbb{N}$.
Example: $gu = u(gs) = 123123$.

- 1.- Description: for $i = 1,...,n$, being a_{j_1} and a_{j_2} the first and the second appearance of i in gu respectively, we reverse all the labels between a_{j_1} and a_{j_2}:
  - Rename gu as gu^o and a_j as a_j^o $\forall j \in \{1,2,...,2n\}$.
  - For $i = 1,...,n$:

$$gu^{i-1} = a_1^{i-1}a_2^{i-1}...a_{2n}^{i-1} \longrightarrow gu^i = a_1^i a_2^i...a_{2n}^i$$

$$a_j^{i-1} \longrightarrow a_j^i$$

$$a_j^i = \begin{cases} a_j^{i-1} & if \quad j \leq j_1 \\ a_j^{i-1} & if \quad j_2 \leq j \\ a_{j_2-(j-j_1)}^{i-1} & if \quad j_1 < j < j_2 \end{cases}$$

  - The resulting gu^n is renamed as gu^*.

  Example: $gu = 123123 \longrightarrow 132123 \longrightarrow 132123 \longrightarrow gu_* = 132123$.

- 2.-Description: we have to join the two appearances of every label in gu^* with an arc, either over or under gu^*, in a way that the arcs (of each label joining) does not intersect each other. To do so, we define two sets, OJ for the labels joined over gu^* and UJ for those joined under it and proceed as follows:
  - For $i = 1$:
    $OJ_1 := \{1\}$ and $UJ_1 := \varnothing$.
  - For $i = 2,...,n$:
    If both appearances of i are simultaneous, either between or outside the two appearances of j, $\forall\ j \in OJ_{i-1}$, then $OJ_i = OJ_{i-1} \cup \{i\}$ and $UJ_i = UJ_{i-1}$. Else:
    - If exactly the two appearances of i are, either between or outside the two appearances of j, $\forall\ j \in UJ$, then $UJ_i = UJ_{i-1} \cup \{i\}$ and $OJ_i = OJ_{i-1}$. Else:
      - The algorithm stops because the code is not planar and outputs 'non-planar'.
  - $OJ = OJ_n$ and $UJ = UJ_n$.

  Example: $OJ = \{1\}$ and $UJ = \{2,3\}$.

Output: If the code is planar: gu^*, OJ and UJ. If the code is not planar: 'non-planar'.

GEOMETRICAL INTERPRETATION:

Let us interpret the algorithm and understand why it works. While giving the geometrical explanation in a general sense, the example before will also be followed here in order to be more concrete:

We will suppose that a planar realization u for gs exists (in our example, this realization is in Figure 2.1 (a)). Recall that shadows do not have a particular orientation, labels or point of starting: all the extra drawings in Figure 2.1 (a) to the trefoil shadow just indicate how to read it to get the underlying code

we are dealing with, are not part of the shadow. This is going to happen with shadows, diagrams and other geometrical objects: extra information will be drawn in order to understand what we are doing, depending on the context of the moment.

- Step 1: Here, $\forall\, i \in \{1,2,...,n\}$ we consider $G_i = B(i,\varepsilon_o(i)) \cap u$, where $B(i,\varepsilon_o(i))$ the opened ball of center intersection i and radius a fixed $\varepsilon_o(i)$ such that $\bigcap_{i\in\{1,2,...,n\}} B(i,\varepsilon_o(i)) = \varnothing$ (see Figure 2.1 (b)).

  For $i = 1,...,n$:
  G_i is composed by 4 segments with a common endpoint, the intersection i. These segments will be denoted by iAA, iAB, iBA, and iBB for the first reached when reading gu, the second, the third and the fourth respectively (see again Figure 2.1 (b)).

  For $i = 1,...,n$:
  Note that, when we travel from $1AB$ to $1BA$ following gu, a closed curve (with intersections in the way) is what actually being travelled (see Figure 2.2 (a)). The idea of this step is to do a reconnection (see Figure 2.2) of the segments that changes the orientation of this closed curve, for every intersection. Thus, a kind of "pseudo-shadow" is generated and a new kind of "pseudo-underlying code" is used to capture what is happening:
  We disconnect the two pairs of non-adjacent segements, $iAA - iAB$ and $iBA - iBB$, and reconnect them as the two pairs $S^i_1 - S^i_3$ and $S^i_2 - S^i_4$, where S^i_j is the segment corresponding to i reached in j^{th} position when reading gu^{i-1} (the pseudo-underlying code generated for $i-1$), for $j \in \{1,2,3,4\}$ (in our example, for $i = 1$: $S^1_1 = 1AA$, $S^1_2 = 1AB$, $S^1_3 = 1BA$, and $S^1_4 = 1BB$, see Figure 2.1 (c)). Furthermore, we will put a segment joining the connecting points of the pair (which preserve the labelling i), to indicate that there is an intersection there (intersection i) and be able to undo this process at the end. We call u_i the resulting "pseudo-shadow".
  When the loop is finished, due to the reconnecting choice we do, the resulting u_n is allways a Jordan curve with segments joining the labelled connecting points inside and outside that never cross as follows in Figure 2.1 (d). This is the key fact that let us reconstruct the planar embedding (if it exists).
  On the other hand, this Jordan curve can be deformed to a circunference (with segments inside and outside that never cross) as in Figure 2.1 (e) and this circunference, to Figure 2.1 (f): we put all the labels in the upper part of the circunference and deform it as a straight line containing all the labels (called the axis), whose endpoints are joined by a curve. Note that the distribution of the labels in the axis is gu^*.
  Now, we can undo the reconnections, substituting the labelled segments by labelled intersections (see Figure 2.1 (g)). All we have done to u until here will be called the *canonical shadow algorithm*, since the resulting shadow is of a special type called *canonical shadow*:

  Definition 2.2.7. Given u a planar shadow, we call *canonical shadow of* u, $c(u)$, the result of applying the canonical shadow algorithm to u.

  Definition 2.2.8. Given $gu \in VUGC$ planar, we call *canonical shadow of* gu, to any shadow that is the result of applying the canonical shadow algorithm to u, where u is a planar shadow verifying gu. The set of all canonical shadows of gu is denoted by $C(gu)$.

  Definition 2.2.9. Given u a planar shadow, we say u is a *canonical shadow* if $u = c(u)$.

  Note that this shadow is strongly equivalent to the original one: we have just done a reordering of u, encoding the intersections with segments and performing deformations that do not involve them. So, every planar realization of an underlying code can be brought to its canonical form, and then, canonical forms represent all the planar realizations of a given underlying code.

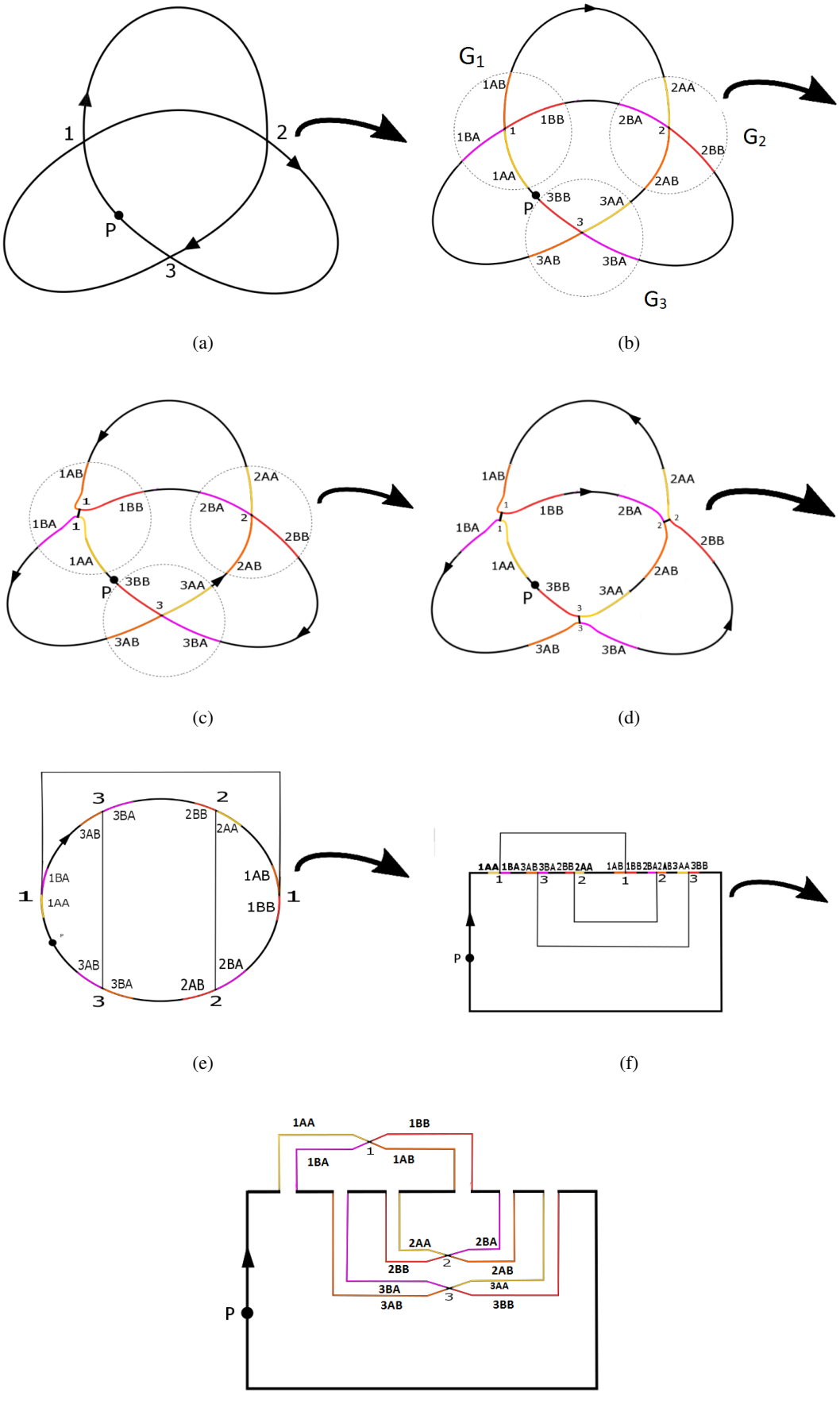
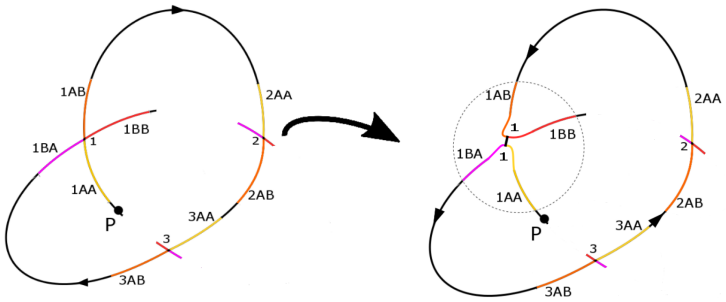


Figure 2.1: Example of underlying code planarity algorithm.



(a) (b)

Figure 2.2: Reconnection scheme.

- Step 2: Actually, every canonical shadow can be encoded as a sequence (of the form of an underlying code) with the appearances of the same label joined by an arc (either over or under the sequence) such that they do not interesect, as represented in Figure 2.3.

  Definition 2.2.10. Given $gu^* \in VUSG$, we call *proper pairing of* gu^* to any way of joining the appearances of the same label by an arc (either over or under the sequence) in a way they do not intersect.

  Then, the problem of finding all the canonical shadows of gu is equivalent to finding all the proper pairings of gu^*. We can see in Figure 2.3 again all the canonical shadows of $gu = 123123$.

  Unfortunately, finding an algorithm in polynomial time that gives all possible pairing does not seem an easy task. However, this step identifies if there exists at least one proper paring and gives it in affirmative case:

  For $i = 1, ..., n$:
  We pair 1 over gu^*.

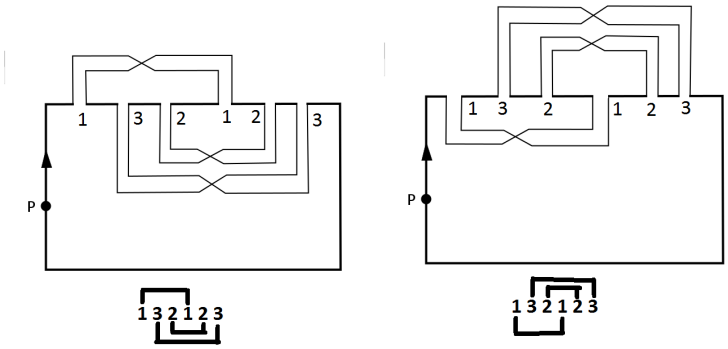
  For $i = 1, ..., n$:
  We check whether or not we can pair i over gu^*, that is, whether or not it intersects with one of the previously paired labels over gu^* and do it in the affirmative case. If it is not possible, we do the same under gu^*. If it is not possible again, there exists no proper pairing and the algorithm stops and outputs:'non-planar', since we have found three labels such that every two of them cannot be paired in the same part (over or under gu^*). Note that if the algorithm does not stop we have found a proper pairing.

  Output:

  * If the code is planar, gu^* and a proper pairing (for $gu = 123123$: $gu^* = 132123$ and $OJ = \{1\}$ and $UJ = \{2,3\}$), what can reconstruct a canonical shadow with an axis determined by gu^* and the position of the intersections by the proper pairing.
  * If the code is not planar, the message: 'non-planar'.

  To conclude, we will review that the algorithm really works:

  Proposition 2.2.2. Given $gu \in VUGC$, gu planar $\Longleftrightarrow$ the underlying code planarity algorithm gives a planar shadow.



(a) (b)

Figure 2.3: All canonical shadows of 123123.

Proof. $\Rightarrow$) Let $gu \in VUGC$ and suppose that the algorithm stops (find that there is no proper pairing for gu^*). Since gu is planar, there exists a planar realization of gu, say u, and if we encode $c(u)$ as gu^* and the induced pairing, we have found a proper pairing of gu^* (since $c(u)$ is planar), a contradiction.
$\Leftarrow$) Trivial. □

2.3 Code Planarity Algorithm

Recall that our main aim is to identify if a given code is planar and give a planar realization in this case. In this direction, the underlying code planarity algorithm exposes some ideas that we are going to use here and clarify our knowlege about shadows and diagrams, but that does not suffice our purposes: to check that a code is planar, it would be enough to obtain all the planar shadows of its underlying code, give them the (O/U) structure and see if the diagram obtained verify the sign sequence. However, we do not know any algorithm in polynomial time to obtain all the planar shadows of an underlying Gauss code.

Fortunately, an extension of the shadow planarity algorithm for the signed Gauss code case identify if the code is planar and gives one of its planar realizations in the affirmative case, whose shadow is canonical:

Definition 2.3.1. Given a diagram d, we say d is a *canonical diagram* if $w(d)$ is a canonical shadow.

In addition, the canonical diagram satisfying a code turns out to be unique: the realization provided by the algorithm is the only canonical diagram satisfying the code, that is, every planar realization of it is strongly equivalent to the one obtained.

Definition 2.3.2. Given $gs \in VSGC$ planar, we call the canonical diagram satisfying gs its *canonical diagram*.

Definition 2.3.3. Given d a diagram and $gs \in VSGC$ a code satified by d, we call *canonical diagram of* d, $c(d)$, to the canonical diagram that verifies gs.

Having all these preliminaries in mind, let us develop the method and see in more detail all the anticipated above. Since it is an extension to the other algorithm and shares the same notation, the explanation won't be as extense as before.

It is important to recall that we have given a simple criterion to determine when a signed Gauss code is valid. Consequently, in order to simplify the algorithm, we will require as its input a (valid signed Gauss) code. Furthermore, as it is a necessary condition for the code to have a planar realization that its underlying Gauss code has a planar realization, it is a good idea to check so by using the underlying code planarity algorithm. Although not necessary since the algorithm itself will give us a more complete answer, it can save us time.

As always, we will expose the abstract description of the algorithm and after, give its geometrical explanation, following our example $gs = 1O+2U+3O+1U+2O+3U+$.

CODE PLANARITY ALGORITHM:

Input: $gs = a_1b_1c_1a_2b_2c_2...a_{2n}b_{2n}c_{2n} \in VSGC$, with $n \in \mathbb{N}$.
Example: $gs = 10+2U+3O+1U+2O+3U+$.

- 1.- Description: we take $u(gs) = gu = a_1a_2...a_{2n}$.

  Example: $gu = 123123$.

- 2.- Description: substitute in gu the first appearance of i, a_{j_1}, by $A_{2j_1-1}A_{2j_1}$, where $A_{2j_1-1} = iAA$ and $A_{2j_1} = iAB$, and the second, a_{j_2}, by $A_{2j_2-1}A_{2j_2}$, where $A_{2j_2-1} = iAA$ and $A_{2j_2} = iAB$, for $i \in \{1,2,...,n\}$ and where $1 \leq j_1 < j_2 \leq 2n$. Then, $GU = A_1A_2...A_{4n}$ and we call *block* to every A_r, with $r \in \{1,2,...,4n\}$:

  - $gu = a_1a_2...a_{2n} \longrightarrow GU = A_1A_2...A_{2n}$.

  Example: $gu = 123123 \longrightarrow GU = 1AA1AB2AA2AB3AA3AB1BA1BB2BA2BB3BA3BB$ and for example, the block $A_3 = 2AA$ and $A_{10} = 2BB$.

- 3.- Description: Rename GU as GU^o. For $i = 1,...,n$: reverse all the blocks between the first and the last block corresponding to the label i in GU^{i-1}, where GU^i is the sequence obtained in step i:

  - Rename A_j as A_j^o and GU as GU^o.
  - For $i = 1,...,n$:

$$GU^{i-1} = A_1^{i-1}A_2^{i-1}...A_{4n}^{i-1} \longrightarrow GU^i = A_1^iA_2^i...A_{4n}^i$$

$$A_j^{i-1} \longrightarrow A_j^i$$

$$A_j^i = \begin{cases} A_j^{i-1} & if \quad j \leqslant 2j_1 - 1\, j \\ A_j^{i-1} & if \quad 2j_2 \leq j \\ A_{2j_2-(j-(2j_1-1))}^{i-1} & if \quad 2j_1 \leq j \leq (2j_2-1) \end{cases}$$

The resulting GU^n is renamed as GU_*.

Example:

$$GU = 1AA1AB2AA2AB3AA3AB1BA1BB2BA2BB3BA3BB \longrightarrow$$
$$1AA1BA3AB3AA2AB2AA1AB1BB2BA2BB3BA3BB \longrightarrow$$
$$1AA1BA3AB3AA2AB2BA1BB1AB2AA2BB3BA3BB \longrightarrow$$
$$GU_* = 1AA1BA3AB3BA2BB2AA1AB1BB2BA2AB3AA3BB$$

- 4.- Description: for $i = 1,...,n$: capture in a 6-component vector v_i:

  - The information of the distribution of the appearances of i in GU_*: in component number j of v_i we write the two letters followed by the j^{th} appearance of i in GU^*, with $j \in \{1,2,3,4\}$.
  - The (O/U) information of i in gs: the 5^{th} component of v_i is, O if the first appearance of i in gs is followed by O and the second by U, or U otherwise.
  - The sign information of i in gs: the 6^{th} component of v_i is, $+$ if the sign associated to i in gs is $+$, or $-$ otherwise.

  Example: $v_1 = (AA, BA, AB, BB, O, +)$, $v_2 = (BB, AA, BA, AB, U, +)$, and $v_3 = (AB, BA, AA, BB, O, +)$.

- 5.- In Figure B.1 we have a table that gives an output, 'o' (over) or 'u' (under), for every possibility of the vectors v_i, with $i \in \{1,2,...,n\}$: save the output of v_i as p_i, $\forall\, i \in \{1,2,...,n\}$.
  Example: $p_1 = o$, $p_2 = u$ and $p_3 = u$.

- 7.- Generate two sets, the set of labels whose output is 'o' and the set of those whose output has been 'u': $OJ := \{i \in \{1,2,...,n\} | p_i = u\}$ and $UJ := \{i \in \{1,2,...,n\} | p_i = d\}$.
  Example: $OJ = \{1\}$ and $UJ = \{2,3\}$.

- 8.-Obtain gu^*:

  - $\forall i \in OJ$ check that, between the two appearances of i in gu^*, there is, either two appearances of j or none, $\forall j \in U \setminus \{i\}$.
  - $\forall i \in UJ$ check that, between the two appearances of i in gu^*, there is, either two appearances of j or none, $\forall j \in D \setminus \{i\}$.

  If this is not the case, then the algorithm stops because there exists no planar realization of gs and outputs:'non-planar', if not we give:

Output: GU_*, OJ, and UJ, which will determine the embedding.

GEOMETRICAL INTERPRETATION:

As said before, this is just an extension of the underlying code planarity algorithm for the diagram case. Following its work line, we will begin by supposing that there exists a planar realization d for $gs \in VSGC$ and bring it to its (unique) canonical diagram $c(d)$. For $gs = 1O + 2U + 3O + 1U + 2O + 3U +$ the diagram d followed will be Figure 1.1.

- Step 1: we obtain its corresponding shadow, $w(d) = u$ (see Figure 2.1 (a)). From now until almost the end, the geometrical process will be exactly the same as in the underlying code planarity algorithm.

- Step 2: we label the four segments corresponding to label i with iAA, iAB, iBA, and iBB $\forall i \in \{1,2,...,n\}$ as in the first step of the underlying code planarity algorithm (see Figure 2.1 (b)), with the difference that we now capture the information of the distribution of these segments instead of the distribution of the labels (GU), which gives us more information: when we get the canonical shadow, this will let us reconstruct the orientation, since iAA goes to iBB (and iBA to iBB) in the original diagram $\forall i \in \{1,2,...,n\}$. The reason that we haven't done so in the other algorithm is that orientation was not important, however, when we give the (O/U) structure back to the resulting canonical shadow, the sign sequence must be verified, what makes orientation an essential part.

- Step 3: For $i = 1, ..., n$:
  We do the segment reconnection done in step one of the underlying code planarity algorithm to i (forming u_i), reversing the orientation of the closed curve (with intersections in the way) explained before (see Figure 2.2), whose starting and endpoint is intersection i and travel through the result to read the new sequence GU^i (see Figure 2.1 (c) and (d)).

  As before, we bring the resultant labelled Jordan curve (with segments joinning the connecting points) u_n to a circunference and after, to the canonical shadow undoing the reconnections, as in Figure 2.1 (e), (f) and (g). Trivially, if we give the (O/U) structure back to our example the sign sequence is satisfied, as we have just done a reordering of a initial planar diagram we knew that satisfies the code and we have obtained the (unique) diagram that satisfies gs (see Figure 2.4).

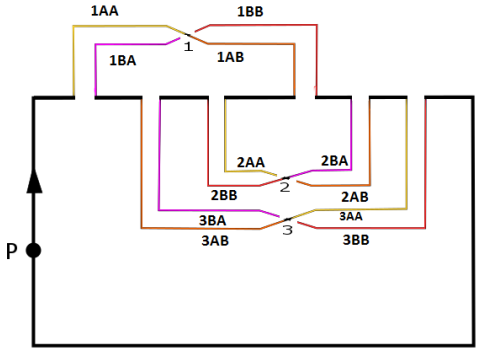


Figure 2.4: Canonical diagram of the trefoil $1O+2U+3O+1U+2O+3U+$

- Steps 4,5 and 6: However, remember that the only information we obtain having gs is GU_*, the (O/U) structure and the sign sequence. The pairing has being obtained due to the fact that we have used a known planar realization of gs in order to explain the algorithm, what is supposed to never happen. Let's care now about obtaining the pairing:

  There exists a simple method that allows us to obtain the position of every crossing indepently: given the label $i \in \{1, 2, ..., n\}$, the distribution of the blocks corresponding to i in GU_* (or the segments corresponding to i in the axis (see Figure 2.1 (f))), the (O/U) structure and the sign associated determines if the crossing must be placed over or under the axis (if the appearances of i must be joined over or under gu^*). Note that this proves that there exists a unique canonical diagram of gs: GU_* (uniquely determined by gs), the (O/U) structure and the sign sequence determine the unique possible position of each label in its canonical shadow if it exists.

  For example, let's obtain the position of crossing 1: we know that every intersection in a canonical shadow will be of the form in Figure 2.5 (a). We will use all the information in v_1=(AA,BA,AB,BB,O-U,+) to get the real form of the crossing:

  - The appearances distribution will provide us the local orientations of the crossing. In the case of crossing 1, the distribution in

    $$GU_* = 1AA1BA3AB3BA2BB2AA1AB1BB2BA2AB3AA3BB$$

    is

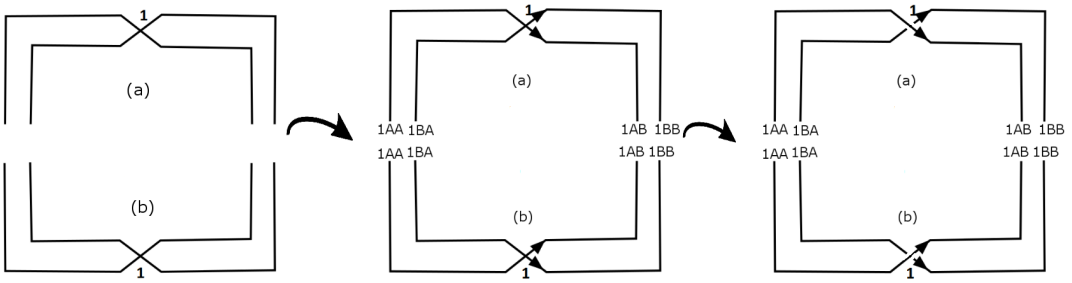
    $$1AA1BA...1BA1BB \longmapsto (1AA, 1BA, 1BA, 1BB).$$

Since $1AA$ goes to $1AB$ and $1BA$ to $1BB$, the orientations must be as in Figure 2.5 (b).

Due to the construction of the algorithm, only some distributions will appear. All these possibilities are collected in table Figure B.1.

- The (O/U) information transforms the intersections in crossings. Therefore, if the first appearance of a certain label $i \in \{1,2,...,n\}$ in gs is associated to an O (and the second a U), which is codified as O, then the segment $iAAiAB$ must overcross $iBAiBB$, on the contrary, codified as U, undercross it. For crossing 1, the first appearance has associated an O, so the information is codified as O and then, $1AA1AB$ overcrosses $1BA1BB$, as seen in Figure 2.5(c).
- If we look again at Figure 2.5 (c), we have the only 2 options for crossing 1, the first with sign $+$ and the second with $-$. This is what allways happens, given all the conditions but the sign information, there are two possibilities for the crossing, over and under the axis, that have a different sign. As we need the crossing to be $+$, we conclude that crossing 1 must be placed over the axis.

Then, working analogously, crossing 2 and crossing 3, we must obtain that both must be placed under the axis, and consequently we get the unique canonical realization of $1O+2U+3O+1U+2O+3U+$ in Figure 2.4. In order to avoid drawing the crossings satisyfing the conditions to get



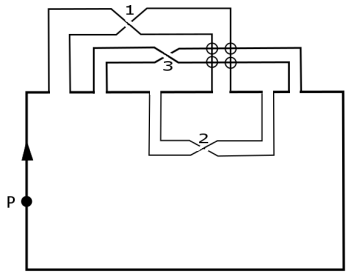
(a) (b) (c)

Figure 2.5: Determination of position of crossing 1.

its position and make this last a computable step, a table in Figure B.1 have been created: when you input vector v_i with all the information, outputs an 'o' if the crossing must be placed over the axis and a 'u', otherwise.

- Step 7: This last step checks if the pairing is proper or not. Doing the above with all the labels, we obtain the only possible pairing of GU^* and then, the only possible canonical diagram of gs, completely determined by gs itself. However, the pairing won't be necessarily proper. In the example followed here (see Figure 2.4) the pairing is proper, and thus we have a planar realization and the code is planar, however, this is not the case for gs=1O+2U+3O-1U+2O+3U-, where crossings 2 and 3 must be placed over the axis to satisfy the code, but they intersect. We obtain the canonical diagram in Figure 2.6, where the intersections generated must be seen as virtual crossings in order to satisfy the code: our realization is not planar.

This means that the code is not planar, since if it exists a planar realization, it would be brought (with the algorithm) to its planar canonical diagram by using the algorithm, but no planar canonical diagram has been obtained and no planar realization exists. This concludes that the code is planar if and only if the algorithm outputs the (unique) canonical diagram. Note that we have identified that $1O+2U+3O+1U+2O+3U+$ is planar and $1O+2U+3O-1U+2O+3U-$



(a)

Figure 2.6: Non-planar realization of non-planar code $1O+2U+3O-1U+2O+3U-$.

is not, what implies that the codes are not equivalent, or what is the same, any of their realizations are not equivalent. This proves what advanced in chapter 1, that the diagrams in Figure 1.1 (a) and (b) are not equivalent, something impossible to check with the given about diagrams.

In conclusion, the algorithm above, by facing the different information given in the code indepently, allows us obtain a planar realization of a code if it exists and identify that is not planar in case that does not exist.

Before concluding this section it is important to know that, since shadows are simply 4-valent graphs, we could have used well-known results in Graph Theory about planarity (if a given graph $G = (V,E)$ has a planar realization or not) to identify if a code is a planar and give a realization in affirmative case: however, we have opted to work in a line according to the idiosyncrasy of Knot Theory.

2.4 Conclusion

This last algorithm about planarity closes this brief exposition about virtual knot theory:

To sum up, we have seen a generalization of knot theory that turn out to fit very well with certain approaches: Gauss codes, the relation with graph theory... and some others that we haven't studied. Due to this, it represents a promising line of research inside the well-stablished knot theory, essential in topology and geometry and that contributes in a wide range of disciplines, within Mathematics and other sciences.

Appendix A

Preliminaries

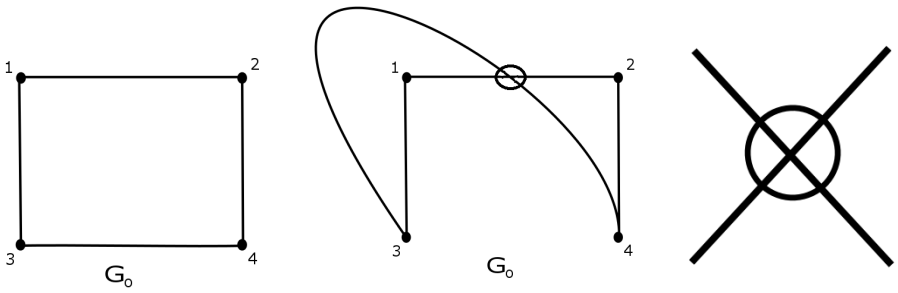
This appendix consists in a brief summary of the basics in Knot Theory and in particular in what is the central topic of the paper, Virtual Knot Theory, in order to gain the necessary background to follow all the exposed.

A.1 Graph Theory

Before entering in the subject and due to the fact that there is a strong relation and common core between knots and graphs, some concepts of these last will be reviewed. *Graph Theory* consists in the study of certain geometrical objects called *graphs*:

Definition A.1.1. We call *abstract graph* any set $G = (V, E)$, where V is a finite set whose elements are called *vertices* and E a multiset whose elements, called *edges*, are sets of two vertices.

These sets are usually represented in $\mathbb{R}^2$, where the vertices are drawn as (different) points and the edges as segments joining its two vertices. We call any of these representations *graphs* and *vertices and egdes of the graph* to the image of the vertices and the edges in the representation. An example of a graph is given in Figure A.1 (a): we have the abstract graph $G_o = (V_o, E_o)$ with $V_o = \{1,2,3,4\}$ and $E_o = \{\{1,2\},\{2,3\},\{3,4\},\{4,1\}\}$ represented in the plane, a graph of G_o.



(a) Planar graph of G_o. (b) Non-planar graph of G_o. (c) Virtual crossing.

Figure A.1: Graphs of G_o and virtual intersection/crossing.

Definition A.1.2. Given an abstract graph $G = (V, E)$, we say $e \in E$ is incident to $v \in V$ if $v \in e$ (if v is one of the endpoints of e in any of its graphs). We denote as $E(v)$ the set of egdes incident to v.

Definition A.1.3. Given a graph $G = (V, E)$ and $n \in \mathbb{N}$, we say G is a *n-valent abstract graph* if $|E(v)| = n \ \forall v \in V$. Any of its graphs are said to be *n-valent graphs*.

On the other hand, in Graph Theory it is very important the concept of planarity:

Definition A.1.4. Given a graph g, we say g is *planar* if the edges do not intersect outside of the vertices.

As we can see, the graph in Figure A.1 (a) is a planar graph, but in Figure A.1 (b) we have a non-planar graph of G_o. The intersection of the edges (outside of the vertices) is represented as in Figure A.1 (c) in order to differenciate it from the vertices. We call this kind of intersections *virtual crossings*.

Definition A.1.5. Given an abstract graph G, we say G is *planar* if there exists a planar graph g of G.

The abstract graph G_o of our example is planar because there exists at least one planar graph of G_o, the given firstly, but not all abstract graphs have a planar graph. We will see that, in the context of Virtual Knot Theory, these concepts serve to clarify and help to understand the complex universe of virtual knots.

A.2 Knot Theory

From the Stone Age to our recent days, knots has been a key element in the development of mankind. Mainly used as tools in devices, constructions and more, its particular structure make them essential for certain specific purposes. Due to this particular geometry, they constitute an object of analysis and study in some disciplines.

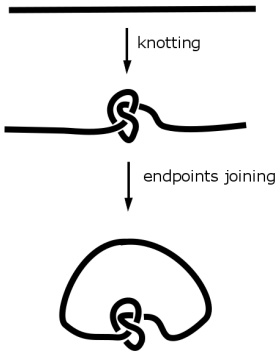


Figure A.2: Description of mathematical knot.

In mathematics, an idealization of what we understand as knot conforms a relatively young branch of topology: routhly speaking, a mathematical knot is the object resultant of knotting a given rope in whatever way and joinning its endpoints, forming a kind of "intertwined circunference" in the 3-dimensional space as represented in Figure A.2. In a more rigurous sense:

Definition A.2.1. Given two topological spaces A and B and an application between them $h : A \longrightarrow B$, we say h is a *homeomorphism* if it is bijective, continuous and h^{-1} is continuous as well.

Definition A.2.2. Given $K : S^1 \longrightarrow \mathbb{R}^3$ map, we say K is a *knot* if it is an embedding (in a topological context), that is, if the map $g : S^1 \longrightarrow K(S^1)$ such that $K = i_{K(S^1)} \circ g$ is a homeomorphism, where $i_{K(S^1)} : K(S^1) \longrightarrow \mathbb{R}^3$ is the inclusion and $K(S^1)$ inherits the topology of R^3.

This topological structure frequently appear describing certain natural geometrical phenomena. Their study is fundamental within Topology and Geometry (where knots appear as selfintersections and boundaries of surfaces as well as in the study of 3-manifolds among other things) and become useful in research in Physics and Biochemistry.

The branch that takes care of understanding them is the so-called *Knot Theory*. Although the first investigations associated to knots took place in the late XVIII Century by the hands of C.F. Gauss or A.

Vandermonde and the (erroneous) atomic model based in knots of Lord Kelvin in the 1860*s* increased the interest in them, they stayed in a secondary position until the beginning of XX Century, in the apogee of topology, with M. Dehn and J.W. Alexander.

Knot Theory has experienced a revolutionary development from then, been faced from many different aproaches from what, among all, we will be particularly interested in the combinatorial one represented by L.H. Kauffman or V. Jones. However, the natural context to define knots is from topology: coming back to the example of the rope, once we have this intertwined circunference, we do not distinguish between the same knot with different sizes, the same knot but being rotated 90 degrees or, in a more general sense, two knots obtained by deforming one to the other with any move that does not cut the knot. In Figure A.3 we have an example of a process in which we get equivalent knots all the time using this type of transformations (notice that we get S^1 (the trivial knot) from other knot that seems more complex).

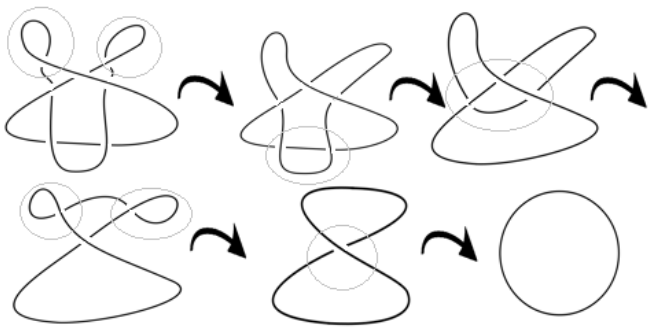


Figure A.3: Equivalent moves.

These moves are captured by a particular map called *isotopy*, a deformation that depends on time and brings one knot to another:

Definition A.2.3. Given two topological spaces X, Y and two embeddings $h_1, h_2 : X \longrightarrow Y$, we call *isotopy from* h_1 *to* h_2 any map

$$\begin{array}{rccc} H: & X \times I & \longrightarrow & Y \\ & (x,t) & \longmapsto & H(x,t) \end{array}$$

such that $H_t(x) = H(x,t)$ is an embedding $\forall t \in I$, $H(x,0) = h_0$ and $H(x,1) = h_1$.

These maps capture the essence of knot equivalence:

Definition A.2.4. Given two topological spaces X, Y and two embeddings $h_1, h_2 : X \longrightarrow Y$, we say h_1 *isotopic to* h_2, $h_1 \sim h_2$, if there exists an isotopy from h_1 to h_2. This is an equivalence relation.

Definition A.2.5. Given two knots K_1 and K_2, we say they are *equivalent*, $K_1 \simeq K_2$, if $K_1 \sim K_2$.

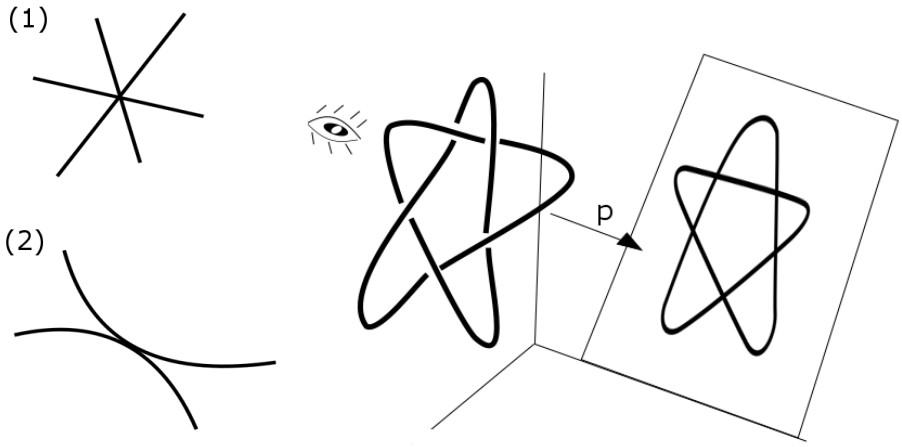
Therefore, what we are actually interested in is these classes of equivalence of knots under isotopy, since englobe knots that are essentially the same for us, knots that can be obtained one from the other by moves that do not cut the knot. All these definitions form the basis of Knot Theory.

However, they just characterize these objects, but in the practice is very hard to find an isotopy between two knots or define knots in terms of embeddings of S^1. Fortunately, we will work with a simpler approach, the universe of diagrams: a diagram is just a representation in the plane of a knot (see Figure A.6 (b)). Before giving a rigurous definition we need to introduce the concept of shadow:

Definition A.2.6. Given a knot K, we call *shadow* of K any closed curve (with or without selfintersections) in the plane that is the resulting of projecting (call p the projection) K into A, where A is some plane in R^3, satisfying:

- There exists no n-points for $n > 2$, where an *n-point* is any point $P \in p(K)$ with n preimages in the projection (see Figure A.4 (a) (1)).
- For every 2-point, the arcs involved intersect transversally (see Figure A.4 (a) (2)).

We call *intersection* of the shadow any 2-point.

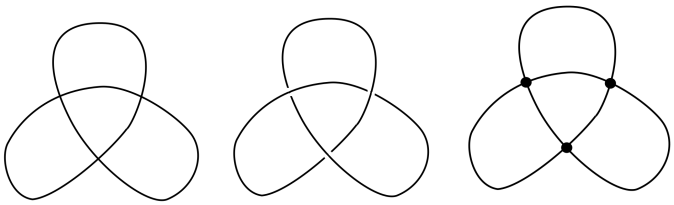


(a) Forbidden situations in conditions 1 and 2 respectively.

(b) Projection of a knot.

Figure A.4: Shadow concepts.

In order to clarify this, we have a projection in Figure A.4 (b), the shadow of the so-called trefoil knot in Figure A.5 (a) and some other shadows in Figure A.6 (a). However, they do not capture all the information: what we want is to represent in the plane a closed curve in $\mathbb{R}^3$. As it is an intertwined curve, it may have intersections when projected in a plane and we have to distinguish between its two preimages. The way to do so is very intuitive, we draw the complete arc that overcrosses and cut in the intersection the one that undercrosses, as we would see it in front of the plane, represented in Figure A.5 (b) and Figure A.6 (b) . Then, intersections are transfromed in what we call *crossings* and every diagram can be seen as a 4-valent graph by substituting crossings by vertices, as in Figure A.5 (c).



(a) Trefoil shadow.

(b) Trefoil diagram.

(c) 4-valent graph of a trefoil diagram.

Figure A.5: Trefoil knot: shadow, diagram and 4-valent graph.

Thus, all the possible diagrams of a given knot represent the knot and from now on, they will be used to develop our study. The next step will be to translate the concept of isotopy to the universe of diagrams.

To do so, we will need to relate the diagrams that come form the same knot.

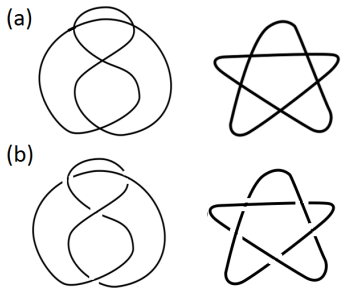


Figure A.6: Examples of shadows and diagrams

Definition A.2.7. Given two diagrams d_1 and d_2, we call *move from d_1 to d_2* any transformation that brings d_1 to d_2.

Equivalence in diagrams will be defined in terms of moves. We will need to be able to determine which moves are allowed and which are not, which moves preserve equivalence in the knots they are representing and which does not. As a necessary condition, it is clear that diagramas that are the same but one is bigger than the other are essentially the same, or that rotations are allowed moves, or in a more general sense, moves that does not erase or generate new crossings, that are just are reordering of the ones we have. These trivial cases are captured by isotopy in the context of graphs: we will see diagrams as graphs (as in Figure A.5 (c)) and define these trivial moves in terms of deformations of these graphs.

Definition A.2.8. Given a planar graph $g_o \subseteq \mathbb{R}^2$, we call *graph embedding (for g_o)* any map $h : g \longrightarrow \mathbb{R}^2$ which is an embedding (in a topological context) and $(h(g) = g_o)$, that is, if the map $f : g \longrightarrow h(g)$ such that $h = i_{h(g))} \circ f$ is a homeomorphism, where $i_{h(g))} : h(g) \longrightarrow \mathbb{R}^3$ is the inclusion and g and $h(g)$ inherits the topology of $\mathbb{R}^2$, and $(h(g) = g_o)$.

Definition A.2.9. Given two planar graphs $g_1, g_2 \subseteq \mathbb{R}^2$, we say g_1 and g_2 are *strongly equivalent*, $g_1 \simeq g_2$, if there exists h_1 a graph embedding for g_1 and h_2 a graph embedding for g_2 that are isotopic $(h_1 \sim h_2)$.

In other words, two graphs are strongly equivalent if one can be brought to the other in a way that its topological structure is preserved. Let´s translate it to diagrams:

Definition A.2.10. We say two diagrams d_1 and d_2 are *strongly equivalent*, $d_1 \simeq_o d_2$, if, seen as graphs, they are strongly equivalent. We call the move from d_1 to d_2 *strong equivalence move*.

In conclusion, given d_1 and d_2 strongly equivalent diagrams, the move that brings d_1 to d_2 is one of these moves that trivially preserves equivalence in its knots, d_1 and d_2 trivially represents two equivalent knots.

However, there are more moves that preserves equivalence in its knots that are not that trivial, as for example, each move involved in Figure A.3. If we pay attention, in each step of the process, the two diagrams involved are equal except from inside the balls drawn. We will define all the moves that preserve equivalence as composition of 3 moves (and its inverses) that leave invariant the diagram except from a local part of it:

Definition A.2.11. We call *Reidemeister move* (i), (ii) or (iii) to any move that leaves invariant the starting diagram except from a local part, where the diagram is transformed as represented in Figure A.7 (a) (i), (ii) or (iii) respectively.

These diagrams form the true basis of diagram equivalence:

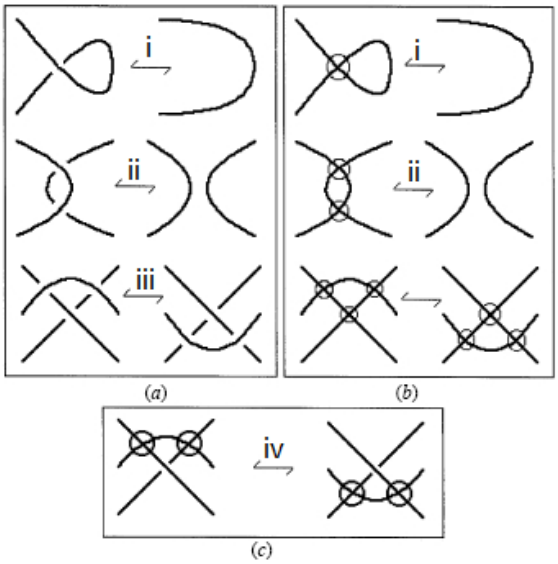


Figure A.7: Generalized Reidemeister moves ([1, page 665]).

Definition A.2.12. Given diagrams d_1 and d_2, we say d_1 is *equivalent* to d_2, $d_1 \simeq d_2$, if d_2 can be obtained from d_1 by using a finite number of Reidemeister moves and/or strong equivalence moves. This is an equivalence relation.

To sum up, strong equivalence moves and Reidemeister moves capture in the universe of diagrams the equivalence under isotopy of the knots they represent:

Proposition A.2.1. Given d_1, d_2 diagrams of K_1, K_2 respectively, $K_1 \simeq K_2 \iff d_1 \simeq d_2$.

Proof. See [3]. □

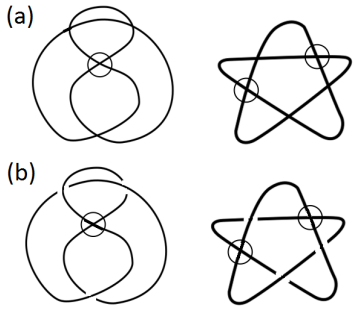
We have defined the concept of knot and equivalence of knots in terms of diagrams in order to have a simpler context to work with knots, but also combinatorial: we have some objects related to each other, related by 3 differents moves. Is not easy to get the way to obtain one from the other, since there exists an infinite number of possiblities to begin with one and continue with others.

This turns the problem of giving a complete classification of knots a hard task: however, partial solutions have been given using more advanced techniques. Following this line, we will continue extending the given before for Virtual Knot Theory, which is the central topic of this paper. So, from now, we will refer to knots as *classical knots* and the concept of knot will be extended.

A.3 Virtual Knot Theory

Discovered in 1996 by L.H. Kauffman, *virtual knots* conform a generalization of the concept of classical knots. In the same way that we have used diagrams to represent classical knots and Reidemeister moves to characterize their equivalence relation, we will identify virtual knots with a new type of diagrams and define a set of moves that characterize its equivalences.

These new diagrams are a generalization of the concept of diagram: being diagrams are a representation of a closed curve of R^3 in the plane that admits objects called *crosings* instead of intersections, we will permit a new type of crossing (Figure A.1 (c)) (called *virtual crossing*), resulting objects as in Figure A.8 (b).



(a) Trefoil shadow.

Figure A.8: Virtual shadows and diagrams.

The idea of virtual crossings is that actually there is no crossing there: although we will define virtual knots in terms of (these new) diagrams, they can be also defined from a topological perspective, as embeddings of S^1 in complex topological spaces. The fact that they do not live in $\mathbb{R}^3$ (as classical knots do) provokes that, when we try to represent them in the plane certain artifacts appear, virtual crossings. In this line, virtual crossings are not real crossings, just a product of the representation in $\mathbb{R}^2$.

Thus, we will redefine and extend the concept of shadow and diagram in order to present virtual knots:

Definition A.3.1. From now, we call *shadow* any 4-valent graph.

This extends the definition given before, and we say a shadow is *planar* if it is a planar 4-valent graph (see Figure A.5 (a)), that is, does not contain virtual intersections, and *non-planar* in other case (see Figure A.8 (a)).

Definition A.3.2. We call *diagram* any shadow with an extra structure in its (non-virtual) intersections as in Figure A.6 (b). In this context, we call *crossing* this resulting intersections and *virtual crossings* to virtual intersections. Moreover, we denote as D the set of all diagrams.

In the same line, we say a diagram is *planar* if it does not contain virtual crossings (see Figure A.5 (b)) and *non-planar* in other case (see Figure A.8 (b)). Notice that these redefinitions effectively extends the concept of shadows and diagrams given for Knot Theory.

The basis of diagrams equivalence will be given as before, in terms of moves: strong equivalence moves and a generalization of Reidemeister moves:

Definition A.3.3. Given a diagram d, we call *planar graphof d*, $p(d)$, to the graph resulting of substituting its crossings and its virtual crossings by vertices.

We redefine the concept of strong equivalence diagrams:

Definition A.3.4. Given two diagrams d_1 and d_2, we say d_1 and d_2 are *strongly equivalent*, $d_1 \simeq_o d_2$, if $p(d_1)$ and $p(d_1)$ are strongly equivalent ($p(d_1) \simeq p(d_2)$).

And generalize Reidemeister moves:

Definition A.3.5. We call *virtual Reidemeister move* (i), (ii), (iii) or (iv) to any move that leaves invariant the starting diagram except from a local part, where the diagram is transformed as represented in Figure A.7 (b) (i), (ii) or (iii) or (c) (iv) respectively. We call *generalized Reidemeister move* to any move that leaves invariant the starting diagram except from a local part, where the diagram is transformed as represented in any of the moves in Figure A.7. From now we call *classical Reidemeister moves (i),(ii) and (iii)* to Reidemeister moves (i), (ii) and (iii).

All these moves form the basis of equivalence in diagrams:

Definition A.3.6. Given diagrams d_1 and d_2, we say d_1 is *equivalent* to d_2, $d_1 \simeq d_2$, if d_2 can be obtained from d_1 by using a finite number of generalized Reidemeister moves and/or strongly equivalence moves. This is an equivalence relation and the classes of equivalence are denoted by $[\cdot]$.

Definition A.3.7. We call *virtual knot* any of these clases.

The concept of virtual knot generalise the concept of classical knot and from now, they will be the object of our study, so, when we say knot we refer to virtual knot. Due to many facts, Virtual Knot Theory represents a promising line of investigation in Knot Theory, which is the main motivation of this paper. After these preliminaries, we will study a specific part of this branch, the capability of certain algebraic elements to represent virtual knots and give us a new interesting and computable perspective to study these particular objects.

Appendix B

Algorithm table

INPUT						OUTPUT
AA	BA	AB	BB	O	+	o
AA	BA	AB	BB	O	-	u
AA	BA	AB	BB	U	+	u
AA	BA	AB	BB	U	-	o
AA	BB	AB	BA	O	+	u
AA	BB	AB	BA	O	-	o
AA	BB	AB	BA	U	+	o
AA	BB	AB	BA	U	-	u
AB	BA	AA	BB	O	+	u
AB	BA	AA	BB	O	-	o
AB	BA	AA	BB	U	+	o
AB	BA	AA	BB	U	-	u
AB	BB	AA	BA	O	+	o
AB	BB	AA	BA	O	-	u
AB	BB	AA	BA	U	+	u
AB	BB	AA	BA	U	-	o
BA	AA	BB	AB	O	+	u
BA	AA	BB	AB	O	-	o
BA	AA	BB	AB	U	+	o
BA	AA	BB	AB	U	-	u
BA	AB	BB	AA	O	+	o
BA	AB	BB	AA	O	-	u
BA	AB	BB	AA	U	+	u
BA	AB	BB	AA	U	-	o
BB	AA	BA	AB	O	+	o
BB	AA	BA	AB	O	-	u
BB	AA	BA	AB	U	+	u
BB	AA	BA	AB	U	-	o
BB	AB	BA	AA	O	+	u
BB	AB	BA	AA	O	-	o
BB	AB	BA	AA	U	+	o
BB	AB	BA	AA	U	-	u

(a)

Figure B.1: Table that outputs of the position of crossings.

Bibliography

[1] L.H. KAUFFMAN, Virtual Knot Theory, *Europ. J. Combinatorics* **20** (1999), 665-671. Disponible en `https://www.math.washington.edu/~reu/papers/2011/allison/VKT.pdf`.

[2] K. MURASUGI, *Knot teory and its applications*. Disponible en `http://www.maths.ed.ac.uk/~aar/papers/murasug3.pdf`.

[3] D. ROLFSEN, *Knots and links*, second ed., Mathematics Lecture Series, vol. 7, Publish or Perish, Inc., 1990.