Reidemeister Moves on Alternating Knots and their Signed Planar Graphs

Internship Report

Submitted to: **Prof. Dr. Peter Tittmann**

Written by:

Name: Ahmed Helali
Matrikel No:

November 6, 2023

Contents

List of Figures

List of Tables

1 Introduction

"We learn the rope of life by untying its knots."

— Jean Toomer

In the nineteenth century, Lord Kelvin proposed the *vortex theory of the atom* to try and explain why chemists found only a few different types of atoms, but there are a lot of each type. The theory stated "atoms were knotted vortices in the ether"[3]. This resulted in efforts to try and find different kinds of knots. And although, Lord Kelvin's theory was wrong, *knot theory* thrived anyway. Many tabulations of knots showed up, mathematicians defined terms such as *achiral*, *amphicheiral*, *prime*, *alternating* to describe different knots.

The question of equivalence lies of course in the core of many of these terms, it is natural to come to the question; when are two knots equivalent? And what seems to be even a simpler question turns out to be not as straight forward; how to know if a knot is not the unknot? A knot as a structure is very unique, two knots could be equal although they look nothing alike. So what is equal in the eyes of knot theory? Knot theorists started to provide such definitions of equivalence and knot invariants.

Although the knot lives in three dimensional space, we study it in two dimensional space. And Translating the properties of the knot to 2D space is done by knot diagrams and other tools. The definition of equivalences of course transitions (although with some care!) to the 2D world.

Another mathematical structure that lives in 2D is planar graphs. Insights regarding the connection between knots and planar graphs were investigated. A representation of knots as planar graphs helped in tabulating knots for example[3]. It is also possible to derive the Jones polynomial, a knot invariant, from the representative planar graphs of knots[4]. An exciting connection exists between those two mathematical concepts, and graph theory has been around at least a century more before any keen mathematical investigations in knots. It is surely beneficial for both fields to aim to develop, create tools for and investigate this connection[3].

The focus of this project was to dapple further with the relation between planar graphs and knots. As we stated, the question of whether two knots are the same is a unique question. Under special constraints, certain actions on the knot does not change the knot. We will discuss such actions in more details in our report. For now, one of these actions fall under the name of *Reidemeister* moves. It has been shown[3][4][7] that these *Reidemeister* moves have an equivalent correspondence when carried to the planar graphs' world.

In this report, we present a program that takes in as input a knot diagram represented in *Gauss* Notation, and outputs the related planar graphs. The program also performs the aforementioned *Reidemeister* moves on the planar graphs.

The rest of report is divided as follows; Section 2 will discuss related background and theory in order to help us build an overview of our objective. Section 3, we state our problem statement and our end goal for the program. Section 4 is the Methodology, we highlight our software tools, class structure, external sources used and the Organization of our program. We then dive into the program's code, we highlight the different aspects and roles of each class, how every piece of the program works with the rest. Section 5, we provide the results and also highlight two examples for better clarity on the program's procedure. In section 6, we discuss our program regarding performance and efficiency, we also highlight major draw backs. we then continue to provide what paths and aspects should be regarded in future work and what protocols might highly impact the success of future projects regarding our problem statement. Section 7, we conclude our report, summarizing everything so far. Lastly, we have provided an Appendix that contains all of our results so far, a tabulation of the knot diagrams the program worked on and the entire code of the program.

2 Background & Theoretical Concepts

"If I have seen further than others, it is by standing upon the shoulders of giants."

— Isaac Newton

2.1 The Knot in $\mathbb{R}^3$

There are many formal definitions for a knot, we choose the following one[2] for its ease to build on further concepts.

Definition 1. *A **knot** is a simple closed polygonal curve in* $\mathbb{R}^3$.

Figure 1: The trefoil as a smooth curve and a polygonal curve.

This definition gives us the benefit to exclude such notions as *wild knots*[2][5].

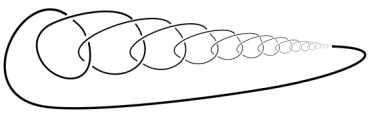


Figure 2: A wild wild knot.

We would then like to define what is a *vertex* of a knot.

Definition 2. *Let the ordered set of points* $(p_1, p_2, ..., p_n)$ *define a knot, and no proper ordered subset defines the same knot, then the points of the set* $\{p_i\}$ *are called the **vertices** of the knot.*

It is only natural to then utilize the notion of vertices of a knot and bask in its merits, and we do so in the next definition.

Definition 3. *Let* k *be a knot defined by the sequence of points* $(p_1, p_2, ..., p_n)$ *and let* j *be a knot defined by the sequence* $(p_0, p_1, p_2, ..., p_n)$*, where:*

1. *The point* p_0 *is not collinear with* p_1 *and* p_n.
2. *The triangle spanned by* (p_0, p_1, p_n) *intersects the knot* k *only in the segment* $[p_1, p_n]$.

Then, the knot j *is called an **elementary deformation** of the knot* k.

But we are not done yet, we can now state the upcoming powerful result[2].

Definition 4. *For any two knots* k *and* j*, if there exists a sequence of knots* $k = k_0, k_1, ..., k_n = j$*, where for each* $k_i + 1$ *an elementary deformation of* k_i*, for* $i < 0$*, then the knots* k *and* j *are **equivalent**.*

Since a knot is considered a closed polygonal curve, it has no starting point and no end point. Therefore, we can assign an orientation to it[5]. In this report, we will use relay on the notion of orientation in our program. So let's delve into it.

Definition 5. *An **oriented** knot consists of a knot and an ordering of its vertices. The ordering must determine the original knot. Two orderings are considered equal if the differ by a cyclic permutation.*

The equivalence relation between knots is also carried through to oriented knots.

Definition 6. *Let there be a sequence of elementary deformations changing one oriented knot to another, then these two knots are called **oriented equivalent**.*

The necessity of displaying knots on paper, on boards and in books led to the formalization of knot projections and diagrams in the plane, or in other words, $\mathbb{R}^2$.

2.2 Knot Projections and Knot Diagrams in $\mathbb{R}^2$

Let k be a knot, then we denote the projection of k by $p(k) = \hat{k}$. If k has an orientation then naturally the projection $\hat{k}$ inherits the orientation. Note that k lives in $\mathbb{R}^3$ space while $\hat{k}$ lives in $\mathbb{R}^2$[4][5]. Where k crosses under or over itself, there will exist an intersection in its projection $\hat{k}$. We would like now to reference the next definition[5].

Definition 7. *Let $\hat{k}$ have the following properties:*

1. *$\hat{k}$ has finite points of intersection.*
2. *If Q is a point of intersection of $\hat{k}$, then the inverse image of Q in k has exactly 2 points.*
3. *A vertex in the knot k is never mapped onto a double point in $\hat{k}$.*

Then, we call k' a regular projection.

Figure 3 show a point that does not abide by property number 2. Similarly, figures 4 and 5 show points that do not abide by property number 3.

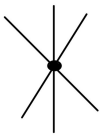


Figure 3: Violation of Property 2.

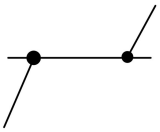


Figure 4: Violation of property 3.

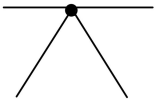
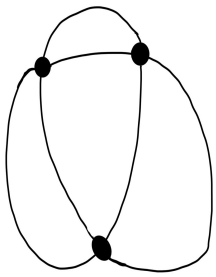
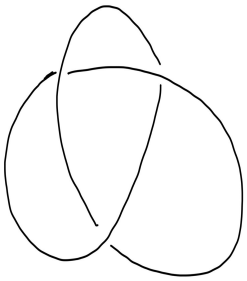


Figure 5: Violation of property 3.

In order to incorporate the "under-and-over" crossing information in the projection, we draw the projection to *appear* as if it cuts itself. And we call this altered projection a *regular diagram*[5]. From here on out, we will denote the diagram of the knot k as k' (e.g. if the knot is called k_1 then its diagram is k_1').



(a) A knot projection.



(b) A knot diagram.

Figure 6: The trefoil as a projection and as a diagram.

Definition 8. *A regular diagram k' with no* ***reducible*** *crossings is called a* ***reduced regular diagram****.*

For example, figures 7 and 8 are reduced diagrams while figure 9 is not.

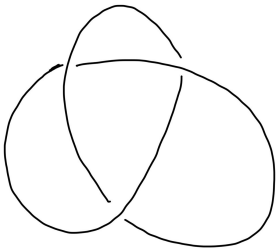


Figure 7: The trefoil.

Figure 8: 7_1 knot.

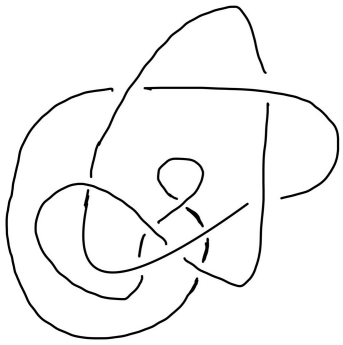


Figure 9: The star with reducible crossings.

Definition 9. *An* alternating *knot diagram is a knot diagram that has the crossings alternate between over and under as we go any orientation[3].*

Definition 10. *An* ***alternating*** *knot is a knot that has at least one alternating diagram.*

2.2.1 Equivalence with respect to Knot Diagrams and Projections

We would like to highlight the importance of knot projections and diagrams. After all, through out our report we will be relaying so much on them in our program.

In [2], there are two theorems regarding projections that set up the field for us. The first theorem states that even if a knot does not have a regular projection, there is an equivalent *nearby* knot with a regular projection. The second theorem states that if a knot has a regular projection, then all *nearby* knots also have a regular projection. The term *nearby* relates to the distance between vertices as we make elementary deformations. We now come to one of major result we wanted to introduce[2].

Theorem 1. *Let k and j be knots that have regular projections and identical diagrams, then k and j are* ***equivalent****.*

2.3 Reidemeister moves

Reidemeister moves are ways to alter a knot diagram that will change the relation between the crossings[3]. There are three Reidemeister moves, each paired with its inverse. The *first Reidemeis-*

ter move allows us to add or take a twist away. The *second Reidemeister move* allows to add or remove two crossings, under certain conditions. The *third Reidemeister move* allows us to slide a segment of the knot from one side of a crossing to the other end. Figure ?? depict those moves.

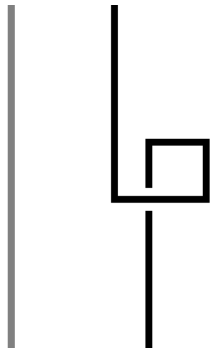


Figure 10: *RI* aka "a twist".

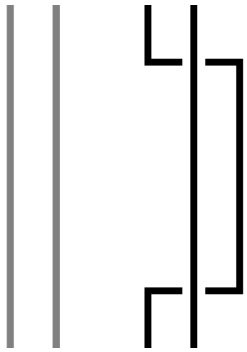


Figure 11: *RII* aka "a poke".

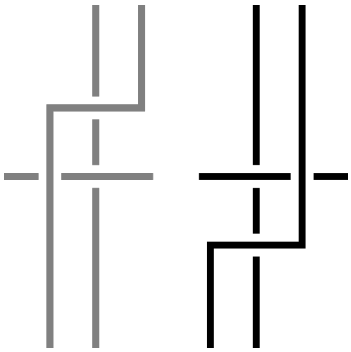


Figure 12: *RIII* aka "a slide".

Figure 13: The Reidemeister Moves

It is not unreasonable to think that the alterations done by these Reidemeister moves on the knot diagram will result in representing a totally different knot. However, we have the following observation by Alexander and Briggs[2]:

Theorem 2. *If Two knots are equivalent, then their diagrams are related by a sequence of Reidemeister moves.*

We also include the next theorem from [11]:

Theorem 3. *Two knot diagrams are equivalent, if and only if, they are related by Reidemeister moves.*

We will sometimes refer to Reidemeister moves by their nicknames; a "twist" for the first, a "poke" for the second and a "slide" for the third.

2.4 Notation for Knots

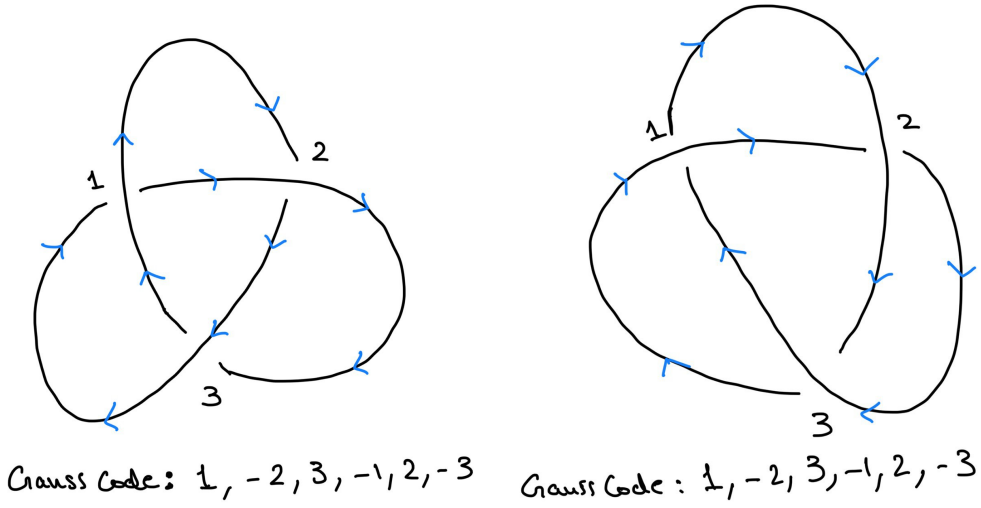
In the process of tabulating knots, codes and notations have been created to represent them in a compact and concise manner. We would like to discuss one specific notation as it relates closely to our program.

Gauss Code

It is amazing that even the great Gauss dabbled with knot theory. The notation is straight forward and has some similarities with another notation, Dowker–Thistlethwaite notation.

If we have some knot diagram, we give it an orientation, we then pick and arbitrary point and walk along the strands of the knot, when we meet a crossing for the first time we label it, starting with 1 and so on. We then start again at the same point, and if we are passing the knot at over-crossing, we we give the label a positive sign, if it was an under-crossing we give the label a negative sign[14].

This method has the downside that it does not differentiate between a knot and its mirror if they are different. For example the Gauss code for the trefoil and the left trefoil is the same: $1, -2, 3, -1, 2, -3$[14].



(a) The trefoil.

(b) Its mirror.

To address this, the *extended Gauss code* was proposed which accounts for the crossings local orientation. To define local orientation, we define first the difference between right-handedness and left-handedness[14]:

- A right-handed crossing means that if our thumb points in the direction of the orientation of the over-strand then the rest of our finger will point in the direction of the orientation in the under-strand.
- A left-handed crossing means the same thing but using our left hand.

And now, the *extended Gauss code* changes the Gauss code as follows[14]:

We do the same thing in our first iteration and label the crossing. However, in the second iteration over the knot the first occurrence of the label has a sign to depict whether it was an over or under crossing and the second occurrence has a sign to depict the local orientation of the crossings. If the local orientation is right-handed, we give the label a positive sign, if it is left-handed, we give it a negative sign.

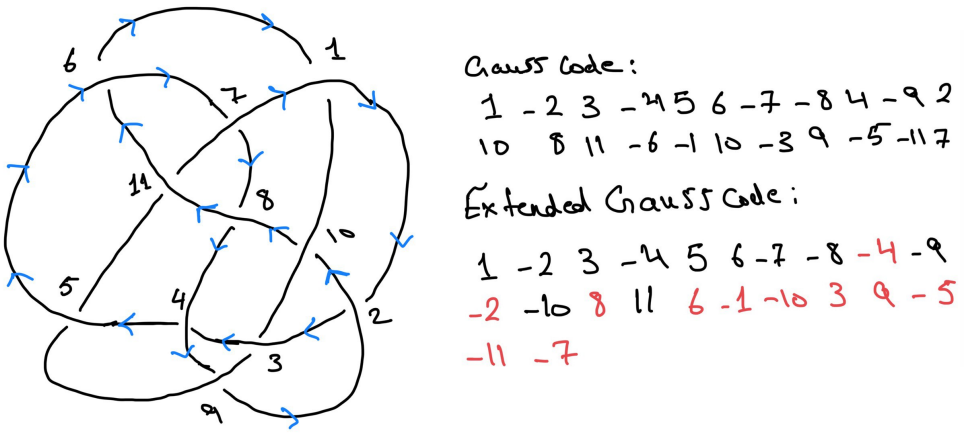


Figure 15: A knot and its Gauss Codes. The red numbers Indicate the difference between the Extended Gauss Code and the normal notation.

2.5 Knots and their Signed Planar Graphs

Most of the work our program does, it will be doing it with the aforementioned *signed planar graphs*. In the upcoming subsection, we will show how they are constructed, their equivalence with their respective knots and more.

The Knot Projection and The Plane Graph

Let's again reconsider the notion of the knot projection. Assume we have a knot k, its projection $\hat{k}$ and its diagram k'. If k has at least one crossing, then its projection can be described as a 4-regular plane graph, where each crossing is a vertex. The diagram k' can also be described by the same 4-regular plane graph, but we will need to account for the information of the over and under crossing at each vertex[4].

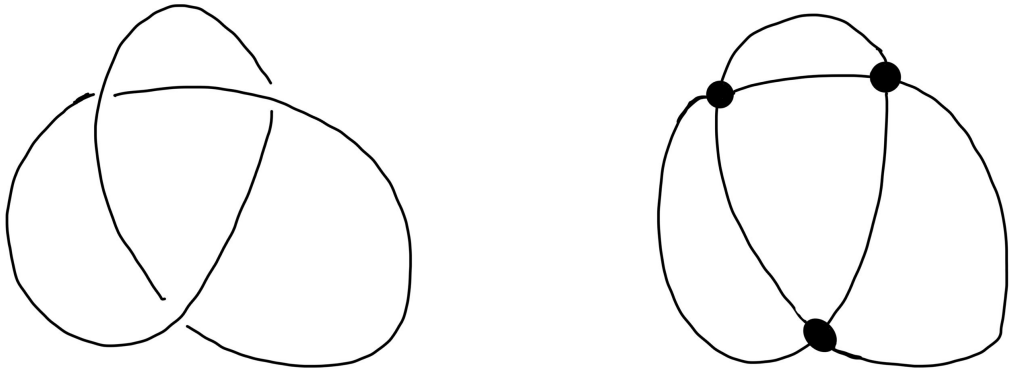


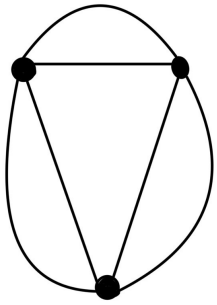
Figure 16: The trefoil and its *plane graph*.

From the Plane Graph to the Signed Planar Graphs

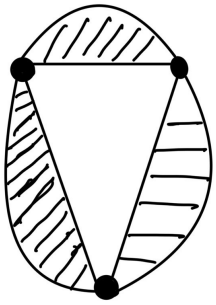
In the context of the following, we can consider $\hat{k}$ as the shadow of k'. Since the knot projection $\hat{k}$ is a 4-regular plane graph, then its dual graph is bipartite[4].

We now can colour the faces of the shadow of the knot projection, or in other words the faces of the plane graph, using two colours, usually they are black and white. We start by colouring the exterior face white, and now each face that share a boundary with the exterior face, we colour black, and then each uncoloured face that shares a boundary with a black face we colour white, and so on we alternate until we have coloured all faces. For its respective colour, each face now represent a vertex for that graph, and we draw an edge between the vertices for each vertex the faces shared in the plane graph. In the case that a vertex of the plane graph lies only in one coloured face, then we draw loop at the respective vertex that represents that face.

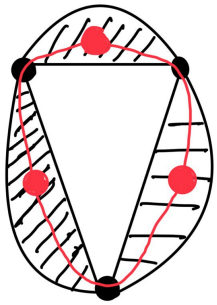
To sum up, we have two planar graphs, one black and one white. let's denote the black graph with b and the white graph with w. We would like to highlight again up to this point what do the graphs actually represent. Let's take b as an example, every vertex in b represents a black face, and every edge of b represents a crossing that these faces share. Figure 17 help to visualise the process.



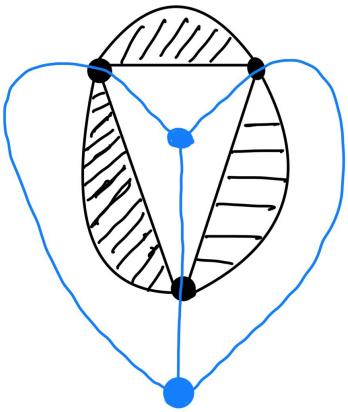
(a) The plane graph for the trefoil.



(b) Shaded plane graph.



(c) Red representing the black graph.



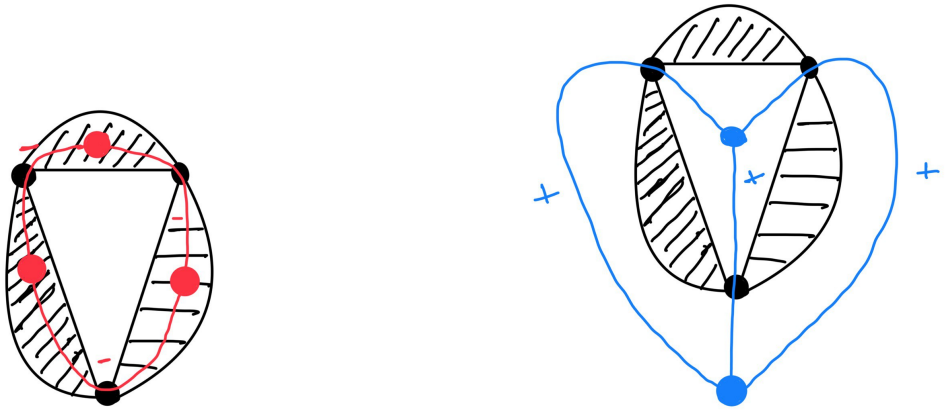
(d) Blue representing the white graph.

Figure 17: Construction of the planar graphs.

The graphs b and w are always connected graphs. Moreover, b and w are the graph duals of each other and from either b or w, we can go backwards and construct the plane graph (the knot projection $\hat{k}$, the shadow of the knot diagram)[4].

What remains now to fully reflect the original knot is the under-and-over information. In order to represent this in the graph, we go to each crossing and we imagine the over-strand rotating *anticlockwise* until it lies on the under-strand, we take note of the face the over-strand traversed[1] and keep track of the face's colour. Back to the graphs b and w, we go to each edge and since each edge symbols a crossing, we check the colour we took note of previously, if it is the same as the graph, the edge gets a positive sign, if it is opposite, then the edge gets a negative sign.

[1]The over-strand passes simultaneously through two faces as it rotates. However, they will always be of the same colour (figure 25 gives a visual representation).



(a) The black graph with signs.

(b) The black graph with signs.

Figure 18: The planar signed graphs for the trefoil.

After adding the signs, we now call b and w the ***signed planar graphs*** of the knot.

We note that from the ***signed planar graphs***[2], we can obtain the original knot diagram. The process is described in [3] and [5].

2.6 Reidemeister Moves on the Signed Planar Graphs

We carry the same concepts of Reidemeister moves to *signed planar graphs*.

The *first Reidemeister move* translates to deleting a loop in one planar graph and a deleting spike in the other.

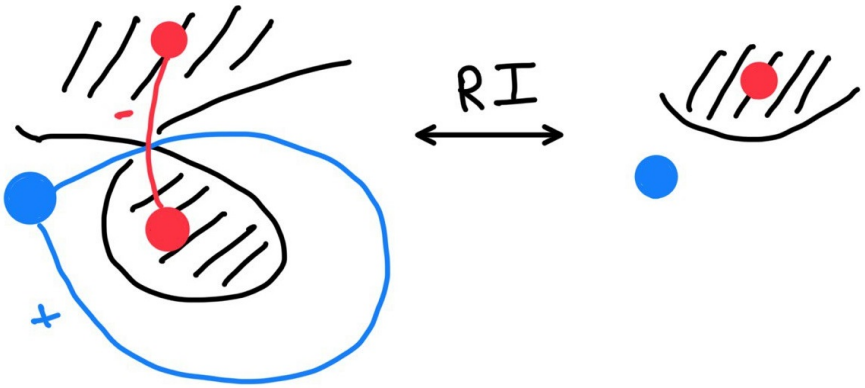


Figure 19: The first Reidemeister move.

[2]Actually, from only just one!!!

The *second Reidemeister move* translates to deleting two parallel edges in on graph of different signs and contracting a 2-path (with edges of different signs) in the other graph.

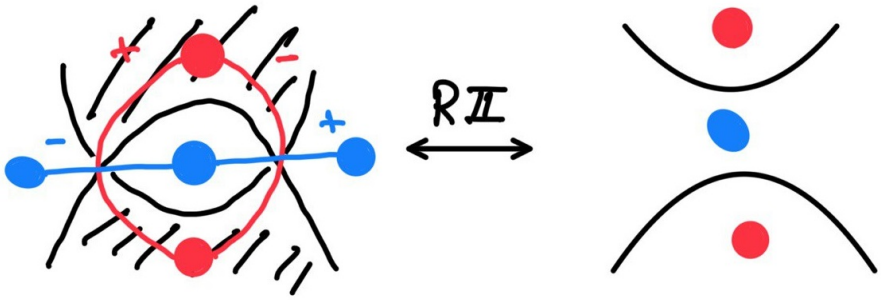


Figure 20: The second Reidemeister move.

The *third Reidemeister move* translates to changing a cycle C_3 in one graph to a star $K_{1,3}$ and doing the reverse in the other.

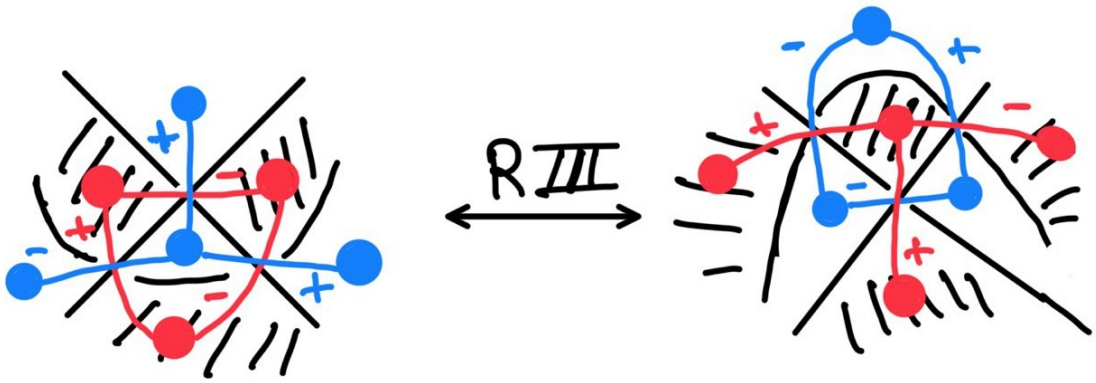


Figure 21: The third Reidemeister move.

More technical detail will be shown in the Methodology(4). The Reidemeister moves here are shown on a more of a local level and we will delve in the details of how our programs handles each *Reidemeister move* and how it affects the whole graph.

3 Problem Statement

"No problem can withstand the assault of sustained thinking."

— Voltaire

The main inspiration for our internship is the following:

- Let k be a knot and k' be its projection.
- We perform a sequence of *Reidemeister moves* on k' to obtain the knot diagram j'.
- Let b_k, w_k, b_j, w_j be the signed planar graphs of k' and j' respectively.

  Now, can we do the same on graphs b_k, w_k to obtain b_j, w_j?

The question as it stands is hard[3], so we have reformulated our starting point.

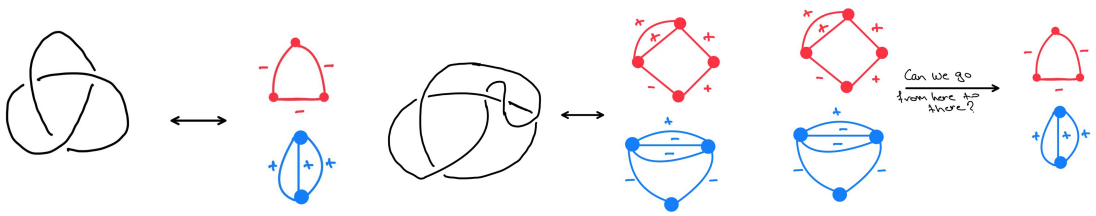
let k be a knot, k' its knot diagram and $\mathbf{j'}$ its ***reduced regular diagram***. Let $(R_1, R_2, ..., R_n)$ be a sequence of Reidemeister moves taking j' to k', $(k'_0, k'_1, ..., k'_n)$ be a sequence of knot diagrams, where $i \in \{1, ..., n\}$, $k'_0 = j'$, $k'_n = k'$ and k'_i is the knot obtained from applying Reidemeister move R_i on k_{i-1}. we constrain that $\forall\, j, l \in \{1, ..., n\}$:

- k'_j does not have more crossings than k'_l, for $j < l$.
- $\nexists\, R_j, R_l$ such that R_j is a *third Reidemeister move* and R_l is its inverse.

These restrictions are mainly to avoid the situation where we have to perform a *third Reidemeister move* and increase the crossings in order to reduce the knot diagram. An example of such can be found in [3].

And now with the above notation, let b_k, w_k be the signed planar graphs of k', our question becomes:

> Can we find a sequence of Reidemeister moves on b_k, w_k to obtain the signed graphs b_j, w_j of j'?



(a) Signed planar graphs from *irreducible* trefoil diagram.

(b) Signed planar graphs from a trefoil with a poke.

(c) Can we reduce the pair to the *irreducible* pair?

Figure 22: A visual overview of our problem statement.

[3] Assuming we know nothing else except the planar graphs, it is difficult to find a suitable sequence of Riedemeister moves.

4 Methodology

"It makes no difference how they are loosed."

— Alexander the Great,
referring to the Gordian Knot.

We have laid out the background, given our motivation and stated our problem. Now, it's time to get to work!

We would like to begin first by detailing the organization, class structure and different software tools.

4.1 Software and Technicalities

4.1.1 Programing Language and Main External Packages

we would like to state the overall structure of our project and the different classes. However, to avoid ambiguity, we will state first the different tools used in our program.

Our program is written in *Python* and nothing else. In order to achieve our objectives we utilized the following two libraries:

- NetworkX.
- pyknotid.

These two packages have been essential to our program. NetworkX[6] deals with graphs, while pyknotid[8] deals with knots and related concepts. More can be found about these two libraries in their respective docs.

Library	Link for Docs
NetworkX	https://networkx.github.io/
pyknotid	https://pyknotid.readthedocs.io/en/latest/

Table 1: Python Libraries and Documentation Links

When we go through the code, we will highlight the crucial methods and classes that we utilized from these sources.

4.1.2 Class Structure and Organization

In our program, there are two classes, one utility class, one helper module, two main scripts and one test script. let's go through them quickly.

HardCodedKnots In this class, we have encoded Gauss codes to represent different knot diagrams. The main functionality of this classs is to call instances of it and pass it to the **SignedPlanarGraph** class.

ReidemeisterMoves The name says it all, this *utility* class is responsible for performing the *Reidemeister moves* on the signed planar graphs.

SignedPlanarGraph This class is the main player. It is responsible for taking a Gauss Code representing a knot diagram and turning it into a pair of signed planar graphs.

main This is one of the main scripts, where we could lauch the whole program.

main_iterative The other main script, the main difference is that it is set up such that the program will run over all knot diagrams encoded in **HardCodedKnots**.

check_gauss_code This test script simply takes a Gauss code and identifies the knot up to reflection. It is useful in checking results, debugging and verifying inputted Gauss codes to the program.

my_helpers A module with many methods we use throughout the code.

File Name	Type	Description
HardCodedKnots	class	Encoded Gauss codes.
ReidemeisterMoves	class	Performing Reidemeister moves.
SignedPlanarGraphs	class	Signed Planar Graph construction and manupilation.
check_gauss_code	test script	verifying input and output through Gauss codes.
main	main script	Launch the program.
main_iterative	main script	Launch the program on everything we got!
my_helpers	module	helper methods used throughout the program.

Table 2: Brief overview of different files.

And while probably not as important, the folder and file structure is displayed in the figure on the right(23).
In the next subsection, we will tackle the main players in the external libraries.

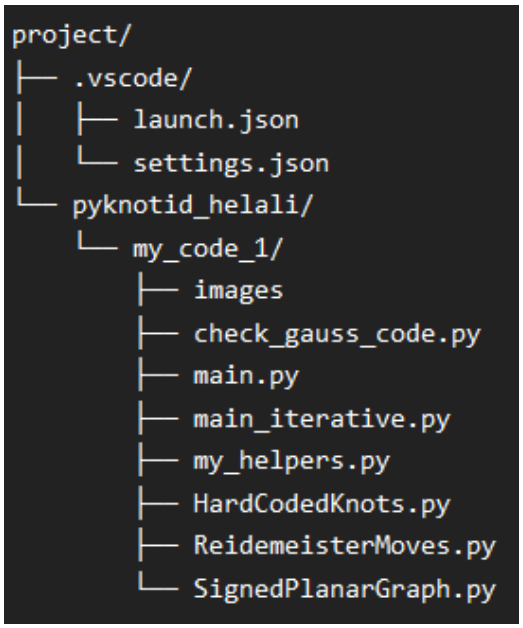


Figure 23: Organization of Project

4.1.3 Major Modules from Imported Libraries

In the upcoming, we will state the major functionalities we used. More details is available in the package respective docs.

NetworkX

Graph This is a class to represent an un-directed graph structure. It allows loops, but no parallel edges are allowed.

MultiGraph This is a class to represent an un-directed graph structure. It allows both loops and parallel edges. This is the main class we will relay to represent our *signed planar graphs*.

PlanarEmbedding This is a class that represents a planar graph with its embedding.

is_isomorphic This is a method that takes two graphs and checks if they are isomorphic.

check_planarity Checks if a graph is planar and returns a embedding for it.

PlanarEmbedding.traverse_face This method is called on an instance of PlanarEmbedding. The method takes an edge and returns the nodes of the face that belong to the edge.

cycle_basis This method takes a graph and returns a list of cycles.

drawing.nx_agraph.to_agraph This method is used to draw graphs and export to images.

pyknotid

catalouge pyknotid has a prebuilt databases for knots up to 15 crossings. This package provides a knot lookup by name and by some invariants. However, the download of the database is required (This is provided in pyknotid's docs).

spacecurves This module includes multiple classes to perform topological analysis on knots and work with them in $\mathbb{R}^3$.

representations This module provides different ways to represent knots from invariants, diagrams and notations.

representations.gausscode.GaussCode This is a Class that represents a knot as a Gauss code. We will provide more details momentarily(4.1.3) as the notation here is slightly different from the usual theoretical notation and from the one we introduced in our background section(2.4).

catalogue.identify.get_knot This method takes a string and returns a an object of type "¡Model: Knot¿". Examples of strings that can be passed are "3_1" for the trefoil and "4_1" for the figure eight knot.

catalogue.database.space_curve This method takes the object returned from get_knot and returns a Knot Object. The method space_curve is defined in the module mentioned before "spacecurves".

representation.space_curve This method creates a Knot object from an representation. It also relays on the module "spacecurves".

In our code, it will become clear how we utilize the different functionalities of pyknotid. Although we mentioned many modules and classes, pyknotid have them connected and integrated in a way that using pyknotid does not actually require deep analysis of the library itself, and going from one type of representation and/or class becomes straightforward after some experimentation.

Gauss Code in pyknotid

The major concepts are the same as the extended Gauss code we provided in our background section. However, pyknotid uses a different kind of notation. As we traverse the knot, each occurrence of a crossing is represented by three characters; the label number, "-" or "+" to indicate "under-and-over" information, and "a" or "c" to indicate local orientation. And the Gauss code is made up of these three characters separated by a comma. What follows elaborates more.

Symbol	Description
+	An over-crossing.
-	An under-crossing.
a	Anticlockwise/right-handed local orientation.
c	Clockwise/left-handed local orientation.

Table 3: Description of Gauss code components in pyknotid.

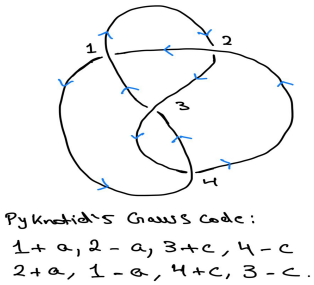


Figure 24: an example of pyknotid's Gauss code notation.

For the diagram of the figure eight shown, with the given labeling and given orientation, the Gauss code according to pyknotid will be "1+a,2-a,3+c,4-c,2+a,1-a,4+c,3-c".

Other Packages

We would be remiss if we don't mention the following packages/libraries.

copy Used to create copies of objects.

collections Used through the construction process of the *signed planar graphs*

planarity Through NetworkX, we use crucial modules from planarity. They are related to planar graphs and their embeddings[20].

4.2 The Algorithm and the Program

4.2.1 Overview of the Algorithm

Keeping things straight forward, we start by stating the steps of our approach:

1. We have a knot k.
2. We create two knot diagrams k' and j'. Keeping to our notation, j' is the reduced diagram.
3. k' has been obtained from a sequence of Reidemeister moves made on j', abiding by the constraints we stated in the Problem Statement(3).
4. We calculate the Gauss code for both knot diagrams.
5. We input the Gauss codes to the program.
6. We obtain the signed graphs b_k, w_k, b_j, w_j.
7. We input b_k, w_k to the program.
8. The program performs all valid Reidemeister moves possible and outputs the signed graphs $\hat{b_k}$, $\hat{w_k}$.

9. We compare b_j, w_j and $\hat{b_k}$, $\hat{w_k}$.

Simple enough!

4.2.2 The Program

main

Our strategy in this subsection is to start with the main script and as we go through it, we will cover all aspects of our program. And without further ado, here it is:

```
1  from SignedPlanarGraph import SignedPlanarGraph
2  from HardCodedKnots import HardCodedKnots
3  from ReidemeisterMoves import ReidemeisterMoves
4
5  hcg = HardCodedKnots(code_no=3113)
6
7  k_gauss = hcg.get_code_gauss()
8
9  sp_graph = SignedPlanarGraph(code_gauss = k_gauss)
10 sp_graph.create_and_draw_sp()
11
12
13 reid_sp = ReidemeisterMoves.reidemeister_main(sp_graph)
14
15 reid_sp = SignedPlanarGraph(code_gauss=reid_sp.code_gauss)
16 reid_sp.create_and_draw_sp()
17
18 #print(reid_sp.code_gauss)
```

The first three lines are simply importing our three classes we mentioned before. let's look at line number 5 with more detail.

```
5  hcg = HardCodedKnots(code_no=3113)
```

In line 5, we create the variable *hcg* and we call the constructor of the class **HardCodedKnots**. In the constructor we pass an identifer to the knot diagram, here it is the variable *code_no* and the value is "3113". We will highligh what does this value mean in the **HardCodedKnots** class. In line 6, we pass the Gauss code to the variable *k_gauss*.

```
9  sp_graph = SignedPlanarGraph(code_gauss = k_gauss)
10 sp_graph.create_and_draw_sp()
```

In line 9, we create an instance of the class **SignedPlanarGraph** and this is saved in the variable *sp_graph*. In line 10, we call the method *create_and_draw_sp()*. This method takes the knot diagram as a Gauss code, produces the two signed planar graphs saved as attributes in the instance *sp_graph*. The method also produces four graph images, two graphs for black and two graphs for white. for each, One graph with no sign labels and the other is with.

```
13 reid_sp = ReidemeisterMoves.reidemeister_main(sp_graph)
```

In line 13, we use the utility class **ReidemeisterMoves**. The method *reidemeister_main()* takes as input the signed planar graph instance *sp_graph* and produces the graph *reid_sp* after performing all possible Reidemeister Moves.

```
15 reid_sp = SignedPlanarGraph(code_gauss=reid_sp.code_gauss)
16 reid_sp.create_and_draw_sp()
```

In line 15, we recreate the instance *reid_sp* this to ensure that all attributes have been cleared and set correctly. In line 16, we again produce the two signed graphs and export the graph images.

HardCodedKnots

```
1  from pyknotid.catalogue import get_knot
2  import pyknotid.representations as rep_
3
4  class HardCodedKnots:
5
6      code_numbers = [31, 311, 3111, 312, 3113, 41, 411, 4112, 412,
           51, 513, 5131, 52, 522, 61, 62, 63,
7                      71, 72, 73, 74, 75, 76, 77]
```

In the beginning of the file we import the modules relates to pyknotid. At line 4, the class begins. The class has a global attribute code_numbers, we will highlight there use later. Let's move forward to the *__init__* function.

```
10     def __init__(self, code_no = None, code_string = None):
11
12         if code_no == 31 or code_string == "trefoil" or code_string
               == "3_1":
13             k = get_knot('3_1').space_curve()
14             self.code_gauss = k.gauss_code()
15             return
16         if code_no == 311 or code_string == "trefoil_twist" or
               code_string == "3_1_1":
17             self.code_gauss = rep_.GaussCode("1+c,2-c,3+c,1-c,2+c,4+
                   c,4-c,3-c")
18             return
```

the *__init__* function can take two parameters, *code_no* or *code_string*. We see that we have *if* blocks, one after the other. There are multiple *if* blocks after the one we shown here, In each *if* block a knot diagram is represented by a Gauss Code. Let's zoom in on the first one of them.

```
12         if code_no == 31 or code_string == "trefoil" or code_string
               == "3_1":
13             k = get_knot('3_1').space_curve()
14             self.code_gauss = k.gauss_code()
15             return
```

The parameters that could be passed to the *__init__* function where *code_no* and/or *code_string*. So this *if* block checks if *code_no* is equal to 31 or *code_string* is equal to "trefoil" or to "3_1". if one of these conditions hold, the program enters the *if* block.

In line 13, we use pyknotid's *get_knot()* method to obtain the trefoil and we use pyknotid's *space_curve()* to flesh out the knot and calculate certain invariants and notations that were not there yet just from the *get_knot()* method. This information is now saved in the variable *k*. Line 14 sets the attribute *code_gauss* of the instance of the class **HardCodedKnots** to the **Gauss Code** related to *k*. Line 15 simply exits the *__init__* function.

We hope it was clear what was the overall objective of the *__init__* function. recall that in "main" we had the following:

```
5  hcg = HardCodedKnots(code_no=3113)
```

if *code_no* was equal to 31, *hcg* would be initialized with the attribute *code_gauss* set to the Gauss code of the trefoil.

All reduced regular diagrams of prime knots up to seven crossings are set up as the trefoil. Here is another example for the knot 6_3.

```
80          if code_no == 63 or code_string == "6_3":
81              k = get_knot("6_3").space_curve()
82              self.code_gauss = k.gauss_code()
83              return
```

let's check the *if* block for the *code_no* of 3113.

```
23          if (code_no == 3113 or code_string == "trefoil_twist_slide"
24                  or code_string == "3_1_1_3"):
25              self.code_gauss = rep_.GaussCode("1+c,2-c,3+c,4+c,5+a,6-
                    a,7-c,3-c,8-a,5-a,6+a,1-c,2+c,7+c,4-c,8+a")
```

In this case, we rely on pyknotid's module "representation". In line 30, we use it to create a representation of the knot using the Gauss code notation of pyknotid.

We would like also to point out, that there is a certain convention we followed while creating the different *code_no* and *code_string* for different Gauss codes. for *code_no*, The name of the knot is followed by numbers representing the Reidemeister moves we performed to obtain the knot. For example, from *code_no* 3113, we deduce that this is the knot 3_1 and we performed a first Reidemeister move then a third Reidemeister move. For *code_string*, we have two options; if the knot has a famous name, we chain the name and then the Reidemeister moves performed, separated by underscore. If there is no famous name, we simply chain the numbers in *code_no* also separated by underscore. That's why in the previous *if* block, "trefoil_twist_slide" and "3_1_1_3" show up.

In table (4), there is a full tabulation showing each knot diagram, its relevant Gauss code, *code_no* and *code_string*.

And to the final detail in *HardCodedKnots*, the getter method *get_code_gauss()*.

```
119     def get_code_gauss(self):
120         return self.code_gauss
```

SignedPlanarGraph

In the land of **SignedPlanarGraph**, knot diagrams find themselves transformed, gaining magical powers to shape-shift from diagrams to graphs and back again.

let's start by stating the imports and global attributes.

```
1  import networkx as nx
2  import my_helpers as help
3  import copy
4
5
6  class SignedPlanarGraph:
7
8      drawing_number = 0
```

NetworkX makes its entrance, we also import other relevant packages and our helpers. the global attribute *drawing_number* will be used when we start producing images for the signed planar graphs.

The following can be passed as parameters to the *__init__* method.

- *black_graph_data.*

- *white_graph_data.*
- *knot.*
- *code_gauss.*
- *signed_black.*
- *signed_white.*

We would like to point out that our approach changed thorough out the project, and that in the program only *code_gauss* will be passed as a parameter.
The class **SignedPlanarGraph** has the following attributes:

- *black_graph*; this has data regarding the black graphs' nodes and edges.
- *white_graph*; The same as above but for the white graph.
- *black_loops*; Loops need special attention and this attribute is responsible to keep track of loops in the black graph.
- *white_loops*; Same as before but for the white graph.
- *code_gauss*; This is initial Gauss code of the knot diagram.
- *signed_black*; this attributes saved the signing labels for the edges of the black graph.
- *signed_white*; the same thing but for the white graph.

```
9  sp_graph = SignedPlanarGraph(code_gauss = k_gauss)
10 sp_graph.create_and_draw_sp()
```

From the main script, in line 9 we created an instance of **SignedPlanarGraph** and saved it in the variable *sp_graph*. As stated we only passed the Gauss code saved in *hcg*. So as things stand, the instance of **SignedPlanarGraph** has the Gauss code saved in the attribute *code_gauss*. Now let's direct our attention to line 10 in the main script and the method *create_and_draw_sp()*.

```
547     def create_and_draw_sp(self):
548         graph = help.create_planar_graph_from_gauss_code(self.
                code_gauss)
549         self.create_nodes_from_faces(graph)
550         self.create_edges_after_nodes()
551         self.sign_sp()
552         self.draw_sp_graph()
553         self.draw_sp_graph_sign()
```

create_planar_graph_from_gauss_code() takes a Gauss code and returns a graph. Although, the name of the method has "planar" in it, we rely on NetworkX and planarity to handle the creation of a planar embedding.

create_nodes_from_faces() is responsible for the creation of the vertices/nodes of the signed planar graphs.

create_edges_after_nodes() is responsible for adding edges with respect to the crossings shared between faces.

sign_sp() adds the signs on the edges.

draw_sp_graph() draws the planar graphs and exports them as images.

draw_sp_graph_sign() draws the graphs with the signs on the edges.

Nodes/Vertices Creation from the Faces of the Shadow

The method *create_nodes_from_faces()* takes variable the plane graph (The shadow of the knot diagram) as input. The following is what occurs;

nx.check_planarity() This method creates an instance of **PlanarEmbedding** from the passed variable *graph*.The method is defined in NetworkX.

get_faces_list() This method takes an instance of **PlanarEmbedding** and produces the faces of the plane graph. we utilize the method **PlanarEmbedding**.*traverse_face()* to create the faces. However, this method does not account for loops or parallel edges.

add_loops_face_list() This methods adds the faces with respect to loops and parallel edges.

get_longest_region() This method chooses the exterior face based on length.

get_outer_reigons() This method starts colouring the exterior face as white and its neighbours as black.

shade_faces() This methods completes the colouring of the faces and returns two lists of faces one for each colour.

We then iterate over the list of faces, add the nodes to their respective graph, i.e. black faces are added to the attribute *black_graph* and in respective fashion for *white_graph*. We also save loops in either *black_loops* or *white_loops* depending on which graph they belong too.

Edge Creation

The method *create_edges_after_nodes()* uses the new initialized attributes to get the job done for edges. for each graph we utilize the node name to check if two nodes share a crossing or not. We also utilize the attributes *black_loops* and *white_loops* to account for loops in the graphs.

Signing the Edges From the Gauss Code

We now have the nodes and edges for each graph, what remains is to sign the edges and our construction is complete. This is the work of the method *sign_sp()*. We rely on the attribute *code_gauss* to help us sign each edge. We also utilize the attributes *black_loops* and *white_loops* as loops create corner cases for signing.

For each edge in one of the signed graph, we check what crossing it represented, then using the Gauss Code, we obtain the face that the over-strand and under-strand bound. Then, depending on the local orientation of the crossing, the colour of the bounded face and which graph we are signing, we know what sign to give to the edge.

The information is saved in *signed_black* for the black graph and *signed_white* for the white.

The following figures display different scenarios for signing.

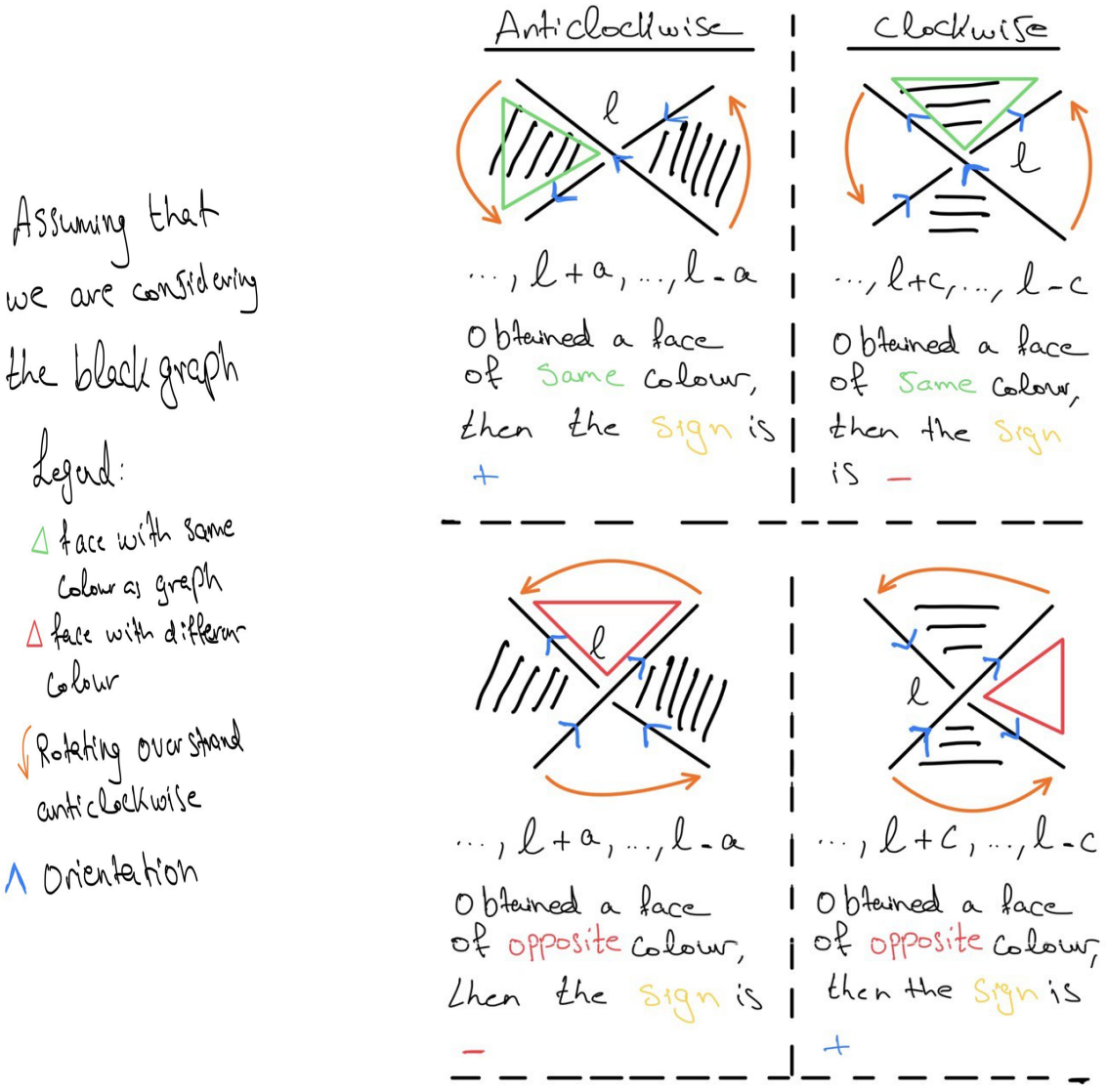


Figure 25: Procedure of how we sign.

Drawing and Exporting the Signed Planar Graphs

What we thought would be one of the main challenges in this program, turned out to be an easy task thanks to the external libraries. The methods *draw_sp_graph()* and *draw_sp_graph_sign()* perform this responsibility. From NetworkX, We utilize the methods *to_agraph()* and *draw()* for producing images.

draw_sp_graph() draws the graphs without signs. Then, we add the signs from *signed_black* and *signed_white* and *draw_sp_graph_sign()* draws them as labels on the edges.

Methods for ReidemeisterMoves

We would like to highlight some methods in this class that are used in the utility class **ReidemeisterMoves**.

create_and_sign_sp() In the process of performing Reidemeister moves, we obtain new signed graphs, this method ensures that all attributes of the new signed graphs are set correctly.

find_cycles_of_length() This method is used for the *third Reidemeister move*, where we find cycles of length three in one of the graphs. We utilize NetworkX's method *cycle_basis()*.

ReidemeisterMoves

The class **ReidemeisterMoves** is a utility class, which is to say it contains only methods and we never create instances of it.

In the main script, we had the following.

```
13 reid_sp = ReidemeisterMoves.reidemeister_main(sp_graph)
14
15 reid_sp = SignedPlanarGraph(code_gauss=reid_sp.code_gauss)
16 reid_sp.create_and_draw_sp()
```

Performing All valid Reidemeister moves

The method *reidemeister_main()* is responsible for performing all possible Reidemeister moves. It contains the following methods.

reidemeister_I() This method performs Reidemeister moves of the first type.

reidemeister_II() This method performs Reidemeister moves of the second type.

reidemeister_III() This method performs Reidemeister moves of the third type.

reidemeister_main() contains a while loop with the above methods in order. It keeps track of whether any of the above methods changed our signed planar graphs. When an iteratoin of the while loop occurs and no changes were performed on the signed planar graphs, it breaks out of the loop and returns the resulted signed planar graphs.

Reidemeister I

The method *reidemeister_I()* is responsible to perform the first Reidemeister move. In similar fashion as *reidemeister_main()*, it keeps performing the *first Reidemeister move* until no more changes occur. It contains the following.

reidemeister_I_helper() This is method contains two smaller helper methods:

reidemeister_I_helper_black()

reidemeister_I_helper_white()

Each one of the smaller helper method is related to one of the signed planar graph. let's consider *reidemeister_I_helper_black()*, as *reidemeister_I_helper_white()* does the exact same steps but for white. *reidemeister_I_helper_black()* starts by searching for loops in the black graph. If we find one, we check if the white graph has a related spike node. If both conditions are met, we modify the Gauss Code to perform the Reidemeister move, and recreate the signed planar graphs.

Reidemeister II

The method *reidemeister_II()* is responsible to perform the second Reidemeister move. It also keeps performing the *second Reidemeister move* until no more changes occur. It contains the following.

reidemeister_II_helper() This is method contains two smaller helper methods:

reidemeister_II_helper_black()

reidemeister_II_helper_white()

Each one of the smaller helper method is related to one of the signed planar graph, we again consider just one of them. *reidemeister_II_helper_black()* starts by saving all available pairs of parallel edges. It then it filters them keeping only pairs of edges with different signs. Then it searches for the related 2-path edges in the white graph. If all conditions are met, we again modify the Gauss code and create the new signed graphs.

Reidemeister III

The method *reidemeister_III()* is for the *third Reidemeister move*. It performs the *third Reidemeister move* until no more changes occur. It contains the following.

reidemeister_III_helper() This is method contains two smaller helper methods:

reidemeister_III_helper_black()

reidemeister_III_helper_white()

Each one of the smaller helper method is related to one of the signed planar graph. for *reidemeister_III_helper_black()*, we use the method *get_cycles_of_length()* to obtain cycles of length three. We iterate over the cycles, checking if one of them have a related star graph in the white graph. If the condition is met for one of the cycles. We modify the Gauss code and recreate the signed planar graphs.

my_helpers

Let's highlight the methods in our helper module.

is_cyclic_list() Used to check whether two lists are equivalent with respect to cyclic order. This method is mainly used to check that we do not have duplicate faces When creating signed planar graphs.

sublistExists() Used to check if a list is a subset of the other, this is used in edge creation.

edges_set_creation() Used in the colouring of the faces, it creates the edges that bounded a face in the plane graph (shadow of the knot diagram).

get_bounded_face() Used in signing to obtain the face bounded by the over-strand and the under-strand at a crossing.

is_subset_both_ways() Used in finding star graphs when performing *third Reidemeister moves*.

create_planar_graph_from_gauss_code() creates a plane graph from the Gauss code. Parallel edges and loops are omitted from the graph.

main_iterartive

Performs the same utility of the main script, with the added feature that it iterates over the global attribute *code_numbers* in the class **HardCodedKnots**.

check_gauss_code

For convenience and testing, we provided a module that identifies the Gauss code associated with a pair of signed planar graphs.

identify() It takes a Gauss code and identifies it, giving a knot tabulation (e.g. "4_1").

Additional Notes

The whole code is provided in the Appendix(C) with comments.

5 Results & Examples

"Mathematics is the art of giving the same name to different things."

— Henri Poincare

Our program worked with 24 knot diagrams, constructing their signed planar graphs, executing valid Reidemeister moves and comparing. We obtained the expected results from our program.

We will provide two examples to display the outputs of the program. And we start with the easiest one.

5.1 Example 1: The Trefoil

We start with the Trefoil, the Gauss code is: "1+c,2-c,3+c,1-c,2+c,3-c". The next figure is its knot diagram and our own colouring work.

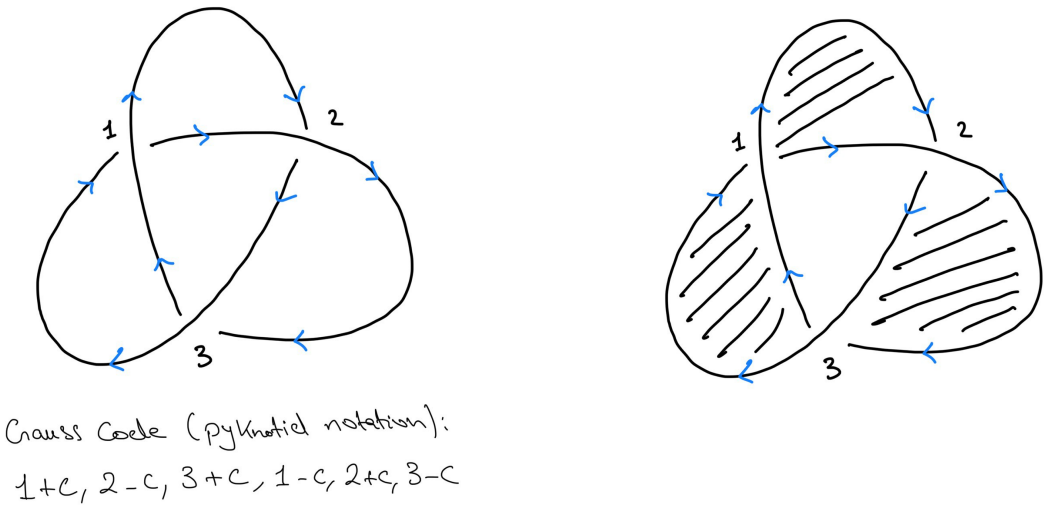


Figure 26: Our example trefoil with its Gauss code and shaded.

What follows is the signed planar graphs construction.

(a) Black Graph.

(b) White Graph.

Figure 27: The Signed Planar Graphs for the Trefoil.

We like to note that the above figure 26 is a ***reduced*** knot diagram. And we obtained its signed planar graphs in figure 27. In what follows, we will have a knot diagram with *reducible* crossings, obtain its signed planar graphs and reduce and compare our results.

5.2 Example 2: The Trefoil With three valid Reidemeister moves

Let's move to the case where there are three valid Reidemeister moves, one for each type. Here is the knot diagram and its respective Gauss Code: "1+c,2-c,3-c,6a,7-a,8-c,6+a,9+c,2+c,0+c,0-c,4-c,5-a,1-c,8+c,7+a,9-c,3+c,4+c,5+a"

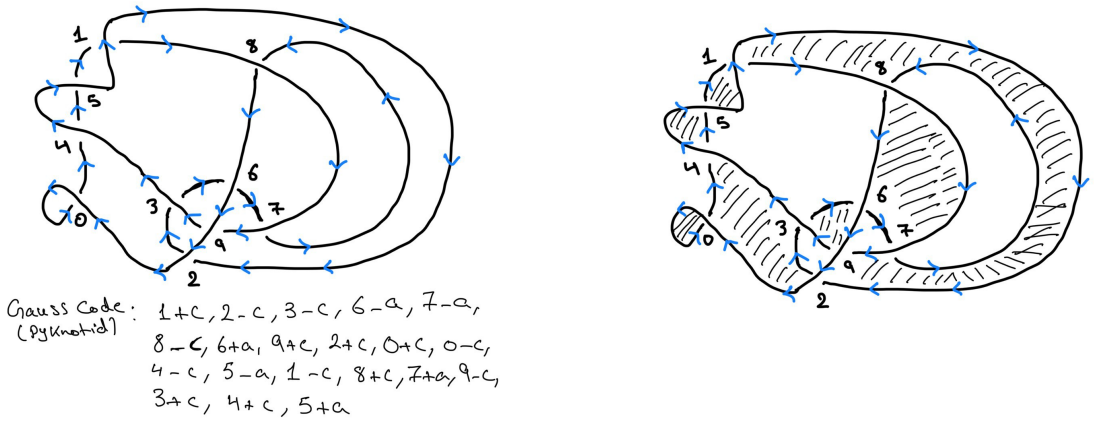


Figure 28: Our second example, the trefoil with three valid Reidemeister moves.

And now we construct the signed planar graphs.

(a) The Black Graph.

(b) The White Graph.

Figure 29: The Signed Planar Graphs for the Trefoil with three valid moves.

If we look closely at the figures above 29, we notice that in the black graph there is a spike node, namely "00". And in the white graph there is a loop in the node "12045". That pair of nodes constitute a Reidemeister move of type one. And our program (thankfully!!) performs the Reidemeister move and reduces the graph.

(a) Black Graph after performing RI.

(b) The White Graph after Performing RI

Figure 30: The Signed Planar Graphs after RI.

The above graphs are the result after performing one Reidemeister move of type one. We observe that both the spike and the loop have vanished. The new name nodes reflect that the crossing "0" is no longer there. The current Gauss Code is: "1+c,2-c,3-c,6-a,7-a,8-c,6+a,9+c,2+c,4-c,5-a,1-c,8+c,7+a,9-c,3+c,4+c,5+a".

Since, there are no more valid Reidemeister moves of type one. The program then searches for a valid Reidemeister move of type two. We were surprised (albeit pleasantly!) that the program choose the crossings "1" and "5" to perform the second Reidemeister move instead of our intention of the crossings of "4" and "5". Still, this is a valid move. The next figures 31 show the resulting operation. If we look closely at the black graph, we will find that the 2-path from node "18792" to "15" to "45" has been replaces by an edge between the nodes "29784" and "243". In the white graph, a pair of parallel edges between "1245" and "154368" have been replaced by one edge in the new graph between "3486". We also note that the naming of the nodes have changed to reflect that the crossings "1" and "5" are no more. The current Gauss Code is: "2-c,3-c,6-a,7-a,8-c,6+a,9+c,2+c,4-c,8+c,7+a,9-c,3+c,4+c".

There are still parallel edges in the black graph. However, the parallel edges between "29784" and "243" have the same sign. **As for the parallel edges between "29784" and "687", although the are of different signs, the relevant faces in the white graph are "3486", "679" and "78". The relevant faces do not pass one of our conditions for the second Reidemeister move, which that they only share two crossings in common ("6", "7" and "8" are common crossings between the faces.). The program concludes there are no more valid Reidemeister moves of type two and moves onward.**

(a) Black Graph after RII.

(b) White Graph after RII.

Figure 31: The Signed Planar Graphs after RII.

The cycle ("29784", "369", "243") in the black graph 31a and the star ("3486", "239", "24", "679") in white 31b fulfill the conditions for the Reidemeister move of type three. Below in figure 32 are the resulting graphs with the Gauss code: "8-c,9+c,4-c,8+c,9-c,4+c". We note that crossings "2", "3", "6", "7" disappear.

(a) Black Graph after RIII.

(b) White Graph after RIII.

Figure 32: The Signed Planar Graphs after RIII.

Although, black and white have swapped graphs. The pair in the above result is isomorphic to our previous signed planar graphs construction of the Trefoil in figure 27.

In the appendix(A), we display our other results.

6 Discussion & Future Work

"The thing that doesn't fit is the thing that's the most interesting: the part that doesn't go according to what you expected."

— Richard Feynman

6.1 Discussion

When we approached our problem, we build up blocks as we tested our implementation on different knot diagrams, accommodating for the different anomalies/corner-cases as they showed up. We believe this approach was heavily flawed. Deep insights are required for each process in the signed planar graph construction. What we have initially assumed would be relatively easy turned out to be major challenges. Loops and parallel edges specifically created a lot of problems regarding the derivation of the faces, their colouring, the edges and their signs.

While the program does achieve the expected results given the knot diagrams in table(4), we cannot claim that it is foolproof, it still has its limitations. We could summarize them as following:

- Works up to ten crossings.
- Approach lacks abstraction.
- More verification and tests needed.
- Dealing with less constraints on Reidemeister moves.
- Program construction could deviate from our construction.

Let's talk about each of these points.

Limit on the Number of Crossings

This limit comes from the nature in our approach to our program. We have used the data type "string" to utilize and manipulate the Gauss Code, naming nodes and crossings. A major issue occurs when we hit the 11^{th} crossing and we realized this issue very late in our coding phase. If we start counting from "0" up to "9", we face the issue when we want to add the next crossing which will take the label "10". Currently, the program is not able to differentiate between the labels "0", "1" and "10".

Approach lacks Abstraction

The design of our program could benefit from abstracting our implementation. In building up our code, we handled different knot diagrams and modified our code as we went along the way to identify errors and faults in our Algorithm's construction. Our design relies heavily on testing, validating our results and spotting anomalies. A more general approach that builds on deeper insights is required.

More Verification and Testing required

There are many Gauss Codes that we have not tested. The program *definitely* will fail in its constructions in many of them. We highlight that parallel edges and loops provide difficult and intriguing corner cases. More tests are needed to validate and update our code.

Less Constraints on Reidemeister Moves

We have altered our initial problem statement. It would be interesting to tackle the problem where the program tries to reduce the knot diagram by adding crossings and not just reduce them. It would also be interesting to take the graphs of knot diagram to another, even if both are not *reduced* knot diagrams.

Program's Construction deviates

While not necessarily a limitation, it does cause its own problems. In many cases, our program would choose a different exterior face/region while constructing the signed planar graphs from our own manual construction. Also, the labeling of the crossing from the Gauss Code heavily influence the program's choices, again causing deviations from our manual construction. This makes it harder to verify results, and keep track of what the program is doing. While this issue is of small significance as the number of crossings is not yet large enough. Issues will definitely rise later, and keeping this in the back of our minds as we recreate and modify our program will be of great benefit.

In the next subsection, we give our humble opinion on how future projects should handle our problem statement, and what could benefit upcoming undertakers (we also are definitely one of them!) as they create their own programs.

6.2 Future Work

The major feature that all of our proposed future works share, is the need for generalization. We believe the major aims of future contributions should be geared towards a more abstract approach. We highly recommend utilizing the more algebraic features of both knots and graphs. We would like to discuss the following ideas:

- Abstraction of processes.
- An Object Oriented Approach.

Abstraction of Processes

Many parts of our construction deal with other major areas in Mathematics. For example, the derivation of the faces from the shadow of the knot diagram could be considered from the perspective of planar graphs and their embedding. We highly recommend (mainly for ourselves!) a deeper study of this area in Graph Theory before tackling our problem statement. The labeling of edges with "+" or "-" could benefit from a deeper understanding of the Gauss Notation and similar algebraic structure. Each major step in the construction should be handled on a deeper level, shifting our perspective so that we underline all the nuisances and trickery that hide within the objective of the considered step.

An Object Oriented Approach

As discussed in the previous subsection, one of the major limitations is the number of crossings the program can work with. An OOP approach can easily eliminate such a drawback. Also, we envision objects specifically designed to handle the complex situations regarding parallel edges and loops as the interact with different elements in the construction. Two objects/classes that come to the forefront mind are; faces, crossings.

We also would recommend the review of the different software and solutions already available regarding graphs and knots to draw inspiration, such as *KnotAtlas*[13] and *Sage*[16]. However, up to the moment of writing this report and to our knowledge, there was no software that constructed the signed planar graphs of a knot diagram.

7 Conclusion

"It ain't over till it's over."

— Yogi Berra

In this report, we present our program dealing with Reidemeister moves on alternating knot diagrams and their signed planar graphs. We highlight theorems (1) and (2) as they provide the basis for our approach. We developed our program to take as input knot diagrams represented as a Gauss codes, one of them is a *reducible* knot diagram, the program then derives their signed planar graphs. It takes the signed planar graph of the **un***reduced* knot diagram, checks if there are any possible Reidemeister moves, performs them and modify its signed planar graphs. In the end, the program compares the pairs of signed planar graphs.

Our program was written using *Python*. We have also utilized external modules and libraries, two important mentions are "NetworkX" and "pyknotid". More details are found throughout the report and also in the References.

The details of the program were shown. We gave an overview the overall structure and algorithm. We then displayed the different classes and their roles. we also introduced the main script and other crucial modules. we walked through the process of the construction of the signed planar graphs and execution Reidemeister moves. We provided detailed examples of such constructions, and in the appendix(A), we display the different outputs on our knot diagram collection.

The program still has many areas that require development. We highlighted the approach in our development phase, and showed how our program might face difficulties with anomali6 es such loops or parallel edges in the signed planar graphs. More tests and different knot diagrams should be run on the program as validation tests, this would definitely optimize our results and discover bugs or errors. Also, a major drawback currently is the limit on knot diagrams to have a maximum of 10 crossings.

We provided insights regarding future work. Firstly, The adaptation of a more general approach than our trial-and-error methodology should be used. Secondly, utilizing OOP to overcome certain challenges, we add that OOP is an easy and straightforward solution for the ten crossing drawback.

References

[1] M. A. Armstrong. *Basic Toplology*. Springer, 1983.

[2] Charles Livingston. *Knot Theory*. The Mathematical Association Of America, 1993. ISBN: 0883850001.

[3] Colin C. Adams. *The Knot Book*. W. H. Freeman and Company, 1994. ISBN: 9780716742197.

[4] Chris Godsil and Gordon Royle. "Algebraic Graph Theory". In: Springer Science+Business Media, LLC, 2001. Chap. 16, 17, 18. ISBN: 9780387952208.

[5] Kunio Murasugi. *Knot theory and Its Applications*. Springer Science+Business Media, LLC, 2006. ISBN: 9780817647186.

[6] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[7] Jonathan Gross. *CS E6204 Lectures 8b and 9a Knots and Graphs*. 2010. URL: http://www.cs.columbia.edu/~cs6204/files/Lec8b,9a-Knots+Graphs.pdf. Accessed: 22.08.2023.

[8] Alexander J Taylor and other SPOCK contributors. *pyknotid knot identification toolkit*. https://github.com/SPOCKnots/pyknotid. 2017. Accessed: 22.08.2023.

[9] Kyle Miller. *What are Gauss and Dowker-Thistlethwaite codes?* URL: https://math.berkeley.edu/~kmill/2019_8_25/gauss_dt_codes.html. Aug. 2019. Accessed: 27.08.2023.

[10] Evan Adams. *What Was the Gordian Knot?* A&E Television Networks, URL = https://www.history.com/news/what-was-the-gordian-knot. History. Aug. 2023. Accessed: 31.08.2023.

[11] Anthony Bosman. Youtube. Math at Andrews University, Knot Theory. July 2023. URL: https://www.youtube.com/playlist?list=PLOROtRhtegr4c1H1JaWN1f6J_q1HdWZOY.

[12] Jeremy Green. *Gauss Code*. URL: https://www.math.toronto.edu/drorbn/Students/GreenJ/gausscode.html. Jan. 2023. Accessed: 31.08.2023.

[13] KnotAtlas. *The KnotAtlas*. Online. 2023. URL: https://knotatlas.org/.

[14] Charles Livingston and Allison H. Moore. *KnotInfo: Table of Knot Invariants*. URL: https://knotinfo.math.indiana.edu/. Aug. 2023.

[15] Henri Poincare. *BrainyQuote.com*. BrainyMedia Inc. 2023. URL: https://www.brainyquote.com/quotes/henri_poincare_208086. Accessed: 31.08.2023.

[16] The Sage Development Team. *SageMath: Open Source Mathematics Software*. Version 9.5. 2023. URL: https://www.sagemath.org/.

[17] Jean Toomer. *BrainyQuote.com*. BrainyMedia Inc. 2023. URL: https://www.brainyquote.com/quotes/jean_toomer_140337. Accessed: 27.08.2023.

[18] Cabiojinia. *The Gordian Knot*. URL: https://cabiojinia.com/it-makes-no-difference-how-they-are-loosed-the-gordian-knot/.

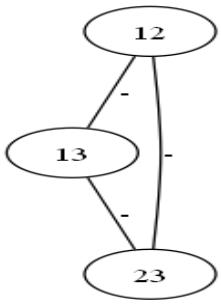
[19] Eric W. Weisstein. *Reduced Knot Diagram.* URL: https://mathworld.wolfram.com/ReducedKnotDiagram.html. From MathWorld–A Wolfram Web Resource. Accessed: 31.08.2023.

[20] Graph planarity tools. A wrapper to Boyer's (C) planarity algorithms: http://code.google.com/p/planarity/. Provides planarity testing, forbidden subgraph finding, and planar embeddings. Accessed: 31.08.2023.

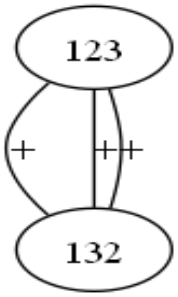
A Other Results

In the upcoming, We consider first the *irreducible* knot diagram and its signed planar graphs. These are the special outputs that we compare with the rest of our graph constructions, checking each knot diagram with its respective *irreducible* diagram.

A.1 The Trefoil



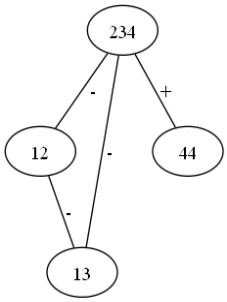
(a) Black Graph.



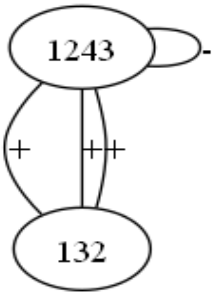
(b) White Graph.

Figure 33: The Signed Planar Graphs for the Trefoil.

A.1.1 The Trefoil with a Twist

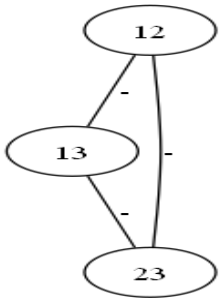


(a) Black Graph.

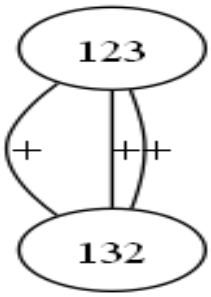


(b) White Graph.

Figure 34: initial graphs for the trefoil with a twist.



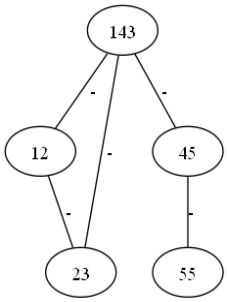
(a) Black Graph.



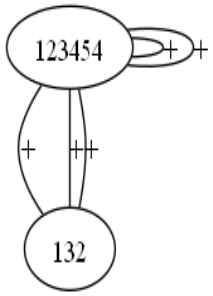
(b) White Graph.

Figure 35: final result for the trefoil with a twist.

A.1.2 The Trefoil with 2 Twists

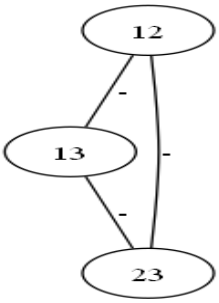


(a) Black Graph.

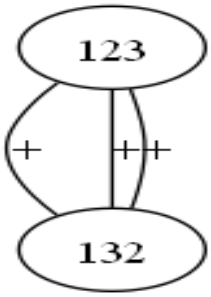


(b) White Graph.

Figure 36: initial graphs for the Trefoil with 2 twist.



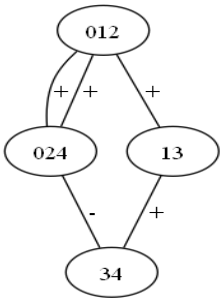
(a) Black Graph.



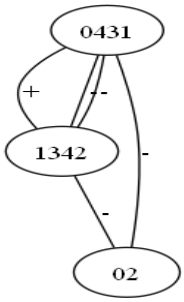
(b) White Graph.

Figure 37: final results for the Trefoil with 2 twist.

A.1.3 The Trefoil with a Poke

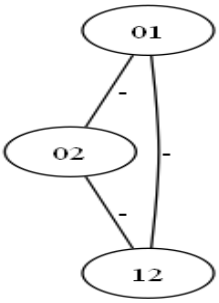


(a) Black Graph.

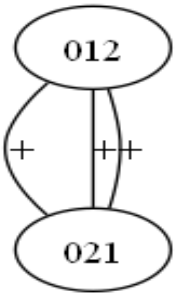


(b) White Graph.

Figure 38: inital graphs for the trefoil with a poke



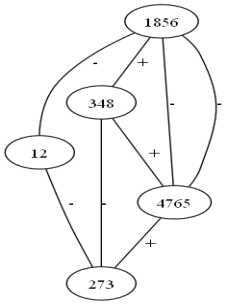
(a) Black Graph.



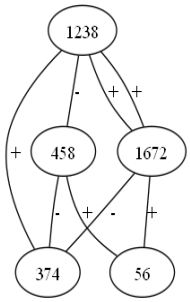
(b) White Graph.

Figure 39: final results for the trefoil with a poke

A.1.4 The Trefoil with a Twist then a Slide

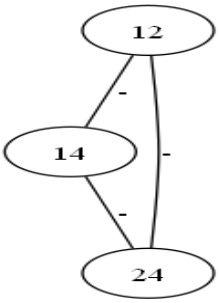


(a) Black Graph.

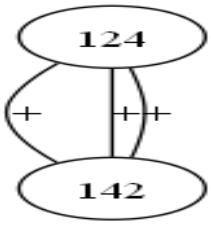


(b) White Graph.

Figure 40: initial graphs for the trefoil with a twist then a slide



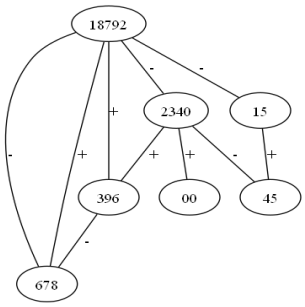
(a) Black Graph.



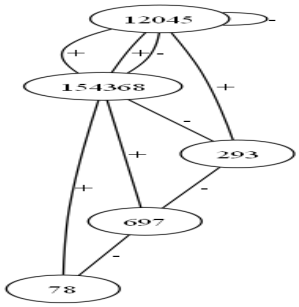
(b) White Graph.

Figure 41: final results for the trefoil with a twist then a slide

A.1.5 The Trefoil with a Twist then a Poke then a Slide

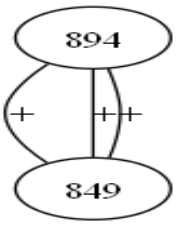


(a) The Black Graph.

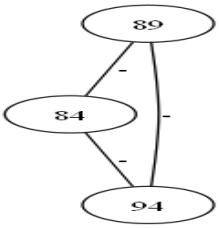


(b) The White Graph.

Figure 42: initial graphs for the trefoil with three valid moves.



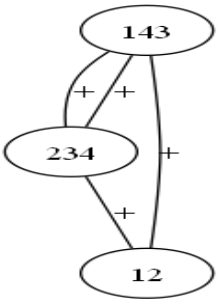
(a) Black Graph.



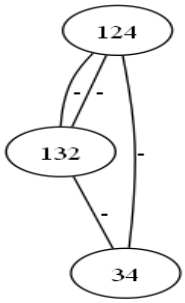
(b) White Graph.

Figure 43: final results of the trefoil with three valid moves.

A.2 The Figure Eight



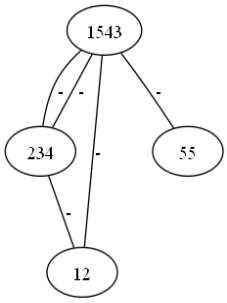
(a) Black Graph.



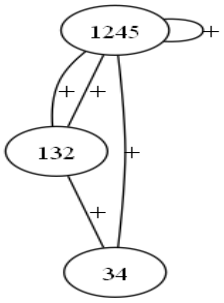
(b) White Graph.

Figure 44: Graphs for the Trefoil with a poke

A.2.1 Figure Eight with a Twist

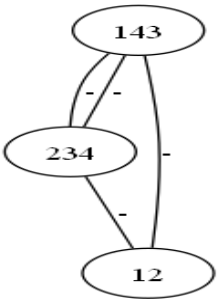


(a) Black Graph.

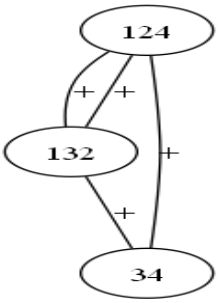


(b) White Graph.

Figure 45: initial graphs for the figure eight with a twist.



(a) Black Graph.



(b) White Graph.

Figure 46: final results for the figure eight with a twist.

A.2.2 Figure Eight with a poke

(a) Black Graph. (b) White Graph.

Figure 47: initial graphs for figure eight with a poke.

(a) Black Graph. (b) White Graph.

Figure 48: final results for figure eight with a poke.

A.2.3 Figure Eight with a Twist then a Poke

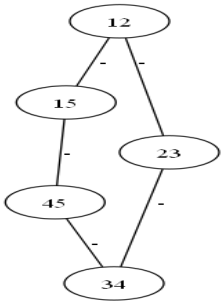
(a) Black Graph. (b) White Graph.

Figure 49: initial graphs for the figure eight with a twist then a poke.

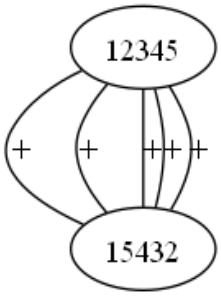
(a) Black Graph. (b) White Graph.

Figure 50: final results for the figure eight with a twist then a poke.

A.3 Star



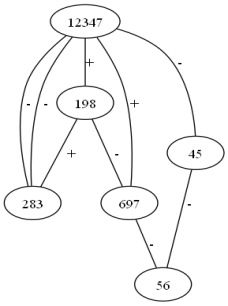
(a) Black Graph.



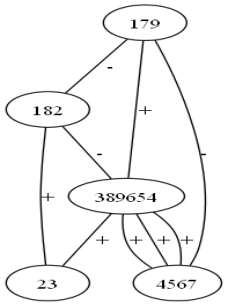
(b) White Graph.

Figure 51: Graphs for the star.

A.3.1 Star with a slide

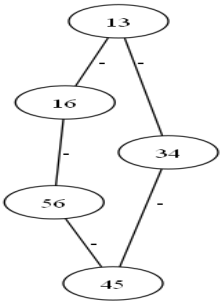


(a) Black Graph.

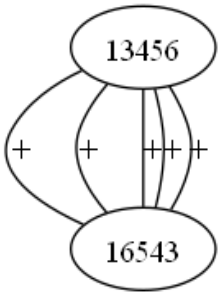


(b) White Graph.

Figure 52: initial graphs for star with a slide.



(a) Black Graph.



(b) White Graph.

Figure 53: final results for star with a slide.

A.3.2 Star with a Slide then a Twist

(a) Black Graph. (b) White Graph.

Figure 54: initial graphs for star with a slide then a twist.

(a) Black Graph. (b) White Graph.

Figure 55: final results for star with a slide then a twist.

A.4 5_2 Knot

(a) Black Graph. (b) White Graph.

Figure 56: Graphs for 5_2.

A.4.1 5_2 Knot with a Poke

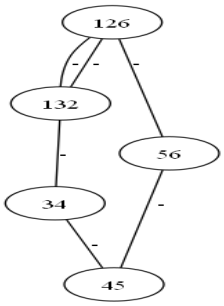
(a) Black Graph. (b) White Graph.

Figure 57: initial graphs for 5_2 with a poke.

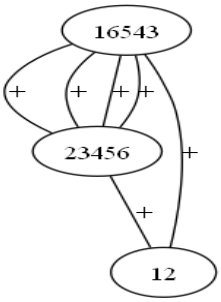
(a) Black Graph. (b) White Graph.

Figure 58: final results for 5_2 with a poke.

A.5 6_1 Knot



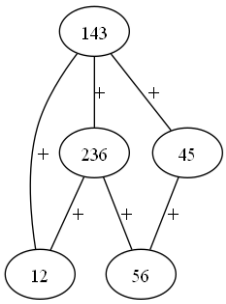
(a) Black Graph.



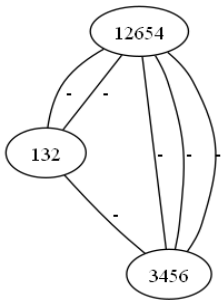
(b) White Graph.

Figure 59: Graphs for 6_1.

A.6 6_2 Knot



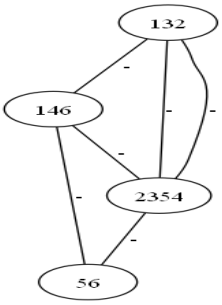
(a) Black Graph.



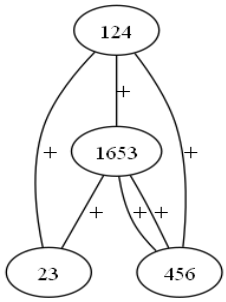
(b) White Graph.

Figure 60: Graphs for 6_2.

A.7 6_3 Knot



(a) Black Graph.



(b) White Graph.

Figure 61: Graphs for 5_3.

A.8 7_1 Knot

(a) Black Graph. (b) White Graph.

Figure 62: Graphs for 7_1.

A.9 7_2 Knot

(a) Black Graph. (b) White Graph.

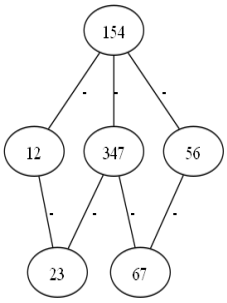
Figure 63: Graphs for 7_2.

A.10 7_3 Knot

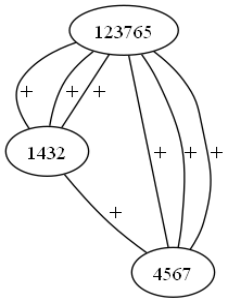
(a) Black Graph. (b) White Graph.

Figure 64: Graphs for 7_3.

A.11 7_4 Knot



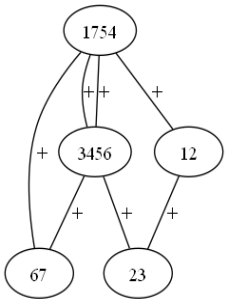
(a) Black Graph.



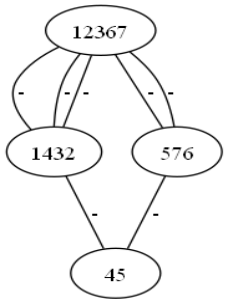
(b) White Graph.

Figure 65: Graphs for 7_4.

A.12 7_5 Knot



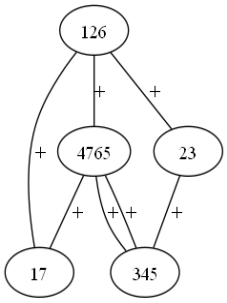
(a) Black Graph.



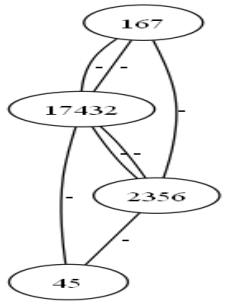
(b) White Graph.

Figure 66: Graphs for 7_5.

A.13 7_6 Knot



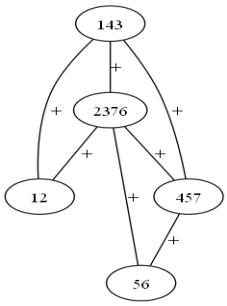
(a) Black Graph.



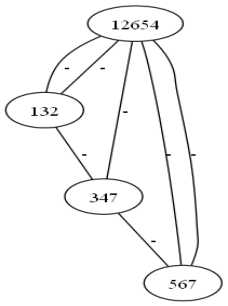
(b) White Graph.

Figure 67: Graphs for 7_6.

A.14 7_7 Knot



(a) Black Graph.

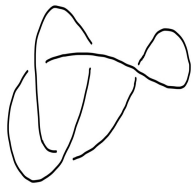
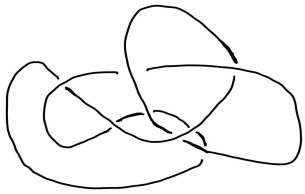
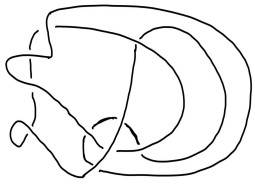


(b) White Graph.

Figure 68: Graphs for 7_7.

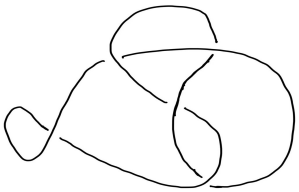
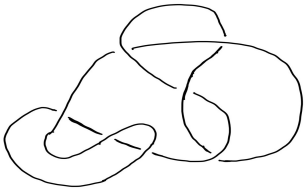
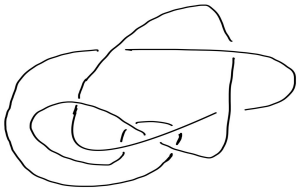
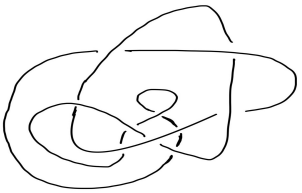
B HardCodedKnots Codes

Table 4: Hard-coded Knots Information

Knot	code_no	code_string	Gauss Code	Description	Knot Diagram
3_1	31	3_1 trefoil	1+c,2-c,3+c, 1-c,2+c,3-c	The trefoil	
3_1	311	3_1_1 trefoil_twist	1+c,2-c,3+c, 1-c,2+c,4+c, 4-c,3-c	The trefoil with one twist	
3_1	3111	3_1_1_1 trefoil_twist _twist	1+c,2-c,3+c, 4-a,5+a,5-a, 4+a,1-c,2+c, 3-c	The trefoil with two twist	
3_1	312	3_1_2 trefoil_poke	0-c,1+c,3-c, 4-a,2-c,0+c, 4+a,3+c,1-c, 2+c	The trefoil with a poke	
3_1	3113	3_1_1_3 trefoil_twist _slide	1+c,2-c,3+c, 4+c,5+a,6-a, 7-c,3-c,8-a, 5-a,6+a,1-c, 2+c,7+c,4-c, 8+a	The trefoil with a twist and a slide	
3_1	31123	3_1_1_2_3 trefoil_twist _poke_slide	1+c,2-c,3-c, 6a,7-a,8-c, 6+a,9+c,2+c, 0+c,0-c,4-c, 5-a,1-c,8+c, 7+a,9-c,3+c, 4+c,5+a	The trefoil with three valid moves.	

Continued on next page

Table 4 – Hardcoded Knots continued

Knot	code_no	code_string	Gauss Code	Description	Knot Diagram
4_1	41	4_1 figure_eight	1-c,2+c,3-a, 4+a,2-c,1+c, 4-a,3+a	The figure eight	
4_1	411	4_1_1 figure_eight _twist	1+a,2- a,3+a, 4-c,2+a,1-a, 5+a,5-a,4+c, 3-c	The figure eight with a twist	
4_1	4112	4_1_1_2 figure_eight _twist_poke	1+a,2-a,3+c, 4-c,2+a,1-a, 5+a,6+c,7+a, 5-a,6-c,7-a, 4+c,3-c	The figure eight with a twist	
5_1	51	5_1 star	1+c,2-c,3+c, 4-c,5+c,1-c, 2+c,3-c,4+c, 5-c	The star	
5_1	513	5_1_3 star_slide	1+c,2+c,3-c, 4+c,5-c,6+c, 7+a,1-c,8+c, 3+c,2-c,8-c, 9-a,7-a,4-c, 5+c,6-c,9+a	The star with a slide	
5_1	5131	5_1_3_1 star_slide _twist	1+c,2+c,3-c, 4+c,5-c,6+c, 7+a,1-c,8+c, 3+c,2-c,8-c, 0+c,0-c, 9-a,7-a,4-c, 5+c,6-c,9+a	The star with a slide and a twist	

Continued on next page

Table 4 – Hardcoded Knots continued

Knot	code_no	code_string	Gauss Code	Description	Knot Diagram
5_2	52	5_2	1+a,2-a,3+a,4-a,5+a,1-a,4+a,3-a,2+a, 5-a	The 5_2 knot	
5_2	522	5_2	1+c,2+a,3+c,4-c,5+c,6-c,7+c,1-c,4+c,3-c,2-a,7-c,6+c,5-c	The 5_2 knot	
6_1	61	6_1	1-c,2+c,3-a,4+a,5-a,6+a,2-c,1+c,6-a,5+a,4-a,3+a	The 6_1 knot	
6_2	62	6_2	1-c,2+c,3-a,4+a,5-a,6+a,2-c,1+c,4-a,5+a,6-a,3+a	The 6_2 knot	
6_3	63	6_3	1+a,2-a,3+a,1-a,4+c,5-c,6+c,4-c,2+a,3-a,5+c,6-c	The 6_3 knot	
7_1	71	7_1	1+c,2-c,3+c,4-c,5+c,6-c,7+c,1-c,2+c,3-c,4+c,5-c,6+c,7-c	The 7_1 knot	

Continued on next page

Table 4 – Hardcoded Knots continued

Knot	code_no	code_string	Gauss Code	Description	Knot Diagram
7_2	72	7_2	1+a,2-a,3+a,4-a,5+a,6-a,7+a,1-a,6+a,5-a,4+a,3-a,2+a,7-a	The 7_2 knot	
7_3	73	7_3	1+a,2-a,3+a,4-a,5+a,6-a,7+a,1-a,2+a,3-a,6+a,5-a,4+a, 7-a	The 7_3 knot	
7_4	74	7_4	1-a,2+a,3-a,4+a,5-a,6+a,7-a,3+a,2-a,1+a,4-a,7+a,6-a,5+a	The 7_4 knot	
7_5	75	7_5	1+a,2-a,3+a,4-a,5+a,6-a,7+a,5-a,4+a,1-a,2+a,3-a,6+a,7-a	The 7_5 knot	
7_6	76	7_6	1-a,2+c,3-c,4+a,5-a,3+c,2-c,6+a,7-a,1+a,6-a,5+a,4-a,7+a	The 7_6 knot	
7_7	77	7_7	1-c,2+c,3-a,4+a,5-c,6+c,2-c,1+c,4-a,7+a,6-c,5+c,7-a,3+a	The 7_7 knot	

C the Whole Code

C.1 main.py

```python
from SignedPlanarGraph import SignedPlanarGraph
from HardCodedKnots import HardCodedKnots
from ReidemeisterMoves import ReidemeisterMoves

hcg = HardCodedKnots(code_no=3113)

k_gauss = hcg.get_code_gauss()

sp_graph = SignedPlanarGraph(code_gauss = k_gauss)
sp_graph.create_and_draw_sp()


reid_sp = ReidemeisterMoves.reidemeister_main(sp_graph)

reid_sp = SignedPlanarGraph(code_gauss=reid_sp.code_gauss)
reid_sp.create_and_draw_sp()

#print(reid_sp.code_gauss)
```

C.2 check_gauss_code.py

```python
'''
We use this file to make sure our creations, our certain results
are correct by calculating manually the Gauss Code and checking if
    it
gives us the expected knot
'''

import pyknotid.representations as rep_
import numpy as np
import sys
import pyknotid.make as mk

gc = rep_.GaussCode("1+c,4-c,5+c,6-c,7+c,1-c,4+c,7-c,6+c,5-c")
rep = rep_.Representation(gc)

k = rep.space_curve(bool=False)
# k.plot()

print(k.identify())
```

C.3 main_iterative.py

```
from pyknotid.catalogue import get_knot
import networkx as nx
import matplotlib.pyplot as plt
from networkx.drawing.nx_agraph import to_agraph
import my_helpers as help
from SignedPlanarGraph import SignedPlanarGraph
from HardCodedKnots import HardCodedKnots
from ReidemeisterMoves import ReidemeisterMoves
import sys


for code_no in HardCodedKnots.code_numbers:

    hcg = HardCodedKnots(code_no = code_no)

    k_gauss =   hcg.get_code_gauss()


    sp_graph = SignedPlanarGraph(code_gauss = k_gauss)
    sp_graph.create_and_draw_sp()


    reid_sp = ReidemeisterMoves.reidemeister_main(sp_graph)

    reid_sp = SignedPlanarGraph(code_gauss=reid_sp.code_gauss)
    reid_sp.create_and_draw_sp()

    # Uncomment following if you would like to check after each run
       of the program
    # on different Gauss Code.
    # print(reid_sp.code_gauss)
    # input()
```

C.4 my_helpers.py

```
'''
This file is for methods and/or modules that will
be used across the whole project.
For easier access and better structure.
'''


import numpy as np
import collections as col
import pyknotid.representations as rep_
import networkx as nx


'''
Used to check whether two lists are equivalent w.r.t. Cyclic order.
Order is considered from left to right.
This method is mainly used to check that we do not have duplicate
   faces
When creating signed planar graphs.

ex 1:
l1 = [ 1, 2 , 3]
l2 = [3 , 1, 2]
is_cyclic_list(l1, l2) => TRUE

ex 2:
l1 = [ 1, 2 , 3]
l2 = [3 , 2, 1]
is_cyclic_list(l1, l2) => False
'''
def is_cyclic_list(l1, l2):
    if len(l1) != len(l2):
        return False
    if  not l1:
        return False
    elem = l1[0]
    for i in range(0,len(l1)):
        if elem == l2[i]:
            if l1 == l2[i : ] + l2[ : i]:
                return True
            return False
    return False

'''
To check if an ordered subset of list1
is contained in list2. Order is important!!
'''
def sublistExists(list1, list2):
    str_l1 = str(list1).strip("[]")
    str_l2 = str(list2).strip("[]")
    return str_l1 in str_l2

```

```
52  '''
53  used to create edges list
54  in order to create the edges between the nodes
55  in the dual graph.
56  '''
57  def edges_set_dual_creation(str_value):
58      if not str_value:
59          return
60
61      edges_set = []
62
63      for i in range(0, len(str_value) - 1):
64          edges_set.append(str_value[i: i+2])
65      edges_set.append(str_value[len(str_value) -1] + str_value[:1])
66
67      return edges_set
68
69  '''
70  To Create all edges as stings
71  each edge consists of two characters
72  the characters represent the nodes.
73  '''
74  def edges_set_creation(str_value):
75      if not str_value:
76          return
77
78      edges_set = []
79
80      for i in range(0, len(str_value) - 1):
81          edges_set.append(str_value[i: i+2])
82      edges_set.append(str_value[len(str_value) -1] + str_value[:1])
83
84      return edges_set
85
86  '''
87  This method is to take a Gauss Code provided from either
88  The library pyknotid, or from usual conventional mathematical
89  notation and modify to start counting from 0.
90  e.g. 1+c,2+a,.... => 0+c,1+a,...
91  '''
92
93  def code_gauss_from_math_to_comp(code_gauss):
94      code_gauss_splitted = str(code_gauss).split(",")
95
96      joker_code_gauss = ""
97
98      for gc in code_gauss_splitted:
99          crossing_no = int(gc[0])
100         crossing_no = crossing_no - 1
101         gc = str(crossing_no) + gc[1] + gc[2]
102         joker_code_gauss = joker_code_gauss + gc + ","
103
104     last_item = joker_code_gauss[len(joker_code_gauss)-1]
105     if last_item == ",":
```

```
106         joker_code_gauss = joker_code_gauss[:len(joker_code_gauss)
                -1]
107
108     return joker_code_gauss
109
110 '''
111 https://stackoverflow.com/questions/48688494/python-string-search-
        regardless-of-character-sequence
112 '''
113
114 def edge_contained(substr, str):
115     c1 = col.Counter(str)
116     c2 = col.Counter(substr)
117     return not(c2-c1)
118
119 '''
120 Create planar diagram from Gauss Code
121 '''
122
123 def create_planar_graph_from_gauss_code(code_gauss):
124
125     if not code_gauss:
126         raise Exception("Gauss Code is Empty")
127
128     str_gc = str(code_gauss)
129     code_gauss_splitted  = str_gc.split(",")
130
131     if len(code_gauss_splitted) % 2 != 0:
132         raise Exception("Gauss Code Is invalid")
133
134     # creating a knot from the gauss code
135     # to make sure that the gauss code
136     # is a valid knot
137     joker_gc = rep_.GaussCode(str(code_gauss))
138     rep = rep_.Representation(joker_gc)
139     k = rep.space_curve(bool=True)
140
141     graph = nx.MultiGraph()
142
143     i = 0
144     while i < len(code_gauss_splitted):
145         gc = code_gauss_splitted[i]
146         if i == len(code_gauss_splitted) - 1:
147             gc_next = code_gauss_splitted[0]
148         else:
149             gc_next = code_gauss_splitted[i+1]
150         graph.add_edge(int(gc[0]),int(gc_next[0]))
151         i+=1
152
153     return graph
154
155 '''
156 To get the face that bounds the over and understrand
157 '''
```

```
158
159 def get_bounded_face(edges, faces):
160     faces_list = []
161     if len(edges[0]) == 2 and edges[0][0] == edges[0][1]:
162         edges[0] = edges[0][0]
163     if len(edges[1]) == 2 and edges[1][0] == edges[1][1]:
164         edges[1] = edges[1][0]
165
166     for face in faces:
167         if edge_contained(edges[0], face):
168             if edge_contained(edges[1], face):
169                 faces_list.append(face)
170
171     if len(faces_list) == 0:
172         return None
173
174     no_of_edges = []
175     for face in faces_list:
176         no_of_edges.append(len(face))
177
178     bounded_face_index = no_of_edges.index(min(no_of_edges))
179     return faces_list[bounded_face_index]
180
181 def is_subset_both_ways(str_1, str_2):
182     chars_1 = set(str_1)
183     chars_2 = set(str_2)
184     return len(str_1) == len(str_2) and chars_1.issubset(chars_2)
        and chars_2.issubset(chars_1)
```

C.5 HardCodedKnots.py

```
from pyknotid.catalogue import get_knot
import pyknotid.representations as rep_

class HardCodedKnots:

    code_numbers = [31, 311, 3111, 312, 3113, 41, 411, 4112, 412,
       51, 513, 5131, 52, 522, 61, 62, 63,
                    71, 72, 73, 74, 75, 76, 77]


    def __init__(self, code_no = None, code_string = None):

        if code_no == 31 or code_string == "trefoil" or code_string
           == "3_1":
            k = get_knot('3_1').space_curve()
            self.code_gauss = k.gauss_code()
            return
        if code_no == 311 or code_string == "trefoil_twist" or
           code_string == "3_1_1":
            self.code_gauss = rep_.GaussCode("1+c,2-c,3+c,1-c,2+c,4+
               c,4-c,3-c")
            return

        if code_no == 3111 or code_string == "trefoil_twist_twist"
           or code_string == "3_1_1_1":
            self.code_gauss = rep_.GaussCode("1+c,2-c,3+c,4-a,5+a,5-
               a,4+a,1-c,2+c,3-c")
            return

        if code_no == 312 or code_string == "trefoil_poke" or
           code_string == "3_1_2":
            self.code_gauss = rep_.GaussCode("0-c,1+c,3-c,4-a,2-c,0+
               c,4+a,3+c,1-c,2+c")
            return

        if (code_no == 3113 or code_string == "trefoil_twist_slide"
                or code_string == "3_1_1_3"):
            self.code_gauss = rep_.GaussCode("1+c,2-c,3+c,4+c,5+a,6-
               a,7-c,3-c,8-a,5-a,6+a,1-c,2+c,7+c,4-c,8+a")

        if code_no == 41 or code_string == "figure_eight" or
           code_string == "4_1":
            k = get_knot('4_1').space_curve()
            self.code_gauss = k.gauss_code()

        if code_no == 411 or code_string == "figure_eight_twist" or
           code_string == "4_1_1":
            self.code_gauss = rep_.GaussCode("1+a,2-a,3+a,4-c,2+a,1-
               a,5+a,5-a,4+c,3-c")
            return

```

```
40          if code_no == 4112 or code_string == "
               figure_eight_twist_poke" or code_string == "4_1_1_2":
41              self.code_gauss = rep_.GaussCode("1+a,2-a,3+c,4-c,2+a,1-
                   a,5+a,6+c,7+a,5-a,6-c,7-a,4+c,3-c")
42              return
43
44          if code_no == 412 or code_string == "figure_eight_poke" or
               code_string == "4_1_2":
45              self.code_gauss = rep_.GaussCode("1-c,2-a,3+c,4-c,2+a,5-
                   a,6+a,1+c,4+c,3-c,5+a,6-a")
46              return
47
48          if code_no == 51 or code_string == "star" or code_string ==
               "5_1":
49              k = get_knot("5_1").space_curve()
50              self.code_gauss = k.gauss_code()
51              return
52
53          if code_no == 513 or code_string == "star_slide" or
               code_string == "5_1_3":
54              self.code_gauss = rep_.GaussCode("1+c,2+c,3-c,4+c,5-c,6+
                   c,7+a,1-c,8+c,3+c,2-c,8-c,9-a,7-a,4-c,5+c,6-c,9+a")
55              return
56
57          if code_no == 5131 or code_string == "star_slide_twist" or
               code_string == "5_1_3_1":
58              self.code_gauss = rep_.GaussCode("1+c,2+c,3-c,4+c,5-c,6+
                   c,7+a,1-c,8+c,3+c,2-c,8-c,0+c,0-c,9-a,7-a,4-c,5+c,6-c
                   ,9+a")
59              return
60
61          if code_no == 52 or code_string == "5_2":
62              k = get_knot("5_2").space_curve()
63              self.code_gauss = k.gauss_code()
64              return
65
66          if code_no == 522 or code_string == "5_2_2":
67              self.code_gauss = rep_.GaussCode("1+c,2+a,3+c,4-c,5+c,6-
                   c,7+c,1-c,4+c,3-c,2-a,7-c,6+c,5-c")
68              return
69
70          if code_no == 61 or code_string == "6_1":
71              k = get_knot("6_1").space_curve()
72              self.code_gauss = k.gauss_code()
73              return
74
75          if code_no == 62 or code_string == "6_2":
76              k = get_knot("6_2").space_curve()
77              self.code_gauss = k.gauss_code()
78              return
79
80          if code_no == 63 or code_string == "6_3":
81              k = get_knot("6_3").space_curve()
82              self.code_gauss = k.gauss_code()
```

```
83              return
84
85          if code_no == 71 or code_string == "7_1":
86              k = get_knot("7_1").space_curve()
87              self.code_gauss = k.gauss_code()
88              return
89
90          if code_no == 72 or code_string == "7_2":
91              k = get_knot("7_2").space_curve()
92              self.code_gauss = k.gauss_code()
93              return
94
95          if code_no == 73 or code_string == "7_3":
96              k = get_knot("7_3").space_curve()
97              self.code_gauss = k.gauss_code()
98              return
99
100         if code_no == 74 or code_string == "7_4":
101             k = get_knot("7_4").space_curve()
102             self.code_gauss = k.gauss_code()
103             return
104
105         if code_no == 75 or code_string == "7_5":
106             self.code_gauss = rep_.GaussCode("1+a,2-a,3+a,4-a,5+a,6-
                    a,7+a,5-a,4+a,1-a,2+a,3-a,6+a,7-a")
107             return
108
109         if code_no == 76 or code_string == "7_6":
110             k = get_knot("7_6").space_curve()
111             self.code_gauss = k.gauss_code()
112             return
113
114         if code_no == 77 or code_string == "7_7":
115             k = get_knot("7_7").space_curve()
116             self.code_gauss = k.gauss_code()
117             return
118
119     def get_code_gauss(self):
120         return self.code_gauss
```

C.6 SignedPlanarGraph.py

```
import networkx as nx
import my_helpers as help
import copy


class SignedPlanarGraph:

    drawing_number = 0

    def __init__(self, black_graph_data = None, white_graph_data = None, knot = None,
                       code_gauss = None, signed_black = None, signed_white = None):
        if black_graph_data:
            self.black_graph = nx.MultiGraph(black_graph_data)
        else:
            self.black_graph = nx.MultiGraph()
        if white_graph_data:
            self.white_graph = nx.MultiGraph(white_graph_data)
        else:
            self.white_graph = nx.MultiGraph()
        if knot:
            self.knot = knot
        else:
            self.knot = None
        if code_gauss:
            self.code_gauss = code_gauss
        else:
            self.code_gauss = None
        if signed_black:
            self.signed_black = signed_black
        else:
            self.signed_black = []
        if signed_white:
            self.signed_white = signed_white
        else:
            self.signed_white = []

    def __eq__(self, other):

        if str(self.code_gauss) == str(other.code_gauss):
            if nx.is_isomorphic(self.black_graph, other.black_graph):
                if nx.is_isomorphic(self.white_graph, other.white_graph):
                    if self.signed_black == other.signed_black:
                        if self.signed_white == other.signed_white:
                            return True
        return False


    def __ne__(self, other):
```

```
49          return not self.__eq__(other)
50
51      def create_nodes_from_faces(self, graph):
52
53          (check, planar_graph) = nx.check_planarity(graph)
54          faces_first_run = SignedPlanarGraph.get_faces_list(
                planar_graph)
55          faces = SignedPlanarGraph.add_loops_face_list(graph,
                faces_first_run)
56          outer_face = SignedPlanarGraph.get_longest_region(faces)
57          outer_regions = SignedPlanarGraph.get_outer_regions(
                outer_face, faces)
58          [black, white] = SignedPlanarGraph.shade_faces_new(
                outer_regions, faces, outer_face)
59
60          black_nodes = []
61          for b in black:
62              black_nodes.append(str(b).strip("[]").replace(",␣", ""))
63          self.black_graph.add_nodes_from(black_nodes)
64
65          white_nodes = []
66          for w in white:
67              white_nodes.append(str(w).strip("[]").replace(",␣", ""))
68          self.white_graph.add_nodes_from(white_nodes)
69
70          self.black_loops = []
71          for b in self.black_graph.nodes():
72              if len(b) == 2:
73                  if b[0] == b[1]:
74                      self.black_loops.append(b)
75
76          self.white_loops = []
77          for w in self.white_graph.nodes():
78              if len(w) == 2:
79                  if w[0] == w[1]:
80                      self.white_loops.append(w)
81
82
83
84      def create_edges_after_nodes(self):
85
86          nodes = self.black_graph.nodes()
87          nodes = list(nodes)
88          edges = []
89
90          i = 0
91          while i < len(nodes):
92              j = i + 1
93              while j < len(nodes):
94                  if len(nodes[j]) == 2:
95                      if nodes[j][0] == nodes[j][1]:
96                          if nodes[j][0] in nodes[i]:
97                              edges.append((nodes[i], nodes[j]))
98                              j += 1
```

```
                        continue
                for ch in nodes[j]:
                    if ch in nodes[i]:
                        edges.append((nodes[i], nodes[j]))
                j +=1
            white_loops = copy.deepcopy(self.white_loops)
            for white_loop in white_loops:
                if white_loop[0] in nodes[i]:
                    edges.append((nodes[i], nodes[i]))
                    the_last = white_loop[0]
                    the_last_index = nodes[i].find(the_last)
                    m = (the_last_index - 1) % len(nodes[i])
                    n = (the_last_index + 1) % len(nodes[i])
                    while True:
                        if nodes[i][m] == nodes[i][n]:
                            edges.append((nodes[i], nodes[i]))
                            another = the_last + "" + nodes[i][m]
                            self.white_loops.append(another)
                            m = (m - 1) % len(nodes[i])
                            n = (n  + 1) % len(nodes[i])
                            continue
                        break
            i += 1


        self.black_graph.add_edges_from(edges)

        nodes = self.white_graph.nodes()
        nodes = list(nodes)
        edges = []

        i = 0
        while i < len(nodes):
            j = i + 1
            while j < len(nodes):
                if len(nodes[j]) == 2:
                    if nodes[j][0] == nodes[j][1]:
                        if nodes[j][0] in nodes[i]:
                            edges.append((nodes[i], nodes[j]))
                            j += 1
                            continue
                for ch in nodes[j]:
                    if ch in nodes[i]:
                        edges.append((nodes[i], nodes[j]))
                j +=1
            black_loops = copy.deepcopy(self.black_loops)
            for black_loop in black_loops:
                if black_loop[0] in nodes[i]:
                    edges.append((nodes[i], nodes[i]))
                    the_last = black_loop[0]
                    the_last_index = nodes[i].find(the_last)
                    m = (the_last_index - 1) % len(nodes[i])
                    n = (the_last_index + 1) % len(nodes[i])
                    while True:
```

```
153                         if nodes[i][m] == nodes[i][n]:
154                             edges.append((nodes[i], nodes[i]))
155                             another = the_last + "" + nodes[i][m]
156                             self.black_loops.append(another)
157                             m = (m - 1) % len(nodes[i])
158                             n = (n  + 1) % len(nodes[i])
159                             continue
160                         break
161             i += 1
162 
163         self.white_graph.add_edges_from(edges)
164 
165     @staticmethod
166     def get_faces_list(planar_graph):
167         graph = planar_graph
168         faces_list = []
169         for node_1 in graph.nodes():
170             for node_2 in graph.nodes():
171                 if node_2 not in graph[node_1]:
172                     continue
173                 faces_list.append(graph.traverse_face(node_1, node_2
                        ))
174 
175         i = 0
176         while i < len(faces_list):
177             j = i + 1
178             while j < len(faces_list):
179                 if help.is_cyclic_list(faces_list[i], faces_list[j])
                        :
180                     faces_list.remove(faces_list[j])
181                     j = j - 1
182                 j =  j + 1
183             i = i + 1
184         return faces_list
185 
186     @staticmethod
187     def get_longest_region(faces_list):
188         no_edges_in_each_face = []
189 
190         for face in faces_list:
191             no_edges_in_each_face.append(len(face))
192 
193         outer_region_index = no_edges_in_each_face.index(max(
                no_edges_in_each_face))
194         return faces_list[outer_region_index]
195 
196 
197     #Add loops as faces
198     @staticmethod
199     def add_loops_face_list(orig_graph, faces_list):
200         graph = orig_graph
201         edges = graph.edges()
202         edges_list = list(edges)
203         edge_count = {}
```

```
204             for e in edges:
205                 e_str = str(e).strip("()").replace(", ", "")
206                 e_str = e_str.replace("'", "")
207                 rev_e_str = e_str[::-1]
208                 if e_str not in edge_count and rev_e_str not in
                        edge_count:
209                     edge_count[e_str] = 1
210                     if e_str[0] == e_str[1]:
211                         edge_count[e_str] += 1
212                 elif e_str in edge_count:
213                     edge_count[e_str] += 1
214                 else:
215                     edge_count[rev_e_str] += 1
216             for e in edge_count:
217                 if edge_count[e] > 1:
218                     face = [int(e[0:1]), int(e[1:])]
219                     for i in range(0, edge_count[e] - 1):
220                         faces_list.append(face)
221             return faces_list
222 
223         @staticmethod
224         def get_outer_regions(longest_region, faces_list):
225 
226             outer_regions = []
227             loops = []
228 
229             for face in faces_list:
230                 if face != longest_region:
231                     if SignedPlanarGraph.check_same_edge_region(face,
                            longest_region):
232                         outer_regions.append(face)
233                         check_if_loop = str(face).strip("[]").replace(",
                                ", "")
234                         if len(check_if_loop) == 2:
235                             loops.append(face)
236             for loop in loops:
237                 if loop[0] == loop[1]:
238                     continue
239                 for region in outer_regions:
240                     if loop != region:
241                         if SignedPlanarGraph.check_same_edge_region(loop
                                , region):
242                             outer_regions.remove(region)
243             return outer_regions
244 
245         @staticmethod
246         def shade_faces_new(outer_reigons,faces_list, outer_face):
247 
248             black_shades = outer_reigons
249             white_shades = [outer_face]
250 
251             coloured = black_shades + white_shades
252 
```

```
253             ple_faces = [str(face[0]) + str(face[1]) for face in
                    faces_list if len(face) == 2]
254
255             nbr_black_turn = True
256             got_coloured= outer_reigons
257
258
259             skipped = False
260             while len(coloured) != len(faces_list):
261                 next_got_coloured = []
262                 if nbr_black_turn:
263                     for face in got_coloured:
264                         for next_face in faces_list:
265                             if next_face not in coloured:
266                                 if face != next_face:
267                                     if SignedPlanarGraph.
                                            check_same_edge_region(face,
                                            next_face):
268                                         if len(next_face) > 2:
269                                             [cond, edge] =
                                                    SignedPlanarGraph.
                                                    check_same_edge_region(
                                                    face, next_face)
270                                             if edge in ple_faces or edge
                                                    [::-1] in ple_faces:
271                                                 skipped = True
272                                                 continue
273                                         coloured.append(next_face)
274                                         white_shades.append(next_face)
275                                         next_got_coloured.append(
                                                next_face)
276                     nbr_black_turn = False
277                     got_coloured = next_got_coloured
278                 else:
279                     for face in got_coloured:
280                         for next_face in faces_list:
281                             if next_face not in coloured:
282                                 if face != next_face:
283                                     if SignedPlanarGraph.
                                            check_same_edge_region(face,
                                            next_face):
284                                         if len(next_face) > 2:
285                                             [cond, edge] =
                                                    SignedPlanarGraph.
                                                    check_same_edge_region(
                                                    face, next_face)
286                                             if edge in ple_faces or edge
                                                    [::-1] in ple_faces:
287                                                 continue
288                                         coloured.append(next_face)
289                                         black_shades.append(next_face)
290                                         next_got_coloured.append(
                                                next_face)
291                     nbr_black_turn = True
```

```
292                 got_coloured = next_got_coloured
293             if skipped and len(got_coloured) == 0 and len(coloured)
                    != len(faces_list):
294                 remaining = [face for face in faces_list if face not
                        in coloured]
295                 for face in remaining:
296                     jump = False
297                     for next_face in coloured:
298                         if SignedPlanarGraph.check_same_edge_region(
                                face, next_face):
299                             if next_face in black_shades:
300                                 white_shades.append(face)
301                             elif next_face in white_shades:
302                                 black_shades.append(face)
303                             coloured.append(face)
304                             jump = True
305                         if jump:
306                             break
307
308         return black_shades, white_shades
309
310     @staticmethod
311     def check_same_edge_region(face_1, face_2):
312         face_str_1 = str(face_1).strip("[]").replace(",␣", "")
313         face_str_2 = str(face_2).strip("[]").replace(",␣", "")
314         subsets = help.edges_set_creation(face_str_1)
315         for subset in subsets:
316             rev_face_str_2 = face_str_2[::-1]
317             if help.sublistExists(subset, face_str_2):
318                 return True, subset
319             if help.sublistExists(subset, rev_face_str_2):
320                 return True, subset
321             last_edge = face_str_2[len(face_str_2)-1:] + face_str_2
                    [:1]
322             rev_last_edge = last_edge[::-1]
323             if help.sublistExists(subset, last_edge):
324                 return True, subset
325             if help.sublistExists(subset, rev_last_edge):
326                 return True, subset
327             if subset[0] == subset[1]:
328                 if help.sublistExists(subset[0], face_str_2):
329                     return True, subset
330         return False
331
332     def draw_sp_graph(self, path=None):
333             if path:
334                 pass
335             else:
336                 from networkx.drawing.nx_agraph import to_agraph
337
338                 A = to_agraph(self.black_graph)
339                 path_final = "pyknotid_helali/my_code_1/images/
                        medial_graph_black" + str(SignedPlanarGraph.
                        drawing_number) + ".png"
```

```
340                 A.draw( path_final, prog='dot')
341                 SignedPlanarGraph.drawing_number += 1
342
343                 pos = nx.planar_layout(self.white_graph)
344                 A = to_agraph(self.white_graph)
345                 path_final = "pyknotid_helali/my_code_1/images/
                        medial_graph_white" + str(SignedPlanarGraph.
                        drawing_number) + ".png"
346                 A.draw( path_final, prog='dot')
347                 SignedPlanarGraph.drawing_number += 1
348
349
350     def draw_sp_graph_sign(self, path=None):
351             if path:
352                 pass
353             else:
354                 from networkx.drawing.nx_agraph import to_agraph
355
356                 joker_graph = nx.MultiGraph()
357                 joker_signs = copy.deepcopy(self.signed_black)
358
359
360                 for sign in joker_signs:
361                     joker_graph.add_edge(sign[0][0],sign[0][1],
                            label = sign[1])
362
363
364                 A = to_agraph(joker_graph)
365                 path_final = "pyknotid_helali/my_code_1/images/
                        medial_graph_black" + str(SignedPlanarGraph.
                        drawing_number) + ".png"
366                 A.draw( path_final, prog='dot')
367                 SignedPlanarGraph.drawing_number += 1
368
369                 joker_graph = nx.MultiGraph()
370                 joker_signs = copy.deepcopy(self.signed_white)
371
372                 for sign in joker_signs:
373                     joker_graph.add_edge(sign[0][0],sign[0][1],
                            label = sign[1])
374
375                 A = to_agraph(joker_graph)
376                 path_final = "pyknotid_helali/my_code_1/images/
                        medial_graph_white" + str(SignedPlanarGraph.
                        drawing_number) + ".png"
377                 A.draw( path_final, prog='dot')
378                 SignedPlanarGraph.drawing_number += 1
379
380
381
382     def sign_sp(self):
383         if self.code_gauss:
384             code_gauss_splitted = str(self.code_gauss).split(",")
385
```

```
386                 black_loops = copy.deepcopy(self.black_loops)
387                 white_loops = copy.deepcopy(self.white_loops)
388
389                 already_signed_black = []
390                 for b_edge in self.black_graph.edges():
391                     if b_edge in already_signed_black:
392                         if b_edge[0] != b_edge[1]:
393                             continue
394                     already_signed_black.append(b_edge)
395                     remove_brackets = str(b_edge).strip("()").replace("'
                        ", "")
396                     nodes = remove_brackets.split(",␣")
397                     shift = False
398                     if nodes[0] != nodes[1]:
399                         for char in nodes[0]:
400                             if char in nodes[1]:
401                                 code_gauss_pair = []
402                                 i = 0
403                                 while i < len(code_gauss_splitted):
404                                     gc = code_gauss_splitted[i]
405                                     if gc[0] == char:
406                                         the_gc = gc
407                                         if i == len(code_gauss_splitted)
                                            - 1:
408                                             code_gauss_pair.append(
                                                code_gauss_splitted[0])
409                                         else:
410                                             code_gauss_pair.append(
                                                code_gauss_splitted[i+1])
411                                     i = i + 1
412                                 if len(code_gauss_pair) != 2:
413                                     raise Exception("Either␣your␣
                                        algorithm␣doesn't␣work␣or␣the␣
                                        Gauss␣code␣is␣wrong")
414
415                                 edges = []
416                                 edges.append(char + code_gauss_pair
                                    [0][0])
417                                 edges.append(char + code_gauss_pair
                                    [1][0])
418
419                                 if len(edges[0]) == 2 and edges[0][0] ==
                                        edges[0][1]:
420                                     shift = True
421                                 if len(edges[1]) == 2 and edges[1][0] ==
                                        edges[1][1]:
422                                     shift = True
423
424                                 faces= list(self.black_graph.nodes()) +
                                    list(self.white_graph.nodes())
425                                 the_face = help.get_bounded_face(edges,
                                    faces)
426                                 if the_face is not None:
```

```
427                                 if the_face in self.black_graph.
                                        nodes():
428                                     if the_gc[2] == "a":
429                                         if not shift:
430                                             self.signed_black.append
                                                ((b_edge, "+", char))
431                                         else:
432                                             self.signed_black.append
                                                ((b_edge, "-", char))
433                                     elif the_gc[2] == "c":
434                                         if not shift:
435                                             self.signed_black.append
                                                ((b_edge, "-", char))
436                                         else:
437                                             self.signed_black.append
                                                ((b_edge, "+", char))
438                                 elif the_face in self.white_graph.
                                        nodes():
439                                     if the_gc[2] == "a":
440                                         if not shift:
441                                             self.signed_black.append
                                                ((b_edge, "-", char)
                                                )
442                                         else:
443                                             self.signed_black.append
                                                ((b_edge, "+", char)
                                                )
444                                     elif the_gc[2] == "c":
445                                         if not shift:
446                                             self.signed_black.append
                                                ((b_edge, "+", char))
447                                         else:
448                                             self.signed_black.append
                                                ((b_edge, "-", char))
449
450                     elif nodes[0] == nodes[1]:
451                         crossed_at = None
452                         for white_loop in white_loops:
453                             if white_loop[0] in nodes[0]:
454                                 crossed_at = white_loop[0]
455                                 white_loops.remove(white_loop)
456                                 break
457                         if crossed_at == None:
458                             raise Exception("Loop␣does␣not␣exsist!!!␣
                                    Edge␣is␣not␣correct!!!")
459                         for gc in code_gauss_splitted:
460                             if gc[0] == crossed_at:
461                                 if gc[2] == "a":
462                                     self.signed_black.append((b_edge, "+
                                            ", crossed_at))
463                                 elif gc[2] == "c":
464                                     self.signed_black.append((b_edge, "-
                                            ", crossed_at))
465                                 break
```

```
466
467             already_signed_white = []
468             for w_edge in self.white_graph.edges():
469                 if w_edge in already_signed_white:
470                     if w_edge[0] != w_edge[1]:
471                         continue
472                 already_signed_white.append(w_edge)
473                 remove_brackets = str(w_edge).strip("()").replace("'
                        ", "")
474                 nodes = remove_brackets.split(",␣")
475                 shift = False
476                 if nodes[0] != nodes[1]:
477                     for char in nodes[0]:
478                         if char in nodes[1]:
479                             code_gauss_pair = []
480                             i = 0
481                             while i < len(code_gauss_splitted):
482                                 gc = code_gauss_splitted[i]
483                                 if gc[0] == char:
484                                     the_gc = gc
485                                     if i == len(code_gauss_splitted)
                                            - 1:
486                                         code_gauss_pair.append(
                                            code_gauss_splitted[0])
487                                     else:
488                                         code_gauss_pair.append(
                                            code_gauss_splitted[i+1])
489                                 i = i+1
490                             if len(code_gauss_pair) != 2:
491                                 raise Exception("Either␣your␣
                                        algorithm␣doesn't␣work␣or␣the␣
                                        Gauss␣code␣is␣wrong")
492
493                             edges = []
494                             edges.append(char + code_gauss_pair
                                    [0][0])
495                             edges.append(char + code_gauss_pair
                                    [1][0])
496
497                             if len(edges[0]) == 2 and edges[0][0] ==
                                     edges[0][1]:
498                                 shift = True
499                             if len(edges[1]) == 2 and edges[1][0] ==
                                     edges[1][1]:
500                                 shift = True
501
502                             faces= list(self.black_graph.nodes()) +
                                    list(self.white_graph.nodes())
503                             the_face = help.get_bounded_face(edges,
                                    faces)
504                             if the_face is not None:
505                                 if the_face in self.white_graph.
                                        nodes():
506                                     if the_gc[2] == "a":
```

```
507                                         if not shift:
508                                             self.signed_white.append
                                                    ((w_edge, "+", char)
                                                    )
509                                         else:
510                                             self.signed_white.append
                                                    ((w_edge, "-", char)
                                                    )
511                                     elif the_gc[2] == "c":
512                                         if not shift:
513                                             self.signed_white.append
                                                    ((w_edge, "-", char))
514                                         else:
515                                             self.signed_white.append
                                                    ((w_edge, "+", char))
516                                 elif the_face in self.black_graph.
                                        nodes():
517                                     if the_gc[2] == "a":
518                                         if not shift:
519                                             self.signed_white.append
                                                    ((w_edge, "-", char)
                                                    )
520                                         else:
521                                             self.signed_white.append
                                                    ((w_edge, "+", char)
                                                    )
522                                     elif the_gc[2] == "c":
523                                         if not shift:
524                                             self.signed_white.append
                                                    ((w_edge, "+", char))
525                                         else:
526                                             self.signed_white.append
                                                    ((w_edge, "-", char))
527                 elif nodes[0] == nodes[1]:
528                     crossed_at = None
529                     for black_loop in black_loops:
530                         if black_loop[0] in nodes[0]:
531                             crossed_at = black_loop[0]
532                             black_loops.remove(black_loop)
533                             break
534                     if crossed_at == None:
535                         raise Exception("Loop does not exsist!!! 
                                Edge is not correct!!!")
536                     for gc in code_gauss_splitted:
537                         if gc[0] == crossed_at:
538                             if gc[2] == "a":
539                                 self.signed_white.append((w_edge, "+
                                        ", crossed_at))
540                             elif gc[2] == "c":
541                                 self.signed_white.append((w_edge, "-
                                        ", crossed_at))
542                             break
543 
544         else:
```

```
545             raise Exception("Gauss Code is Empty!!! We are flying
                    blind!!!")
546
547     def create_and_draw_sp(self):
548         graph = help.create_planar_graph_from_gauss_code(self.
                code_gauss)
549         self.create_nodes_from_faces(graph)
550         self.create_edges_after_nodes()
551         self.sign_sp()
552         self.draw_sp_graph()
553         self.draw_sp_graph_sign()
554
555     def create_and_sign_sp(self):
556         graph = help.create_planar_graph_from_gauss_code(self.
                code_gauss)
557         self.create_nodes_from_faces(graph)
558         self.create_edges_after_nodes()
559         self.sign_sp()
560
561
562     def label_edges_sp(Agraph, signs):
563         temp_signs = copy.deepcopy(signs)
564         for data in signs:
565             edge = data[0]
566
567     def find_loops_in_sp(self):
568
569         self.black_loops = []
570         for b in self.black_graph.nodes():
571             if len(b) == 2:
572                 if b[0] == b[1]:
573                     self.black_loops.append(b)
574
575         self.white_loops = []
576         for w in self.white_graph.nodes():
577             if len(w) == 2:
578                 if w[0] == w[1]:
579                     self.white_loops.append(w)
580
581     def find_cycles_of_length(graph, desired_length):
582         no_multi_graph = SignedPlanarGraph.multigraph_to_graph(graph
                )
583         cycles = nx.cycle_basis(no_multi_graph)
584         cycles_of_length = [cycle for cycle in cycles if len(cycle)
                == desired_length]
585         return cycles_of_length
586
587     def multigraph_to_graph(multi_graph):
588         graph = nx.Graph()
589         for u, v, key, data in multi_graph.edges(data=True, keys=
                True):
590             graph.add_edge(u, v)
591         return graph
```

C.7 Reidemeistermoves.py

```python
from SignedPlanarGraph import SignedPlanarGraph
import copy
import my_helpers as help

class ReidemeisterMoves:


    def reidemeister_main(sp_graph: SignedPlanarGraph) -> SignedPlanarGraph:

        reid_I, reid_II, reid_III = True, True, True
        joker_sp = copy.deepcopy(sp_graph)

        while reid_I or reid_II or reid_III:
            if reid_I or reid_II or reid_III:
                joker_sp_reid_I = ReidemeisterMoves.reidemeister_I(joker_sp)
                if joker_sp != joker_sp_reid_I:
                    joker_sp = joker_sp_reid_I
                    reid_I = False
                    reid_II, reid_III = True, True
                else:
                    reid_I = False

            if reid_I or reid_II or reid_III:
                joker_sp_reid_II = ReidemeisterMoves.reidemeister_II(joker_sp)
                if joker_sp != joker_sp_reid_II:
                    joker_sp = joker_sp_reid_II
                    reid_II = False
                    reid_I, reid_III = True, True
                else:
                    reid_II = False

            if reid_I or reid_II or reid_III:
                joker_sp_III = ReidemeisterMoves.reidemeister_III(joker_sp)
                if joker_sp != joker_sp_III:
                    joker_sp = joker_sp_III
                    reid_III = False
                    reid_I, reid_II = True, True
                else:
                    reid_III = False

        joker_sp = SignedPlanarGraph(code_gauss = joker_sp.code_gauss)
        joker_sp.create_and_sign_sp()
        return joker_sp

    def reidemeister_I(sp_graph: SignedPlanarGraph) -> SignedPlanarGraph:

        reid_I = copy.deepcopy(sp_graph)
        while True:
            reid_I_new = ReidemeisterMoves.reidemeister_I_helper(reid_I)
            if reid_I == reid_I_new:
                return reid_I_new
            reid_I = reid_I_new

    def reidemeister_II(sp_graph: SignedPlanarGraph) -> SignedPlanarGraph:
        reid_II = copy.deepcopy(sp_graph)
        while True:
            reid_II_new = ReidemeisterMoves.reidemeister_II_helper(reid_II)
            if reid_II_new == reid_II:
                return reid_II_new
            reid_II = reid_II_new

    def reidemeister_III(sp_graph: SignedPlanarGraph) -> SignedPlanarGraph:
        reid_III = copy.deepcopy(sp_graph)
        while True:
            reid_III_new = ReidemeisterMoves.reidemeister_III_helper(reid_III)
```

```
66              if reid_III_new == reid_III:
67                  return reid_III_new
68              reid_III = reid_III_new
69
70      def reidemeister_I_helper(sp_graph: SignedPlanarGraph) -> SignedPlanarGraph:
71          '''
72          we start by checking if the black graph has loops
73          '''
74          if sp_graph.white_loops != None:
75              [crossing, ch_r_i] = ReidemeisterMoves.reidemeister_helper_I_black(
                    sp_graph)
76              if ch_r_i:
77                  '''
78                  We have found an RI pair!!!
79                  Cleaning up
80                  '''
81                  code_gauss_final = copy.deepcopy(sp_graph.code_gauss)
82                  code_gauss_final = str(code_gauss_final)
83                  code_gauss_final_splitted = code_gauss_final.split(",")
84                  code_gauss_final = ""
85                  code_gauss_final = []
86                  for code in code_gauss_final_splitted:
87                      if crossing not in code:
88                          code_gauss_final += code + ","
89
90                  code_gauss_final = code_gauss_final[ : len(code_gauss_final) -
                        1]
91                  code_gauss_final = ''.join(code_gauss_final)
92
93                  final_sp = SignedPlanarGraph(code_gauss= code_gauss_final)
94                  final_sp.create_and_sign_sp()
95                  return final_sp
96
97          if sp_graph.black_loops != None:
98              [crossing, ch_r_i] = ReidemeisterMoves.reidemeister_I_helper_white(
                    sp_graph)
99              if ch_r_i:
100                 '''
101                 We have found an RI pair!!!
102                 Cleaning up
103                 '''
104
105                 code_gauss_final = copy.deepcopy(sp_graph.code_gauss)
106                 code_gauss_final = str(code_gauss_final)
107                 code_gauss_final_splitted = code_gauss_final.split(",")
108                 code_gauss_final = ""
109                 for code in code_gauss_final_splitted:
110                     if crossing not in code:
111                         code_gauss_final += code + ","
112
113                 code_gauss_final = code_gauss_final[ : len(code_gauss_final) -
                        1]
114                 code_gauss_final = ''.join(code_gauss_final)
115
116                 final_sp = SignedPlanarGraph(code_gauss= code_gauss_final)
117                 final_sp.create_and_sign_sp()
118                 return final_sp
119
120         return sp_graph
121
122     def reidemeister_helper_I_black(sp_graph):
123         if len(sp_graph.white_loops) != 0:
124             white_loops = copy.deepcopy(sp_graph.white_loops)
125             for white_loop in white_loops:
126                 crossing = white_loop[0]
127                 for edge in sp_graph.black_graph.edges():
128                     if edge[0] == edge[1]:
129                         if crossing in edge[0]:
```

```
130                             for signed_edge in sp_graph.signed_black:
131                                 if signed_edge[0] == edge:
132                                     sign = signed_edge[1]
133                                     for w_node in sp_graph.white_graph.nodes():
134                                         if len(w_node) == 2:
135                                             if w_node[0] == w_node[1]:
136                                                 if crossing in w_node:
137                                                     if len(sp_graph.white_graph[
                                                            w_node]) == 1:
138                                                         for signed_opp_edge in
                                                                sp_graph.signed_white:
139                                                             if w_node in
                                                                    signed_opp_edge[0]:
140                                                                 sign_opp =
                                                                        signed_opp_edge
                                                                        [1]
141                                                                 if sign != sign_opp:
142                                                                     return crossing,
                                                                            True
143         return None, False
144
145     def reidemeister_helper_I_white(sp_graph):
146         if len(sp_graph.black_loops) != 0:
147             black_loops = copy.deepcopy(sp_graph.black_loops)
148             for black_loop in black_loops:
149                 crossing = black_loop[0]
150                 for edge in sp_graph.white_graph.edges():
151                     if edge[0] == edge[1]:
152                         if crossing in edge[0]:
153                             for signed_edge in sp_graph.signed_white:
154                                 if signed_edge[0] == edge:
155                                     sign = signed_edge[1]
156                                     for b_node in sp_graph.black_graph.nodes():
157                                         if len(b_node) == 2:
158                                             if b_node[0] == b_node[1]:
159                                                 if crossing in b_node:
160                                                     if len(sp_graph.black_graph[
                                                            b_node]) == 1:
161                                                         for signed_opp_edge in
                                                                sp_graph.signed_black:
162                                                             if b_node in
                                                                    signed_opp_edge[0]:
163                                                                 sign_opp =
                                                                        signed_opp_edge
                                                                        [1]
164                                                                 if sign != sign_opp:
165                                                                     return crossing,
                                                                            True
166         return None, False
167
168     def reidemeister_II_helper(sp_graph: SignedPlanarGraph) -> SignedPlanarGraph:
169
170         black_reid = ReidemeisterMoves.reidemeister_II_helper_black(sp_graph)
171         if black_reid is not None:
172             return black_reid
173
174         white_reid = ReidemeisterMoves.reidemeister_II_helper_white(sp_graph)
175         if white_reid is not None:
176             return white_reid
177
178         return sp_graph
179
180
181     def reidemeister_II_helper_black(sp_graph: SignedPlanarGraph) ->
            SignedPlanarGraph:
182         '''
183         We start by checking parrallel edges in the black graph.
184         and keeping note of their different signs
```

```
185         '''
186         duplicates = {}
187         signs = {}
188         found_parallel = False
189         for edge in sp_graph.signed_black:
190             if edge[0] in duplicates:
191                 duplicates[edge[0]] += 1
192                 signs[edge[0]] += edge[1]
193                 found_parallel = True
194             else:
195                 duplicates[edge[0]] = 1
196                 signs[edge[0]] = edge[1]
197 
198         # Parrallel Edges exist
199         if found_parallel:
200             # We filter the edges from the dictionaries,
201             # to only have edges that have count more than 1.
202             filtered_duplicates = {edge: count for edge, count in duplicates.items()
                    if count >= 2}
203             filtered_signs = {edge: sign for edge, sign in signs.items() if edge in
                    filtered_duplicates}
204 
205             for edge in filtered_duplicates:
206                 sign = filtered_signs[edge]
207                 # Skip parrallel edges with only one type of sign.
208                 if len(set(sign)) == 1:
209                     continue
210 
211 
212                 parrallel_edges = []
213                 for signed_edge in sp_graph.signed_black:
214                     if signed_edge[0] == edge:
215                         parrallel_edges.append(signed_edge)
216                     i = 0
217                     while i < len(parrallel_edges):
218                         j = i + 1
219                         break_outer = False
220                         face = ""
221                         while j < len(parrallel_edges):
222                             pe_1 = parrallel_edges[i]
223                             pe_2 = parrallel_edges[j]
224                             if pe_1[1] != pe_2[1]:
225                                 face = pe_1[2] + pe_2[2]
226                                 if face in sp_graph.white_graph.nodes():
227                                     face_edges = []
228                                     for w_edge in sp_graph.signed_white:
229                                         if face in w_edge[0]:
230                                             face_edges.append(w_edge)
231                                     if len(face_edges) == 2:
232                                         if face_edges[0][1] != face_edges[1][1]:
233                                             break_outer = True
234                                             break
235                                 if face[::-1] in sp_graph.white_graph.nodes():
236                                     face = face[::-1]
237                                     face_edges = []
238                                     for w_edge in sp_graph.signed_white:
239                                         if face in w_edge[0]:
240                                             face_edges.append(w_edge)
241                                     if len(face_edges) == 2:
242                                         if face_edges[0][1] != face_edges[1][1]:
243                                             break_outer = True
244                                             break
245                             j += 1
246                         if break_outer:
247                             break
248                         i += 1
249 
250                 if face is None:
```

```
251                     raise Exception("Something is wrong!!")
252                 if len(face) == 0:
253                     raise Exception("Something is wrong!!")
254 
255                 '''
256                 Cleaning up
257                 '''
258 
259                 code_gauss_final = copy.deepcopy(sp_graph.code_gauss)
260                 code_gauss_final = str(code_gauss_final)
261                 code_gauss_final_splitted = code_gauss_final.split(",")
262 
263                 code_gauss_final = []
264                 for code in code_gauss_final_splitted:
265                     if face[0] not in code and face[1] not in code:
266                         code_gauss_final += code + ","
267 
268                 code_gauss_final = code_gauss_final[ : len(code_gauss_final) - 1]
269                 code_gauss_final = ''.join(code_gauss_final)
270 
271                 final_sp = SignedPlanarGraph(code_gauss= code_gauss_final)
272                 final_sp.create_and_sign_sp()
273                 return final_sp
274         return None
275 
276     def reidemeister_II_helper_white(sp_graph: SignedPlanarGraph) ->
        SignedPlanarGraph:
277         '''
278         We start by checking parrallel edges in the black graph.
279         and keeping note of their different signs
280         '''
281         duplicates = {}
282         signs = {}
283         found_parrallel = False
284         for edge in sp_graph.signed_white:
285             if edge[0] in duplicates:
286                 duplicates[edge[0]] += 1
287                 signs[edge[0]] += edge[1]
288                 found_parrallel = True
289             else:
290                 duplicates[edge[0]] = 1
291                 signs[edge[0]] = edge[1]
292 
293         # Parrallel Edges exist
294         if found_parrallel:
295             # We filter the edges from the dictionaries, to only have edges that
                have count more than 1.
296             filtered_duplicates = {edge: count for edge, count in duplicates.items()
                 if count >= 2}
297             filtered_signs = {edge: sign for edge, sign in signs.items() if edge in
                filtered_duplicates}
298 
299             for edge in filtered_duplicates:
300                 sign = filtered_signs[edge]
301                 # Skip parrallel edges with only one type of sign.
302                 if len(set(sign)) == 1:
303                     continue
304 
305 
306                 parrallel_edges = []
307                 for signed_edge in sp_graph.signed_white:
308                     if signed_edge[0] == edge:
309                         parrallel_edges.append(signed_edge)
310                 i = 0
311                 while i < len(parrallel_edges):
312                     j = i + 1
313                     break_outer = False
314                     face = ""
```

```
315                     while j < len(parrallel_edges):
316                         pe_1 = parrallel_edges[i]
317                         pe_2 = parrallel_edges[j]
318                         if pe_1[1] != pe_2[1]:
319                             face = pe_1[2] + pe_2[2]
320                             if face in sp_graph.black_graph.nodes():
321                                 face_edges = []
322                                 for b_edge in sp_graph.signed_black:
323                                     if face in b_edge[0]:
324                                         face_edges.append(b_edge)
325                                 if len(face_edges) == 2:
326                                     if face_edges[0][1] != face_edges[1][1]:
327                                         break_outer = True
328                                         break
329                             if face[::-1] in sp_graph.black_graph.nodes():
330                                 face = face[::-1]
331                                 face_edges = []
332                                 for b_edge in sp_graph.signed_black:
333                                     if face in b_edge[0]:
334                                         face_edges.append(b_edge)
335                                 if len(face_edges) == 2:
336                                     if face_edges[0][1] != face_edges[1][1]:
337                                         break_outer = True
338                                         break
339                         j += 1
340                     if break_outer:
341                         break
342                     i += 1
343 
344                 if face is None:
345                     raise Exception("Something is wrong!!")
346                 if len(face) == 0:
347                     raise Exception("Something is wrong!!")
348 
349                 '''
350                 Cleaning up
351                 '''
352 
353                 code_gauss_final = copy.deepcopy(sp_graph.code_gauss)
354                 code_gauss_final = str(code_gauss_final)
355                 code_gauss_final_splitted = code_gauss_final.split(",")
356 
357                 code_gauss_final = []
358                 for code in code_gauss_final_splitted:
359                     if face[0] not in code and face[1] not in code:
360                         code_gauss_final += code + ","
361 
362                 code_gauss_final = code_gauss_final[ : len(code_gauss_final) - 1]
363                 code_gauss_final = ''.join(code_gauss_final)
364 
365                 final_sp = SignedPlanarGraph(code_gauss= code_gauss_final)
366                 final_sp.create_and_sign_sp()
367                 return final_sp
368         return None
369 
370     def reidemeister_III_helper(sp_graph:SignedPlanarGraph) -> SignedPlanarGraph:
371         to_remove_gc_wh = ReidemeisterMoves.reidemeister_III_helper_white(sp_graph)
372         if len(to_remove_gc_wh) != 0:
373                 code_gauss_final = copy.deepcopy(sp_graph.code_gauss)
374                 code_gauss_final = str(code_gauss_final)
375                 code_gauss_final_splitted = code_gauss_final.split(",")
376 
377                 code_gauss_final = []
378                 for code in code_gauss_final_splitted:
379                     if code[0] not in to_remove_gc_wh:
380                         code_gauss_final += code + ","
381 
382                 code_gauss_final = code_gauss_final[ : len(code_gauss_final) - 1]
```

```
383                 code_gauss_final = ''.join(code_gauss_final)
384
385                 final_sp = SignedPlanarGraph(code_gauss= code_gauss_final)
386                 final_sp.create_and_sign_sp()
387                 return final_sp
388
389         to_remove_gc_bl = ReidemeisterMoves.reidemeister_III_helper_black(sp_graph)
390         if len(to_remove_gc_bl) != 0:
391                 code_gauss_final = copy.deepcopy(sp_graph.code_gauss)
392                 code_gauss_final = str(code_gauss_final)
393                 code_gauss_final_splitted = code_gauss_final.split(",")
394
395                 code_gauss_final = []
396                 for code in code_gauss_final_splitted:
397                     if code[0] not in to_remove_gc_bl:
398                         code_gauss_final += code + ","
399
400                 code_gauss_final = code_gauss_final[ : len(code_gauss_final) - 1]
401                 code_gauss_final = ''.join(code_gauss_final)
402
403                 final_sp = SignedPlanarGraph(code_gauss= code_gauss_final)
404                 final_sp.create_and_sign_sp()
405                 return final_sp
406
407         return sp_graph
408
409     def reidemeister_III_helper_black(sp_graph:SignedPlanarGraph) ->
        SignedPlanarGraph:
410         black_cycles = ReidemeisterMoves.reidemeister_III_find_cycles(sp_graph, "
            black")
411         white_stars = []
412
413         for bl_cycle in black_cycles:
414             found = False
415             opp_face = bl_cycle[1][0]
416             for white_face in sp_graph.white_graph.nodes():
417                 if help.is_subset_both_ways(white_face, opp_face):
418                     opp_face = white_face
419                     found = True
420                     break
421             if not found:
422                 continue
423             faces_to_consider = []
424             for face in bl_cycle[0]:
425                 if len(face) < 4:
426                     faces_to_consider.append(face)
427
428             for char in opp_face:
429                 if char in faces_to_consider[0] and char in faces_to_consider[1]:
430                     remains = char
431                     break
432             faces_to_consider.append(opp_face)
433             faces_to_consider_str = "".join(faces_to_consider)
434             faces_to_consider_str = faces_to_consider_str.replace(char, "")
435             if len(set(faces_to_consider_str)) != 4:
436                 continue
437             return faces_to_consider_str
438         return ""
439
440     def reidemeister_III_helper_white(sp_graph:SignedPlanarGraph) ->
        SignedPlanarGraph:
441         white_cycles = ReidemeisterMoves.reidemeister_III_find_cycles(sp_graph, "
            white")
442         black_stars = []
443
444         for wh_cycle in white_cycles:
445             found = False
446             opp_face = wh_cycle[1][0]
```

```
447             for black_face in sp_graph.black_graph.nodes():
448                 if help.is_subset_both_ways(black_face, opp_face):
449                     opp_face = black_face
450                     found = True
451                     break
452             if not found:
453                 continue
454             faces_to_consider = []
455             for face in wh_cycle[0]:
456                 if len(face) < 4:
457                     faces_to_consider.append(face)
458 
459             for char in opp_face:
460                 if char in faces_to_consider[0] and char in faces_to_consider[1]:
461                     remains = char
462                     break
463             faces_to_consider.append(opp_face)
464             faces_to_consider_str = "".join(faces_to_consider)
465             faces_to_consider_str = faces_to_consider_str.replace(char, "")
466             if len(set(faces_to_consider_str)) != 4:
467                 continue
468             return faces_to_consider_str
469         return ""
470 
471     def reidemeister_III_find_cycles(sp_graph:SignedPlanarGraph, colour) -> list:
472         if colour == "black":
473             return ReidemeisterMoves.reidemeister_III_find_cycles_black(sp_graph)
474         elif colour == "white":
475             return ReidemeisterMoves.reidemeister_III_find_cycles_white(sp_graph)
476 
477     def reidemeister_III_find_cycles_black(sp_graph:SignedPlanarGraph):
478         # We start by looking for cycles in the black graph
479         black_cycles = SignedPlanarGraph.find_cycles_of_length(sp_graph.black_graph,
                3)
480         filtered_bl_cycles = []
481         #we check the cycles to pick only with edges of different signs.
482         for cycle in black_cycles:
483             i = 0
484             edges_sign_crossing = []
485             while i < len(cycle):
486                 j = (i+1) % len(cycle)
487                 edges_sign_crossing += [(sgn_edge[1], sgn_edge[2]) for sgn_edge in
                        sp_graph.signed_black
488                             if sgn_edge[0] == (cycle[i], cycle[j]) or sgn_edge[0] == (
                                cycle[j],cycle[i])]
489                 i += 1
490             crossings = []
491             signs = []
492             for sgn_edge in edges_sign_crossing:
493                 signs.append(sgn_edge[0])
494                 crossings.append(sgn_edge[1])
495 
496             cycle_data = ("".join(crossings), "".join(signs))
497 
498             if len(cycle_data[1]) == 3 and len(set(cycle_data[1])) > 1:
499                 filtered_bl_cycles.append((cycle, cycle_data))
500 
501         return filtered_bl_cycles
502 
503     def reidemeister_III_find_cycles_white(sp_graph:SignedPlanarGraph):
504         # We start by looking for cycles in the black graph
505         white_cycles = SignedPlanarGraph.find_cycles_of_length(sp_graph.white_graph,
                3)
506         filtered_wh_cycles = []
507         #we check the cycles to pick only with edges of different signs.
508         for cycle in white_cycles:
509             i = 0
510             edges_sign_crossing = []
```

```
511             while i < len(cycle):
512                 j = (i+1) % len(cycle)
513                 edges_sign_crossing += [(sgn_edge[1], sgn_edge[2]) for sgn_edge in
                        sp_graph.signed_white
514                         if sgn_edge[0] == (cycle[i], cycle[j]) or sgn_edge[0] == (
                            cycle[j],cycle[i])]
515                 i += 1
516
517             crossings = []
518             signs = []
519             for sgn_edge in edges_sign_crossing:
520                 signs.append(sgn_edge[0])
521                 crossings.append(sgn_edge[1])
522
523             cycle_data = ("".join(crossings), "".join(signs))
524
525             if len(cycle_data[1]) == 3 and len(set(cycle_data[1])) > 1:
526                 filtered_wh_cycles.append((cycle, cycle_data))
527
528         return filtered_wh_cycles
```