KnotWeaver:
An interactive knot editor

by

Peter Vos

Supervisors:

dr. ir. Huub van de Wetering
prof. dr. Arjeh Cohen
drs A.B. van der Roest

Eindhoven, June 2007
Abstract

Knot theory is a mathematical subject and as many mathematical subjects it can be studied at many levels of details. But regardless of how familiar a person is with the subject, an interactive visualization can provide valuable insights in order to understand more about this field of mathematics. The goal of this master project is to create a computer application able to construct, edit and visualize a wide variety of knots and to deliver information about them to the user. In this respect it can serve as an educational tool for math students that begin learning the theory but also allows for academic researchers to interactively delve into the involving math. This report documents the process of the project, the design decisions that were taken and the resulting application.
# Contents

1 Concepts and Definitions 5
   1.1 Concepts ................................................................. 5
   1.2 Notations ............................................................... 9

2 Requirements 10
   2.1 Functional Requirements ............................................ 10
   2.2 Non-Functional Requirements ...................................... 11

3 Algorithms 12
   3.1 Reidemeister Moves .................................................. 12
      3.1.1 Introduction ..................................................... 12
      3.1.2 Algorithm Overview ............................................ 13
      3.1.3 Detailed Algorithmic Steps ................................... 13
      3.1.4 Notations ....................................................... 16
      3.1.5 Definitions .................................................... 16
      3.1.6 Detailed Algorithms .......................................... 18
   3.2 Seifert Circles ..................................................... 22
      3.2.1 Introduction ..................................................... 22
      3.2.2 Representations ................................................ 23
      3.2.3 Algorithms ..................................................... 24
   3.3 Knot Layout Creation from Gauss Codes ......................... 24
      3.3.1 Introduction ..................................................... 24
      3.3.2 Algorithm Overview ............................................ 25
      3.3.3 Detailed Algorithms .......................................... 26
Introduction

Knot theory is a subfield of the mathematical subject of topology. It considers closed curves in 3D space, called knots. One could compare such a curve with an electrical extension cord wound and knotted with itself, the two connectors plugged into each other. The result is a closed curve with a multitude of mathematically interesting properties. For example the amount of times the cord crosses itself when looking from a particular angle at it, or the minimum number of crossings it has when spread out on a surface.

By nature knot theory is a visual mathematical domain: as such it is no surprise that there are already computer programs available that visualize knots. For example Knotilus [FRF03] is an automated knot drawing program and KnotPlot[Sch98] is an elaborate knot theory application. Other applications range from 2D to 3D and from static to interactive, mostly designed to tackle a specific problem or to display a particular property. There is a lack however of programs that allow editing under the constraint of knot equivalence.

The lack of an editing tool that allows editing while keeping the knot invariant prompted members of the mathematics and computer science department at the Technische Universiteit Eindhoven to formulate a computer science graduation project. They experienced that drawing and redrawing knots on paper when manipulating them was time consuming and prone to errors as well as being demotivating for students learning the theory. What they aimed for was a program that could emulate knot physics such that the drawing and manipulation of knot diagrams became easier and more time efficient for the user.

Abstract as it may seem, knot theory has many practical applications. Chemistry and biology for example are confronted with extensive molecules that show different properties when knotted in different ways; a well known example of such a molecule is the DNA molecule. Other applications range from statistical mechanics to particle physics.
Chapter 1

Concepts and Definitions

The following chapter explains the concepts used in this document, detailed information about these concepts can be found in Adams’ ’The Knot Book’ [Ada94] which gives an excellent introduction to knot theory.

1.1 Concepts

The main subject in knot theory is the knot itself, it is defined as a non-self-intersecting closed curve embedded in 3D space, much like a piece of wire with the ends fused together. The piece of wire can be twisted, turned and strung through loops it makes any number of times before the ends are fused together. Once closed the knot is finished and uniquely defined.

Knots can also tangle into other knots, the resulting configurations, made out of multiple closed curves are called links and the different curves that make up a link are called components. Because links and knots share a great deal of their properties they are often used as synonyms, when the distinction would make a difference, it will be explicitly mentioned.

If part of a knot is smoothly moved around without intersecting the knot itself, we consider it mathematically unchanged. Two knots are considered equivalent if one can be moved continuously to coincide with the other. For example if one would take a piece of wire, tie a knot with it and fuse the ends together, every knot resulting from stretching, bending and knotting that wire without breaking it would be equivalent to the starting knot.

Knots are 3D objects, so visualizing them on a piece of paper or a computer screen requires the elimination of one dimension. This is done in a knot diagram where the 3D object is projected onto a 2D surface. In such a projection lines and points must not coincide and to ensure the knot diagram represents the knot correctly each crossing on the diagram has to have indicated which line section is above and which one is below. In a knot diagram the lines between crossings are called strands.
A special form of knot is a **stick knot**. A stick knot is a knot that is constructed only from straight lines. This opens up possibilities for new properties, like what is the minimum amount of sticks needed to draw a particular knot. Note that knot diagrams consisting of only straight lines are not always stick knots. In the projection from 3D to 2D that transforms a knot into a knot diagram curves can become straight lines. A 2D knot diagram can fool the viewer by presenting curved line segments as a straight line, effectively hiding the curve to the viewer.

![Figure 1.1: Two knot diagrams of the two trefoil knots](image)

In order to still modify a knot when it is projected into a knot diagram so-called Reidemeister moves can be used (After the German mathematician Kurt W.F. Reidemeister 1893-1971). A **Reidemeister move** is a modification of a knot diagram which preserves equivalence of corresponding knots. There are 3 atomic Reidemeister moves, any combination of these moves will preserve knot equivalence and likewise every move that preserves knot equivalence can be split up into a sequence of Reidemeister moves[Rei32]. To say Reidemeister move \( r \) transforms knot \( \alpha \) into knot \( \beta \), \( \alpha \xrightarrow{r} \beta \) is used.

A knot diagram is not the only way of representing a knot in 2D. Another way of representing knots is by use of a braid. A **braid** is a special kind of knot drawing consisting of a number of vertical strands that proceed monotonically downward (See Figure 1.3). By crossing over neighboring strands, different braids can be obtained. The knot represented by a braid is achieved by looping each begin point with the end point directly above without making new crossings with the loops [Art47].

![Figure 1.2: A stick diagram of a trefoil knot](image)
A simple way of describing a knot diagram is by use of a Gauss code. A technique named after the famous German mathematician Carl Friederich Gauss (1777-1855). A Gauss code is a set of oriented lists of signed crossings. It is obtained by the following basic algorithm [Ada94]:

1. Number the crossings of the link from 1 to n
2. Pick a component and start tracing it from an arbitrary begin point until you return to the begin point. When a crossing is encountered write down the number of that crossing with positive or negative sign when going over or under the crossed strand respectively.
3. Repeat step 2 for all other components of the link (if any).

This algorithm generates a simple Gauss code. From now on Gauss codes will be represented by Greek letters $\alpha, \beta, ...$, their components by capital letters $A, B, ...$ and their crossings by small letters $a, b, ...$
The following simple Gauss code is one of the possible Gauss codes for the trefoil knot displayed in Figure 1.1 a):

\[ \alpha = \left[ [-1, +2, -3, +1, -2, +3] \right] \]

Several Gauss codes are possible for a single knot, depending on the choices made in step 1. and 2. of the algorithm. Another undesirable property of simple Gauss codes is that they do not unambiguously define a single link. In order to create a single unique knot from a Gauss code more information than given by a simple Gauss code is needed. When trying to construct a knot diagram from a simple Gauss code an ambiguity will arise that forces a choice between a knot and its mirror image. In Figure 1.1 two trefoil knots are drawn which are each others mirror image, these may look very much alike but they are essentially different knots. It is impossible to transform a) to b) with Reidemeister moves. This ambiguity arises due to the uncertainty of crossing orientation. There are two different crossing orientations — shown in Figure 1.4 —. A ‘$+$’ is used for crossings such as in Figure 1.4 a) and a ‘$-$’ is used for crossings such as in Figure 1.4 b). This information is added to the Gauss code after all the components are listed and is sequentially matched to crossings in order of appearance in the components.

The result is a complete Gauss code, for the knot diagrams in Figure 1.1 the following Gauss codes are obtained when we start counting from the arrow marker:
\[ a) = \left[ \text{-}1, +2, -3, +1, -2, +3 \right] / \text{---} \]
\[ b) = \left[ +1, -2, +3, -1, +2, -3 \right] / \text{+++} \]

**Figure 1.4:** a) crossing type '+' b) crossing type '-'

There is still not a single unique complete Gauss code for a given knot but at least each complete Gauss code uniquely determines a knot.

One of the visual properties of a knot diagram are **Seifert circles** [Sci35], they are used to transform knot diagrams into braid diagrams and can be constructed by eliminating each crossing by connecting each of the strands coming into the crossing to the crossed strands leaving the crossing. The result is a set of non-intersecting circles called Seifert circles (After the German mathematician Herbert K.J. Seifert 1907-1996). See Figure 1.5.

**Figure 1.5:** a) Transformation of the figure eight knot to its Seifert circles, b) Transformation of a knot crossing during the Seifert circle algorithm, From: [vWC06]
1.2 Notations

To facilitate the explanations in this document, some notation is given here. Having explained earlier the different notations for Gauss codes, components and crossings, now the derived notations can be explained by an example. To indicate the set of signed crossings in Gauss code $\alpha$ and component $A$ the symbols $X_{\alpha}^\pm$ and $X_A^\pm$ are used respectively, for unsigned crossings the symbols $X_\alpha$ and $X_A$ are used. For the set of components in Gauss code $\alpha$, $C_\alpha$ is used. To indicate the size of a set, the prefix $\# \, \#$ is used. For example:

\[
\begin{align*}
\alpha &= [+1, -2, +3, -1, +2, -3] [-4, +4]/ - - - \\
A &= [+1, -2, +3, -1, +2, -3], \quad B = [-4, +4] \\
C_\alpha &= \{A, B\} \\
X_\alpha^\pm &= \{-1, -2, -3, -4, +1, +2, +3, +4\} \\
X_\alpha &= \{1, 2, 3, 4\} \\
X_A^\pm &= \{-1, -2, -3, +1, +2, +3\}, \quad X_B^\pm = \{-4, +4\} \\
X_A &= \{1, 2, 3\}, \quad X_B = \{4\} \\
\#X_A^\pm &= 6, \quad \#C_\alpha = 2
\end{align*}
\]

Crossings are signed in Gauss codes, so to denote their sign and absolute value, $\text{sign}(a)$ and $|a|$ are used. With the absolute value it can be determined if two crossings in a Gauss code correspond to the same crossing in the knot diagram.

\[
\begin{align*}
\text{sign}(+3) &= +1, \quad \text{sign}(-3) = -1 \\
| -3 | &= 3 = | +3 |
\end{align*}
\]

Because components are inherently circular it does not matter where the counting of crossings is started when constructing a Gauss code, but in order to write them down some begin point has to be chosen in the components of $\alpha$. In the previous example the first crossing in component $A$ is $+1$. The notation $A_i$ allows indexing for components.

\[
\begin{align*}
A &= [+1, -2, +3, -1, +2, -3] \\
A_0 &= +1, \quad A_2 = +3, \quad A_5 = -3
\end{align*}
\]

The following chapters will describe the process of creating a knot program. First, in Chapter 2, the requirements for the program are given, describing what functionality the program should have. Then, in Chapter 3, a number of algorithms is given which solve the most important problems that need to be overcome in order to implement the requirements. In chapter 4 the design of the application is explained and chapter 5 goes into detail about the implementation.
Chapter 2

Requirements

This graduation project is inspired by the mathematics department of the Technische Universiteit Eindhoven. The idea is to create a program that allows a realistic interaction with knots. A program that forms a solid base with actual functionality but that is still extendable. This chapter describes in detail the user requirements of this program, called KnotWeaver. In essence it defines ‘what’ the application is supposed to do, not ‘how’ it is supposed to do it. The purpose of this chapter is to define a contract between developer and customer which determines which features the software must contain, and what their priorities are.

2.1 Functional Requirements

When it comes to input the most important way to enter a knot into the program is the manual creation method. User needs to be able to manually create a knot with ease in such a fashion that is both intuitive and powerful. Mouse actions are highly preferred over keyboard actions because of their intuitive nature.

Creating knots by entering their complete Gauss code should be possible too, important here is that a good looking resulting knot is the product of this entry method.

In the eye of interoperability with other knot programs it should be possible to read different formats from different existing knot programs such as KnotPlot and accept braid notation as input to ensure a parallel with SeifertView[vWC06].

The output requirements are similar to the input requirements. Interoperability with other knot programs comes in to play when it comes to output formats. KnotWeaver should be able to export KnotPlot files, braid notation and Gauss codes. A visual export method should be available too.

In the spectrum of editing requirements two modes appear, one where there is no limit on the modification and alteration of a knot and its diagram, and another where the modifications in the knot diagram are limited to Reidemeister moves as not to alter the knot itself. In the first mode full control over lines and crossings in the knot diagram is possible, in the latter only changes to the knot diagram that can be classified as a Reidemeister move are allowed.

An essential part of the program is the visualization of knots and their properties. In this respect a clear visualization that is intuitive for the user is needed for the property of Seifert circles and the Yamada-Vogel algorithm. As an additional feature smoothing a knot such that it has a pleasant shape should be added.
From the worded requirements above a table is constructed listing all requirements in atomic parts, these parts can then later be held next to the implementation to check whether they are implemented or not. Each requirement gets a priority assigned indicating its importance for the finally delivered application.

<table>
<thead>
<tr>
<th>Input Requirements</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>IR01 Manual input with mouse interaction</td>
<td>1</td>
</tr>
<tr>
<td>IR02 Gauss code</td>
<td>2</td>
</tr>
<tr>
<td>IR03 KnotPlot format</td>
<td>3</td>
</tr>
<tr>
<td>IR04 Braid notation</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output Requirements</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>OR01 Gauss Code</td>
<td>1</td>
</tr>
<tr>
<td>OR02 Visual Output</td>
<td>2</td>
</tr>
<tr>
<td>OR03 KnotPlot format</td>
<td>3</td>
</tr>
<tr>
<td>OR04 Braid notation</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Editing Requirements</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ER01 Manual editing that does not change the knot</td>
<td>1</td>
</tr>
<tr>
<td>ER01a Dragging of the knot limited to Reidemeister moves</td>
<td>1</td>
</tr>
<tr>
<td>ER01b Direct information about Reidemeister moves during editing</td>
<td>1</td>
</tr>
<tr>
<td>ER02 Manual editing that changes the knot</td>
<td>1</td>
</tr>
<tr>
<td>ER02a Full edit control over dragging</td>
<td>1</td>
</tr>
<tr>
<td>ER02b Swapping crossings</td>
<td>1</td>
</tr>
<tr>
<td>ER02c Change orientation of a component</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Visualisation Requirements</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VR01 Seifert Circles</td>
<td>1</td>
</tr>
<tr>
<td>VR02 Yamada-Vogel Algorithm</td>
<td>2</td>
</tr>
<tr>
<td>VR03 Smoothing options for the link</td>
<td>3</td>
</tr>
</tbody>
</table>

**Requirement legend**

1. High
2. Medium
3. Low

### 2.2 Non-Functional Requirements

**Interface:** An easy to use interface must be created for the application, this means it should be as easy as possible for new users to start taking full advantage of the capabilities of the program. If possible, conventions around common functionalities should be adhered to and when that is not possible it should be clear how the functionality is accessed.

**Platform:** The application has to be constructed in Java in order for it to be platform independent and for possible transport to a browser based web applet.

**Expandability:** Future expansion of the application should be possible and as easy as possible. During each construction phase of the software expandability should be taken into consideration.

**Documentation:** In light of expandability and correctness all interfaces should be well documented through the JavaDoc standard. Furthermore documentation about the internal functionality is highly appreciated.

**Testing:** To ensure implementation correctness unit tests should be constructed for all important Java classes that make up the program.
In this chapter the most interesting algorithms used by KnotWeaver are explained. First a section about Reidemeister moves is presented, in that section more details and information will be given about how KnotWeaver recognizes Reidemeister moves and how they are used to satisfy requirement ER01 from the requirements (Manual editing that does not change the knot). Second a small section will deal with Seifert circles, this section is used to explain more about how requirement VR01 is realized but also to explain some functionality that is used by the algorithm in the last section. In the final section an algorithm is presented to create and layout a knot from a Gauss code (Requirement IR02). This algorithm is not implemented in KnotWeaver as there are a few research and implementation questions left open for future work.

3.1 Reidemeister Moves

3.1.1 Introduction

Reidemeister moves are local changes in a knot diagram that do not alter the knot itself. Only three Reidemeister moves exist and any transformation between two equivalent knots with different knot diagrams can be achieved by applying a series of Reidemeister moves between those two knot diagrams. Reidemeister moves are used to prove equivalence of knots.

\[ \text{Figure 3.1:} \ \ a) \ R1 \ : \ \text{a strand is twisted so that a loop is created or removed, b) } R2 : \text{ two strands are moved so that they overlap or that an overlap is removed } \text{c) } R3 : \text{ a strand that is under (or over) two crossing strands is pulled under (or over) that crossing} \]
Reidemeister moves are referred to as R1, R2 and R3. R1 and R2 change the amount of crossings, while R3 keeps the crossing number invariant. Due to this asymmetry R1 and R2 moves that gain crossings are respectively called ‘R1 add’ and ‘R2 add’, and the R1 and R2 moves that lose crossings are respectively called ‘R1 remove’ and ‘R2 remove’. Since there is no sense of direction in R3, it is called ‘R3 shift’ either way. (Figure 3.1)

In order to create an editing mode in the software that only allows Reidemeister moves (Requirement ER01) it is necessary to be able to say whether a move was a Reidemeister move. When a Reidemeister move is performed some more information about it is needed, such as what kind of Reidemeister move was performed and which crossings were involved in the move (Requirement ER01b). Algorithms that handle the above problems are introduced in section 3.1.

3.1.2 Algorithm Overview

The goal of the following algorithms is to make it possible to compare two knots and determine if they are exactly one Reidemeister move apart. Applying a particular Reidemeister move is not considered but rather the knots before and after applying the Reidemeister move are considered. By comparing two corresponding Gauss codes it is possible to extract whether a particular Reidemeister move has occurred. In order to state if two Gauss codes $\alpha$ and $\beta$ are one Reidemeister move apart, three steps are distinguished:

**Step one:** Detecting possible Reidemeister moves between $\alpha$ and $\beta$.

**Step two:** Executing a Reidemeister move in $\alpha$ or $\beta$.

**Step three:** Determining equivalence of $\alpha$ and $\beta$. (as further described in Definition 3.5 in Section 3.1.5)

Steps 2 and 3 are repeated for each possible Reidemeister move of step 1.

3.1.3 Detailed Algorithmic Steps

**Step one:** Detecting a possible Reidemeister move in a Gauss codes

In the first step two Gauss codes are compared and checked if they are one Reidemeister move apart. Because each Reidemeister move affects a different amount of crossings it can very quickly be seen which (if any) Reidemeister move type has occurred.

- $R1$ remove ($R1^-$) reduces the number of crossings by 2.
- $R1$ add ($R1^+$) increases the number of crossings by 2.
- $R2$ remove ($R2^-$) reduces the number of crossings by 4.
- $R2$ add ($R2^+$) increases the number of crossings by 4.
- $R3$ shift does not change the number of crossings.

Further more because ‘add’ and ‘remove’ are each other’s inverses, for R1 the same algorithm can be applied to detect $\alpha \xrightarrow{R1^+} \beta$ and $\beta \xrightarrow{R1^-} \alpha$. Similarly for R2, $\alpha \xrightarrow{R2^+} \beta$ and $\beta \xrightarrow{R2^-} \alpha$ can use the same detection algorithm. The choice is now to make algorithms that detect ‘adds’ or algorithm that
detect 'removes'. 'Adds' can be inserted at any point in the Gauss code, but 'removes' can only be done at places in the Gauss code that match a specific pattern. That makes finding a possible 'remove' harder, but eventually more rewarding because fewer possibilities make it through this first step.

By looking at the gauss code and detecting certain patterns in it, the places where a Reidemeister move can be executed become visible. Listed below are the patterns for each Reidemeister move.

a) A possible 'R1 remove' is characterized by the patterns:
   
   
   
   \[ 
   \ldots + x, \ldots - x.. \] and \[ 
   \ldots - x, \ldots + x.. \] 
   
   or generalized as a crossing immediately followed by the same crossing with the opposite sign.

b) For an 'R2 remove' the patterns searched for are:
   
   \[ 
   \ldots + x, \ldots + y.. \] and \[ 
   \ldots - x, \ldots - y.. \] and \[ 
   \ldots + x, \ldots + y.. \] and \[ 
   \ldots - y, \ldots - x.. \] 
   
   or generalized as a positive pair of consecutive crossings and a negative pair of the same consecutive crossings.

c) A possible 'R3 shift' can be found by searching for the pattern:
   
   \[ 
   \ldots + x, \ldots + y.. \] and \[ 
   \ldots - y, \ldots + z.. \] and \[ 
   \ldots - z, \ldots - x.. \] 
   
   or generalized as a positive pair of consecutive crossings, a negative pair of consecutive crossings and a mixed pair of consecutive crossings, where in total only three different crossings are concerned.

Step two: Executing a Reidemeister move in a Gauss code

In the second step one of the in the first step found Reidemeister moves is executed. For the execution of a 'R1 remove' with crossing \( x \) in a Gauss code it suffices to remove both positive and negative reference to crossing \( x \) from that Gauss code. In the same way an 'R2 remove' can be executed by removing crossings \( x \) and \( y \) from the Gauss code. Executing R3 with crossings \( x, y \) and \( z \) is slightly more complicated, pairs: \[ 
   \ldots + x, \ldots + y.. \] and \[ 
   \ldots - y, \ldots + z.. \] and \[ 
   \ldots - z, \ldots - x.. \] are converted to \[ 
   \ldots + y, \ldots + x.. \] and \[ 
   \ldots + z, \ldots - y.. \] and \[ 
   \ldots - x, \ldots - z.. \] as seen in Figure 3.3 meaning each pair gets inverted. In this step all additional information about the Reidemeister move is available: What type of Reidemeister move it is, and which crossings are involved.
Step three: Determining equivalence of Gauss codes

In the third step the Gauss code with a Reidemeister move executed on is compared with the unchanged Gauss code to see if they are equivalent. Two Gauss codes may look quite different, but still represent the same link. The differences can be found in three places: The ordering of the components in the Gauss code, the numbering of crossings in the Gauss code and the beginning point from which the crossings are written down per component.

\[ \alpha : [1, +2, -3, +1, -4, +5, -2, +3][+6, -5, +4, -7][+7, -6] \]

\[ \beta : [1, +3, -1, +5][+6, -3, +6][+2, -5, +1, -4, +7, -2, +4, -7] \]

**Figure 3.4:** Two different Gauss codes representing the same link

In Figure 3.4 two equivalent Gauss codes are shown: \( \alpha \) consisting of component \( A, B \) and \( C \), and \( \beta \) consisting of component \( D, E \) and \( F \). To show that \( \alpha \) is equivalent to \( \beta \) we define \( m, n \) and \( \text{offset} \) as follows. Component map \( m : C_\alpha \rightarrow C_\beta \) maps the components of \( \alpha \) to the components of \( \beta \), telling which component of \( \alpha \) and which component from \( \beta \) represent the same component in the link.

\[ m : C_\alpha \rightarrow C_\beta = \{(A, F), (B, D), (C, E)\} \]

Crossing map \( n : X_\alpha \rightarrow X_\beta \) tells which numbers in \( \alpha \) and \( \beta \) represent the same crossings.

\[ n : X_\alpha \rightarrow X_\beta = \{(1, 2), (2, 4), (3, 7), (4, 5), (5, 1), (6, 3), (7, 6)\} \]

Finally there is an offset for each component that tells the difference in position between the beginning crossings of corresponding components.

\[ \text{offset}_A = 1, \ \text{offset}_B = 1, \ \text{offset}_C = 5 \]

Determining these three elements is what is needed to determine whether two Gauss codes represent the same link.

In section 3.1.4 and 3.1.5, notations and definitions will be given to understand the detailed algorithms in pseudo code, in section 3.1.6.
3.1.4 Notations

For the algorithms given in section 3.1.6 some notations for the relevant concepts are introduced. They are used in preconditions, postconditions, annotations and the pseudo-code.

To execute R1 and R2 moves (as mentioned in the Algorithm Overview) a notation is needed that indicates that all positive and negative references to a particular crossing $x$ are removed from a Gauss code $\alpha$:

$$\alpha \setminus \{x\}$$

Crossings are ordered elements of a component, which means they have successors. $a \oplus n$ is the notation for the $n$th successor of $a$ modulo the amount of crossings in the component of $a$:

$$\text{in component } [c, d, e, f, a, b], a \oplus 3 = d$$

Because components are circular concepts it does not matter at which crossing they start; therefore components can be rotated by a particular offset in a way very similar to determining the $n$th successor of a crossing. $A \oplus n$ is the component resulting from taking each crossing $A_i$ and replacing it with $A_i \oplus n$, this is called shifting of a component.

$$[c, d, e, f, a, b] \oplus 3 = [f, a, b, c, d, e]$$

In Figure 3.4 and the subsequent elaboration on the figure the concepts of mappings and how they are used is explained. To add item $'x$ to $y'$ to a map $m : \Phi \rightarrow \Psi$ the pseudo-code $m.add(x, y)$ is used. The requirement is that $x$ and $y$ are of the domain ($\Phi$) and range ($\Psi$) type respectively of the map. The result of the pseudo-code $m.add(x, y)$ is that $x$ maps to $y$ in $m$:

$$m(x) = y$$

In the pseudo code all method parameters are called by reference. This means the argument is passed by implicit reference rather than the argument value. Any changes the called function makes to the argument will be visible to the calling function as well.

3.1.5 Definitions

In this section some new definitions will be presented that will later make it easier to reason about the given algorithms. The given definitions range over all the previously presented concepts: crossings, components and Gauss codes. They combine the notations of section 3.1.4 and add an abstraction level.

The crossing is at the bottom level of the Gauss code structure. The relevant concept concerning crossings when comparing two Gauss codes for equivalence is crossing uniformity. Meaning that one crossing is mapped to the other by the crossing map that connects both Gauss codes, and the crossings have the same sign.

**Definition 3.1 (Uniform crossings)** For two crossings $a$, $b$, and bijective crossing map $n$, $a$ is uniform with $b$ by $n$ iff the absolute value of $n(a)$ equals the absolute value of $b$ and $n(a)$ and $b$ have the same sign. Formally:

$$a \sim_n b \iff |n(a)| = |b| \land \text{sign}(n(a)) = \text{sign}(b)$$

At the middle level of the Gauss code structure is the component. The relevant concept defined here is component congruence. This says two components are congruent when they have the same amount of crossings, and crossings at the same index of the components are uniform by the crossing map that connects the two Gauss codes of the two components.
Definition 3.2 (Congruent components) For two components $A$, $B$ and bijective crossing map $n$, $A$ is congruent with $B$ by $n$ iff $A$ and $B$ have the same amount of crossings and for each crossing $A_i$ of $A$ the crossing $B_i$ is uniform to $B_i$ by $n$. Formally:

$$A \approx_n B \iff |A| = |B| \land \forall i \in [0,|A|): A_i \sim_n B_i$$

Three definitions are given to step by step derive Gauss code equivalency:

Definition 3.3 introduces the concept of uniform Gauss codes. These are Gauss codes that have the same number of components and are connected by a component map. This is the first requirement for two Gauss codes to match to the same link, when two Gauss codes have a different number of components they can never represent the same link.

Definition 3.3 (Uniform Gauss codes) For two Gauss codes $\alpha$, $\beta$ and a bijective component map $m$, the Gauss codes $\alpha$ and $\beta$ are uniform to each other by $m$ iff $\alpha$ and $\beta$ have the same amount of components and $m$ maps each component $A$ from $\alpha$ to a component $B$ from $\beta$. Formally:

$$\alpha \sim_m \beta \iff \#C_\alpha = \#C_\beta \land m : C_\alpha \rightarrow C_\beta$$

The next definition presents congruent Gauss codes, which are uniform Gauss codes that for each pair of mapped components have the same size components. This is an extension of the uniform Gauss code requirement for Gauss code matching. On top of the Gauss codes having the same size, all of their mapped components need to have the same amount of crossings too.

Definition 3.4 (Congruent Gauss codes) For two Gauss codes $\alpha$, $\beta$ and a bijective component map $m$, holds $\alpha$ is congruent to $\beta$ by $m$ iff $\alpha$ and $\beta$ are uniform to each other by $m$, and for every component $A$ from $\alpha$ mapped to component $B$ from $\beta$ it holds that $A$ and $B$ have the same amount of crossings. Formally:

$$\alpha \approx_m \beta \iff \alpha \sim_m \beta \land \forall (A,B) \in m : \#X_A^\pm = \#X_B^\pm$$

The final definition defines Gauss code equivalence by saying that two Gauss codes are equivalent when there is a crossing map, a component map and an offset for each component so that each component of the first Gauss code rotated by its offset is uniform by $n$ to the component it is mapped to in the second Gauss code.

Definition 3.5 (Gauss code equivalence) For two Gauss codes $\alpha$ and $\beta$ it holds that $\alpha$ is equivalent to $\beta$ iff there exists a bijective component map $m : C_\alpha \rightarrow C_\beta$ and a bijective crossing map $n : X_\alpha \rightarrow X_\beta$ such that for each component $A$ of $\alpha$ there exists a positive integer offset which when used to shift $A$ makes $A$ congruent by $n$ to the component from $\beta$ that $A$ is mapped to by $m$. Formally:

$$\alpha \equiv \beta \iff \exists m : C_\alpha \rightarrow C_\beta \exists n : X_\alpha \rightarrow X_\beta \forall A \in C_\alpha \exists \text{offset} \in [0,\#X_A^\pm) : A \oplus \text{offset} \approx_n m(A)$$
3.1.6 Detailed Algorithms

The gcMatch algorithm takes two Gauss codes and a bijective component map that maps components from one Gauss code to the other. By searching for a suitable offset for each pair of components in the component map it establishes if the two Gauss codes are equivalent. After having gone through all component pairs gcMatch returns a boolean saying whether or not the two Gauss codes are equivalent.

\textbf{Algorithm } gcMatch(\alpha, \beta : \text{Gausscode}; m : C_\alpha \rightarrow C_\beta) \quad (* \text{Precondition} : \alpha \approx_m \beta *)
\quad (* \text{Return} : \alpha \equiv \beta *)
1. (\text{* Initialization *})
2. boolean codematch ← true;
3. incMap ← new map: Crossing → Crossing
4. for each \(A \in C_\alpha\)
5. \quad do \(B \leftarrow m(A)\)
6. \quad codematch ← codematch \& findOffset(A, B, incMap)
7. (\text{* codematch } \equiv \forall A \in C_\alpha \exists \text{offset} \in [0, \#X_A^\pm) : A \oplus \text{offset} \approx_{incMap} m(A) *)
8. return codematch

The findOffset algorithm takes two components \(A, B\), and a bijective crossing map. By searching for a suitable crossing matching for each possible offset it establishes if \(A\) can be made congruent with \(B\) by shifting \(A\) by that particular offset. The algorithm expands the given crossing map with additional crossing pairs as the map is passed by reference. Before termination findOffset returns a boolean indicating whether or not a suitable offset was found so that when used to shift \(A\), \(A\) is congruent to \(B\).

\textbf{Algorithm } findOffset(A, B : Component; incMap : Crossing → Crossing) \quad (* \text{Precondition} : \#X_A^\pm = \#X_B^\pm \land incMap \text{ is a bijective map } *)
\quad (* \text{Return} : \exists \text{offset} \in [0, \#A) : A \oplus \text{offset} \approx_{incMap} B *)
1. (\text{* Initialization *})
2. offsetExists ← false
3. for each \(j \in [0, \#X_A^\pm)\)
4. \quad do tempMap ← deep copy of incMap
5. \quad compmatch ← componentMatch(A, B, tempMap, offset)
6. \quad if compmatch (\text{* } A \oplus j \approx_{tempMap} A *)
7. \quad then offsetExists ← true
8. \quad incMap ← deep copy of tempMap
9. (\text{* offsetExists } \equiv \exists \text{offset} \in [0, \#A) : A \oplus \text{offset} \approx_{incMap} B *)
10. return offsetExists
The \textit{componentMatch} algorithm takes two components $A$, $B$, a crossing map $n$ and an integer offset. It returns if the given crossing map $n$ can be expanded so that $A$ with offset projected onto it is congruent with $B$ by $n$. The algorithm adjusts the incoming map which is passed by reference so that it can be used further by its caller.

**Algorithm** \textit{componentMatch}(A, B : component; n : Crossing → Crossing; offset : Integer)

(* Precondition : $|X_A^\pm| = |X_B^\pm| \land offset \in [0, |X_A^\pm|] \land n$ is a bijective map *)

(* Return : $f : n \subseteq f \land A \oplus offset \cong_f B$ *)

1. (* Initialization *)
2. \textit{componentMatch} ← true
3. $i ← 0$
4. while $i < |X_A^\pm|$ do
5. \hspace{1em} (* Invariant: \textit{componentMatch} $\equiv \forall j \in [0, i) : A_j \oplus offset \cong_n B_j$ *)
6. \hspace{2em} $a ← A_i \oplus offset$
7. \hspace{2em} $b ← B_i$
8. \hspace{2em} $D ← |a| \in \text{Domain}(inc)$
9. \hspace{2em} $R ← |b| \in \text{Range}(inc)$
10. \hspace{2em} if $D$ and $R$
11. \hspace{3em} then if $n(|a|) \neq |b|$
12. \hspace{4em} then \textit{componentMatch} ← false
13. \hspace{2em} if $\neg D$ and $\neg R$
14. \hspace{3em} then $n.add(|a|, |b|)$
15. \hspace{2em} if $(\neg D$ and $R$) or $(D$ and $\neg R$)
16. \hspace{3em} then \textit{componentMatch} ← false
17. \hspace{2em} \textit{componentMatch} ← \textit{componentMatch} $\land \text{sign}(a) = \text{sign}(b)$
18. \hspace{2em} $i ← i + 1$
19. (* $i \geq |A| \lor \neg \text{componentMatch}$ $\land \text{componentMatch} \equiv \forall j \in [0, i) : A_j \oplus offset \cong_n B_j$ *)
20. return \textit{componentMatch}

The \textit{r1delete} algorithm takes two Gauss codes $\alpha$ and $\beta$ that are uniform for all but one component which has two more crossings in $\alpha$. It determines if $\alpha$ can be matched to $\beta$ by conducting an 'R1 remove' on $\alpha$. If a suitable R1 move is found a set containing the involved crossing is returned, otherwise an empty set is returned.

**Algorithm** \textit{r1delete}(\alpha, \beta : GaussCode)

(* Precondition : $\alpha \sim_m \beta \land \left( \exists A \in c_{\alpha} \forall A' \in C_{\alpha} \setminus A : |X_A^\pm| + 2 = |X_{m(A)}^\pm| \land |X_{A'}^\pm| = |X_{m(A')}^\pm| \right) $ *)

(* Postcondition : $\exists \alpha' : \alpha \overset{\text{r1}}{\Rightarrow} \beta \Rightarrow \text{ret} = \text{r1}^{-}, \neg \exists \alpha' : \alpha \overset{\text{r1}^{-}}{\Rightarrow} \beta \Rightarrow \text{ret} = \phi$ *)

1. $r1set ← \phi$
2. (* find possible r1 moves *)
3. for each crossing $c \in X_\alpha^\pm$
4. \hspace{1em} if $c \oplus 1 = \neg c$
5. \hspace{2em} then $r1set += (c, \neg c)$
6. (* for each possible r1 move stored in r1set, test if that move transforms $\alpha$ into a Gauss code that matches $\beta$ *)
7. for each pair $(c, \neg c)$ in $r1set$
8. \hspace{1em} $aCopy ← \alpha$
9. \hspace{2em} $aCopy ← aCopy \setminus \{c, \neg c\}$ (* execute r1 remove on aCopy *)
10. \hspace{1em} if $gcMatch(aCopy, \beta)$
11. \hspace{2em} then return $\{c\}$
12. (* if none of the possible r1 moves in $r1set$ were correct r1 moves *)
13. return $\phi$
The \texttt{r2delete} algorithm takes two Gauss codes $\alpha$ and $\beta$ that are uniform for all but one component which has four more crossings in $\alpha$. It determines if $\alpha$ can be matched to $\beta$ by conducting an ‘R2 remove’ on $\alpha$. If a suitable R2 move is found a set containing the involved crossings is returned, otherwise an empty set is returned.

\textbf{Algorithm} \texttt{r2delete}(\alpha, \beta : GaussCode)

\hspace{1em}(* \textbf{Precondition}: $\alpha \sim_m \beta \land \left( \exists A \in C_\alpha \forall A' \in C_\alpha \setminus A : \#X^+_A + 4 = \#X^+_m(A) \land \#X^+_A' = \#X^+_m(A') \right) \*)

\hspace{1em}(* \textbf{Postcondition}: $\exists \alpha \rightharpoonup_2 \beta \Rightarrow ret = r2^{-}, \neg \exists \alpha \rightharpoonup_2 \beta \Rightarrow ret = \phi \*)

1. (* find all possible r2 moves by sorting all consecutive pairs with the same sign in $\alpha$ into 2 sets, one with positive crossings and one with negative crossings *)
2. positivePairs $\leftarrow \phi$
3. negativePairs $\leftarrow \phi$
4. for each crossing $c \in X^+_\alpha$
5. \hspace{1em} if \hspace{1em} sign($c$) = 1 = sign($c \oplus 1$)
6. \hspace{1em} then \hspace{1em} positivePairs $+= (c, c \oplus 1)$
7. \hspace{1em} if \hspace{1em} sign($c$) = -1 = sign($c \oplus 1$)
8. \hspace{1em} then \hspace{1em} negativePairs $+= (c, c \oplus 1)$
9. r2set $\leftarrow \phi$
10. for each $(c, c \oplus 1)$ in positivePairs
11. \hspace{1em} for each $(c', c' \oplus 1)$ in negativePairs
12. \hspace{2em} if \hspace{2em} $c \in (c', c' \oplus 1) \land c \oplus 1 \in (c', c' \oplus 1)$
13. \hspace{2em} then \hspace{2em} r2set $+ = ((c, c \oplus 1), (c', c' \oplus 1))$
14. (* for each possible r2 move stored in r2set, test if that move transforms $\alpha$ into a Gauss code that matches $\beta$ *)
15. for each $((c, c \oplus 1), (c', c' \oplus 1))$ in r2set
16. \hspace{1em} $\alphaCopy \leftarrow \alpha$
17. \hspace{1em} $\alphaCopy \leftarrow \alphaCopy \setminus \{c, c \oplus 1, c', c' \oplus 1\}$ (* execute r2 remove on $\alphaCopy$ *)
18. \hspace{1em} if \hspace{1em} gcMatch($\alphaCopy, \beta$)
19. \hspace{1em} then \hspace{1em} return $\{c, c \oplus 1\}$
20. (* if none of the possible r2 moves in r2set were correct r2 moves *)
21. return $\phi$
The \( r^3 \text{shift} \) algorithm takes two uniform Gauss codes \( \alpha \) and \( \beta \) and determines if \( \alpha \) can be matched to \( \beta \) by conducting an 'R3 shift' on \( \alpha \). If a suitable R3 move is found a set containing the involved crossing is returned, otherwise an empty set is returned.

**Algorithm** \( r^3 \text{shift}(\alpha, \beta: \text{GaussCode}) \)

(\* Precondition: \( \alpha \approx_m \beta \) \*)

(\* Postcondition: \( \exists r^3: \alpha \xrightarrow{r^3} \beta \Rightarrow \text{ret} = r^3, \neg \exists r^3: \alpha \xrightarrow{r^3} \beta \Rightarrow \text{ret} = \phi \) \*)

1. (\* find all possible r3 moves by sorting all consecutive pairs into 3 sets, one with positive crossings, one with negative crossings and one with the remaining mixed crossings \*)
2. \( \text{positivePairs} \leftarrow \phi \)
3. \( \text{negativePairs} \leftarrow \phi \)
4. \( \text{mixedPairs} \leftarrow \phi \)
5. for each crossing \( c \in X^+ _{\alpha} \)
6. if \( \text{sign}(c) = 1 = \text{sign}(c \oplus 1) = 1 \)
7. then \( \text{positivePairs} += (c, c \oplus 1) \)
8. if \( \text{sign}(c) = -1 = \text{sign}(c \oplus 1) = -1 \)
9. then \( \text{negativePairs} += (c, c \oplus 1) \)
10. if \( \text{sign}(c) = 1 \text{xor} \text{sign}(c \oplus 1) = 1 \)
11. then \( \text{mixedPairs} += (c, c \oplus 1) \)
12. (\* Find all Reidemeister triangles \*)
13. for each pair \( (c_1, c_1 \oplus 1) \) in \( \text{positivePairs} \)
14. for each pair \( (c_2, c_2 \oplus 1) \) in \( \text{negativePairs} \)
15. for each pair \( (c_3, c_3 \oplus 1) \) in \( \text{mixedPairs} \)
16. if \( \text{isReidemeisterTriangle}(c_1, c_2, c_3, \alpha) \)
17. then \( \text{r3set} += (c_1, c_2, c_3) \)
18. (\* for each possible r3 move stored in \( \text{r3set} \), test if that move transforms \( \alpha \) into a Gauss code that matches \( \beta \) \*)
19. for each Reidemeister triangle triple \( (c_1, c_2, c_3) \) in \( \text{r3set} \)
20. \( \alphaCopy \leftarrow \alpha \)
21. \( \alphaCopy \leftarrow \text{executeR3}(c_1, c_2, c_3, \alphaCopy) \)
22. if \( \text{gcMatch}(\alphaCopy, \beta) \)
23. then return \( \{c_1, c_2, c_3\} \)
24. (\* if none of the possible r3 moves in \( \text{r3set} \) were correct r3 moves \*)
25. return \( \phi \)

The \( \text{isReidemeisterTriangle} \) algorithm takes three crossings and a Gauss code \( \alpha \) in which they appear and returns whether the given crossings forms a Reidemeister triangle in \( \alpha \). A Reidemeister triangle is a triangle of three crossings that are each other’s neighbors, meaning an R3 move can be executed on them.

**Algorithm** \( \text{isReidemeisterTriangle}(a,c,e: \text{Crossing}; \alpha: \text{GaussCode}) \)

(\* Precondition: \( a, c, e \in X^+ _{\alpha} \) \*)

1. (\* a Reidemeister triangle is formed between 3 crossings if and only if these 3 crossings are all adjacent in the knot diagram \*)
2. \( b \leftarrow a \oplus 1 \)
3. \( d \leftarrow c \oplus 1 \)
4. \( f \leftarrow e \oplus 1 \)
5. return \( \left( \begin{array}{l}
\neg c \in (a, b) \text{xor} - d \in (a, b) \\
\neg e \in (a, b) \text{xor} - f \in (a, b) \\
\neg e \in (c, d) \text{xor} - f \in (c, d) \\
\neg c \in (e, f) \text{xor} - d \in (e, f)
\end{array} \right) \)
The executeR3 algorithm performs an 'R3 shift' on a given Reidemeister triangle in a given Gauss Code

**Algorithm** executeR3\((a,c,e : \text{Crossing}; \alpha : \text{GaussCode})\)

\((\ast\text{ Precondition: isReidemeisterTriangle}(a, c, e, \alpha) \text{ and } a, c, e \in X_\alpha^\pm \ast)\)

1. swap\((a, a \oplus 1)\) in \(\alpha\)
2. swap\((c, c \oplus 1)\) in \(\alpha\)
3. swap\((e, e \oplus 1)\) in \(\alpha\)
4. return \(\alpha\)

The algorithms listed above are implemented in KnotWeaver, they are used to allow editing under knot equivalence. During execution each action on a Link is checked to maintain that knot equivalence by comparing the Gauss code from before and after the action. If an action has before and after Gauss codes that are unmatchable by any Reidemeister move it is not allowed. If the Gauss codes can be matched with a Reidemeister move then that move is executed and the information given by the used algorithm (\(r1\text{delete}, r2\text{delete}\) or \(r3\text{shift}\)) is displayed. In case multiple Reidemeister moves are possible (for example an R2 add can pull a strand over or under another strand) it is possible to browse through the different moves.

### 3.2 Seifert Circles

This chapter describes the concepts and algorithms needed to fulfill requirement VR01. Seifert circles are a property that can be viewed and studied on its own, but as we will later see they will also prove useful in an algorithm to create a knot layout from a Gauss code.

#### 3.2.1 Introduction

Seifert circles are circles that originate from removing the crossings in a link by connecting all incoming strands to adjacent strands leaving the crossings (see Figure 1.5). When applied on a knot diagram a set of non-intersecting oriented circles appears. One of their purposes is to construct braid diagrams from a knot diagram, but they are also used to construct Seifert surfaces (which is out of the scope of this project so we will not go in detail on Seifert surfaces). When all Seifert circles are concentric and have the same orientation the knot diagram can be easily transformed into a braid diagram. Of course Seifert circles are not always concentric and oriented in the same way (all clockwise or all counter-clockwise). There is however an algorithm described by Yamada and Vogel[Yam87][Vog90] that transforms a knot diagram by means of Reidemeister moves such that all Seifert circles are concentric and oriented in the same way. This algorithm can be used to obtain special knot diagrams which are easily transformed to braid diagrams.
3.2.2 Representations

Multiple representations for Seifert circles exist, aside from the visual representation two textual representations can be distinguished. The first one lists each unsigned crossing per Seifert circle. A second notation lists pairs of signed crossings that represent a particular part of the knot between two crossings.

Example from the knot in Figure 1.5 on Page 8:

**Gauss Code:** \[[+1, -2, +3, -4, +2, -1, +4, -3]\]

The single notation lists for each Seifert circle which crossings are in that Seifert circle and shows in which order they occur in the circle. It does not matter which crossing is written down first as a circle is represented, but usually it is the crossing with the lowest number that is written down first.

**Single Notation:**
\[
S_1 : 1, 2 \\
S_2 : 3, 4 \\
S_3 : 1, 4, 2, 3
\]

A more complex notation for Seifert circles is the paired notation. This notation lists the parts of the link between the crossings of the Seifert circle. The result is a list of pairs that say from which strand of one crossing to which strand of another crossing part of the Seifert circle goes.

**Paired Notation:**
\[
S_1 : (+1, -2), (+2, -1) \\
S_2 : (+3, -4), (+4, -3) \\
S_3 : (-1, +4), (-4, +2), (-2, 3), (-3, 1)
\]

Note that in the single notation one cannot always distinguish all Seifert circles from each other. When two Seifert circles contain the same crossings only the paired notation can show the difference between them. For example imagine the unknot with a R1 twist: \[[1, -1]\], in single notation this yields \(S_1 : 1, S_2 : 1\) but in paired notation the difference becomes clear: \(S_1 : (1, -1), S_2 : (-1, 1)\).

The notations for crossings is extended for Seifert circles.
\(X^s_s\) : All signed crossings contained in Seifert circle \(s\)

In the the next section (3.2.3) the algorithms that find the Seifert circles in a knot are given.
### 3.2.3 Algorithms

The `getSeifertCircle` algorithm creates and returns the Seifert circle starting at crossing $x$ that contains a given crossing in a given Gauss code. It does this by walking along the link and each time a crossing is encountered switching to the other strand that passes through that crossing until it finally reaches $x$ again. Note that $x$ will always be reached again because the $\oplus$ (successor) function is bijective, meaning that it is impossible to get stuck in a loop that does not contain $x$. This ensures the algorithm `getSeifertCircle` terminates.

**Algorithm** `getSeifertCircle(seed : Crossing; \alpha : GaussCode)`

1. $sCircle \leftarrow \phi$
2. $x \leftarrow seed$
3. repeat
4. $y \leftarrow x \oplus 1$
5. $sCircle += (x, y)$
6. $x = -y$
7. until $|seed| = |x|$
8. return $sCircle$

The `getAllSeifertCircles` algorithm takes a Gauss code and returns a set of Seifert circles.

**Algorithm** `getAllSeifertCircles(\alpha : GaussCode)`

1. $unusedCrossings \leftarrow X_\alpha^\pm$
2. $set \leftarrow \phi$
3. while $unusedCrossings \neq \phi$
4. (* Invariant: $\forall s \in set \forall c \in X_s^\pm : c \notin unusedCrossings$ *)
5. do $c \leftarrow$ an element of $unusedCrossings$
6. $s \leftarrow getSeifertCircle(c, \alpha)$
7. $unusedCrossings \leftarrow unusedCrossings - X_s^\pm$
8. $set += s$
9. return $set$

### 3.3 Knot Layout Creation from Gauss Codes

#### 3.3.1 Introduction

The creation of a link from a Gauss code is a complex problem. Finding the essential form (that is finding which lines cross which) of the link is not hard, finding a suitable layout in the plane is. The main problem with such a layout is that it must not create extra crossings, so apart from the crossings that are in the Gauss code the rest of the knot has to be planar. The algorithm below gives a solution to this problem. The algorithm however is purely conceptual and some implementational and minor conceptual gaps still need to be covered.
3.3.2 Algorithm Overview

The goal of the Layout algorithm is to create a knot diagram from a complete Gauss code with a good looking embedding in the plane. The key feature the following algorithm uses is the fact that Seifert circles don’t intersect each other due to the way they are constructed. This base of planarity can be extended to create a layout of a knot without creating extra crossings. The here presented algorithm consists of six conceptual steps.

**Step one:** Determine the Seifert circles from the Gauss code as described in section 3.2.

**Step two:** Construct a Seifert graph (defined later in the elaboration of this step) based on the connectivity of Seifert circles in the knot.

**Step three:** Collapse nodes in the Seifert graph that represent embedded Seifert circles. This is done now to later create embedded Seifert circles when the actual layout is formed.

**Step four:** Find a planar embedding for the collapsed Seifert graph.

**Step five:** Create a global layout based on the graph embedding of the Seifert graph.

**Step six:** Connect the Seifert circles with crossings by using the orientation information stored in the complete Gauss code.

Different embeddings for the same complete Gauss code are possible (as demonstrated in Figure 3.5) but the layout algorithm attempts to minimize the needed space by collapsing as many nodes as possible in **Step three** in order to obtain as many embedded Seifert circles in **Step six** as possible. One of the results of this embedding ambiguity is that the orientation of Seifert circles cannot be inferred directly from the complete Gauss code (as confirmed by [Bar04]).

![Image of Gauss code and Seifert circles](image)

**Figure 3.5:** One complete Gauss code can lead to different knot diagrams, with different Seifert circle embeddings, of the same knot
3.3.3 Detailed Algorithms

To facilitate the reading of the algorithms we use a running example based on the Gauss code (Another less generic example can be found in Appendix B):

\[
\alpha = [[-1, +2, -3, +4][+1, -4, +5, -5, +3, -2]/--++]
\]

We will present the algorithms in a top-down fashion.

The *makeKnotDiagram* algorithm creates a knot diagram from a complete Gauss code as described in the six steps of Algorithm Overview section. This is the main algorithm.

**Algorithm** *buildKnotDiagram*(\(\alpha\): *CompleteGaussCode*)

1. Set \(<\text{SeifertCircle}>\) \(scs \leftarrow \text{getAllSeifertCircles}(\alpha)\) (* see section 3.2.3 *)
2. SeifertGraph \(sg \leftarrow \text{buildSeifertGraph}(scs)\)
3. \(\text{collapseGraph}(sg)\)
4. \(\text{planarEmbed}(sg)\)
5. \(kd \leftarrow \text{createLink}(sg)\)
6. \(\text{addCrossingsToDiagram}(kd, \alpha)\)
7. \(\text{return} \ kd\)
Step one: Determining Seifert circles

By applying the `getAllSeifertCircles` algorithm on $\alpha$, as described in section 3.2.3 the following Seifert circles are obtained for our running example:

- $S_1 : 5$
- $S_2 : 2, 1$
- $S_3 : 2, 3$
- $S_4 : 4, 1$
- $S_5 : 4, 5, 3$

Step two: Construction of the Seifert graph

The `buildSeifertGraph` algorithm builds a Seifert graph. A Seifert graph is a graph representing the structure of the Seifert circles, this is done by representing each Seifert circle with a node and each crossing with an edge. When two Seifert circles share a crossing then the nodes that represent those circles are connected by an edge that represents the crossing.

Algorithm `buildSeifertGraph(scs : Set<SeifertCircle>)`:

1. $sg \leftarrow$ new SeifertGraph
2. for each SeifertCircle : $S$ in $scs$
3. do Add a node in $sg$ corresponding to $S$
4. for each Crossing : $c$ in $scs$
5. do Find the Seifert circles that $c$ belongs to: $S_1$ and $S_2$
6. Add an edge in $sg$ corresponding to $c$ between the nodes corresponding to $S_1$ and $S_2$
7. return $sg$

Figure 3.6: The Seifert graph of $\alpha$. 
Step three: Collapsing the Seifert graph

The \textit{collapseGraph} algorithm takes a graph and folds as many nodes as possible into their neighboring nodes. Folding one node \( n \) into another \( n' \) means that \( n \) is removed from the graph and \( n \) becomes a subnode of \( n' \). As mentioned before this is done to later create embedded Seifert circles. A Seifert circle \( n \) can be embedded in another Seifert circle \( n' \) if \( n \) is connected only to \( n' \) in the Seifert graph and \( n' \) does not already have other neighboring nodes folded into it. In an attempt to maximally collapse as many nodes as possible into other nodes this process is iteratively attempted on each node that receives new subnodes until either the node has multiple neighbors or its neighbor already has subnodes. The result is what is called a \textit{collapsed Seifert graph}.

Algorithm \textit{collapseGraph}(G : Graph)
1. \( N \leftarrow \) The set of nodes in \( G \)
2. \textbf{while} there exist a node: \( n \) in \( N \), with degree 1
3. \hspace{1em} \textbf{repeat}
4. \hspace{2em} \( n' \leftarrow \) the neighbor of \( n \)
5. \hspace{2em} \textbf{if} \( n' \) has no subnodes
6. \hspace{3em} \textbf{then} \( n \) becomes a subnode of \( n' \)
7. \hspace{3em} \( N \leftarrow N \setminus n \)
8. \hspace{3em} \( n \leftarrow n' \)
9. \hspace{2em} \textbf{else} \( N \leftarrow N \setminus n \)
10. \hspace{1em} \textbf{until} \( n' \) has subnodes \textbf{or} \( n' \) has degree \( \neq 1 \)

In the running example only one node can be folded into its neighbor and that is the one corresponding to \( S_1 \) into its neighbor corresponding to \( S_5 \). All other nodes have more than one neighbor and hence cannot be collapsed. The resulting collapsed Seifert graph looks as follows:

![Figure 3.7: The collapsed Seifert graph of \( \alpha \)](image_url)
Step four: Find a planar embedding

The $\text{planarEmbed}$ algorithm creates an aesthetically pleasing planar embedding of the graph.

**Algorithm** $\text{planarEmbed}(g : \text{Graph})$
1. Apply PC-Tree planarity algorithm [SH99] on $g$
2. Apply a force-directed graph layout algorithm on $g$ that preserves planarity and gives room to nodes based on the amount of subnodes

![Figure 3.8: The planar embedding of the collapsed Seifert graph of $\alpha$](image)

In this step a planar embedding must be found for the Seifert graph, to make **Step six** easier it is useful to use a straight line planar embedding algorithm because it makes sure there is a direct path between two Seifert circles in the final knot layout. 'A straight line drawing is a drawing of a graph in which each edge of the graph is drawn as a straight line segment' [NR04] This is possible for each planar graph [Wag36]. To improve the graphical layout a force-directed graph layout algorithm, that preserves planarity and gives more room to nodes with heavier weights, can be applied to the planar embedded graph. However the question remains if such a force-directed algorithm exists. We know there are force-directed algorithms that preserves planarity [Ber99], and that there are force-directed algorithms that take weights into account. But it is is unclear if a force-directed algorithm with both properties exists. This is something that needs to be investigated in future work.
Step five: create the knot layout

The \textit{createLink} algorithm transforms the processed Seifert graph into a diagram which looks very similar to the end layout. All Seifert circles are drawn, including the embedded ones.

The \textit{createLink} algorithm creates a knot diagram from a simplified Seifert graph by first determining the radius of all Seifert circles with the \textit{determineOptimalRadii} algorithm and then filling the available radius equally with its subnodes. Because the \textit{determineOptimalRadii} algorithm produces circles that touch each other the radii need to be scaled down to leave room for crossings later on. A value of 90% of the original size is chosen.

\textbf{Algorithm} \textit{createLink}(sg : SeifertGraph)
\begin{enumerate}
    \item \( kd \leftarrow \text{new KnotDiagram} \)
    \item \( \vec{p} \leftarrow \text{The geometric location of each node in } sg \)
    \item \( w \leftarrow \text{The number of subnodes of each node in } sg \text{ plus one} \)
    \item \( r \leftarrow 90\% \times \text{determineOptimalRadii}(p, w) \)
    \item \textbf{for} each node \( scNode \) in \( sg \)
    \item \( \quad \vec{p} \leftarrow \text{The location of } scNode \)
    \item \( \quad n \leftarrow \text{The number of subnodes of } scNode \)
    \item \( \quad r \leftarrow \text{The radius of } scNode \text{ as attributed by } \text{determineOptimalRadii} \)
    \item \( \quad \text{Create a circular component in } kd \text{ with as center point } p \text{ and radius } r \)
    \item \( \quad \text{Create } n \text{ equidistant concentric components around } p \text{ within } r \)
\end{enumerate}

\textbf{Figure 3.9:} The global layout of the knot

Given a simplified Seifert graph with a number of points that determine the center points of the representing Seifert circles (delivered by the planarEmbed algorithm), the radii for each circle have to be determined. the \textit{determineOptimalRadii} algorithm takes the set of center point locations and a weight equal to the number of subnodes plus one (one for the outer circle too) and returns a radius for each point, so that none of the circles overlap and that the radii of the circles are proportional to their number of subnodes. This means that whenever two circles are not hampered by the proximity of another circle their size will be in proportion to their relative weights. An algorithm for this problem is presented in Appendix A.
Step six: Connecting the Seifert circles with crossings

In the final step the crossings are added by the `addCrossingsToDiagram` procedure. The orientation of the different Seifert circles, which was previously undecided, follows from the orientation of the crossings.

![Diagram](image)

**Figure 3.10:** The knot diagram of the Gauss code `cgc` as created by the layout algorithm

The `addCrossingstoDiagram` algorithm takes a knot diagram without crossings and a complete Gauss code and adds the crossings as listed in the complete Gauss code to the knot diagram, in this stage proper orientation is added to each crossing. This step finalizes the knot diagram.

**Algorithm** `addCrossingstoDiagram(kd: KnotDiagram, α: CompleteGaussCode)`

1. for all crossings in $\alpha$
2. do Add the crossing to the knot diagram between the appropriate Seifert circles and set the orientation to correspond with the crossing information in $\alpha$

Some work concerning the precise implementation of this last step is needed. Such as where precisely between two Seifert circles is the ideal place to add crossings and how to derive the Seifert circle orientation.

This concludes the layout algorithm. Not all questions have been answered but a general method has been described that makes it possible to create a working implementation in KnotWeaver that solves the problem of creating a knot layout from a complete Gauss code.
Chapter 4

Design

Due to the scope of the requirements it is very important that further additions can be made to the program beyond this project. Naturally the application implements the needed functionality, but it is likely that ideas for additional functionality will pop up later. It is therefore important that the application code is easily reused and extended.

To achieve this goal a 2-level design is created: At the bottom of the application is the KnotEngine, this package of classes represent the relevant mathematical concepts used in knot theory. On top is the user interface which is divided into two parts: a display part which allows visualization, and a functionality part which allows various plugins. These plugins implement and separate different sections in the requirements and in usage.

Because manipulation of a knot structure is so important, the design is chosen to allow a system similar to the interaction model of KnotPlot [Sch98] where Knots consist of a number of connected points in space. This design opts for a similar structure but in 2D. In this system a knot consists of a number of components each built from connected labels. Figure 4.2 shows a knot built from 5 labels, each marked by an arrow indicating the orientation. Each label represents a key point in the link, breaking up the concept of a strand into discrete pieces of lines and points. These points and lines can later be used for interaction.
Figure 4.2: A screenshot of how knots appear in KnotWeaver

4.1 KnotEngine

The KnotEngine delivers a set of classes with carefully designed API. The importance of an API is twofold: on the one hand it allows other programmers to work easily with predefined methods and on the other hand new implementations can be written for old methods, without having to change the classes that use them.

The link data structure consists of 4 classes, each representing certain functionality regarding links. The KnotEngine also contains a number of classes that support these classes in their functionality but will not be explained here. For further details consult the JavaDoc, delivered with the software.

The 4 main classes of the KnotEngine:

- Link
- ProjectedLink
- ConstrainedLink
- UndoableLink

The first 3 classes each represent a mathematical concept while the bottom class serves to implement editing functionality.
4.1.1 Link

The class Link corresponds with the mathematical concept of a link. Link is a template class with as parameter the class type for labels. This is done to maintain genericity because location information is not yet incorporated in the Link class. To achieve the concept of a link, two sub data structures are used:

- A set of components, each representing the order of labels in a single knot.
- A map which maps each line between two consecutive labels to a set containing all crossings on that line, sorted by order from the begin point.

Labels are used to mark the beginning and end of strands in the link, where consecutive strands share a common begin and end label where they connect to each other. Any type of label can be used but it is important to note that Link itself has no notion of location or position of the labels. The labels only function as a unique identifier to structure the link and its components. This means crossings cannot be automatically inferred from the location of labels and hence have to be separately entered into the data structure.

The conceptual actions the class Link can perform on the data structure are:

- Add label before or after labels in existing components or as the first label of a new component.
- Add crossings between two lines or invert existing crossings
- Remove labels and crossings
- Open and Close components
- Invert the orientation of a component

Furthermore there are a number of retrieval methods that return various types of data about the link; these are documented in the JavaDoc, delivered with the software.

Figure 4.3: a) The global KnotEngine structure b) the detailed view of the Link class with template E
4.1.2 ProjectedLink

To visualize a link on a two dimensional surface each label has to be given a position on the plane. This translates to creating a child class from Link with a two dimensional point as the label type. From now on the geometry of the link can be determined by the location of its labels and crossings can be automatically inferred from the lines that geometrically form them. This information allows ProjectedLink to support the translation of points and lines.

The conceptual actions the class ProjectedLink can perform on the data structure are:

- All actions from Link, extended with automatic crossing location information
- Translation of lines and points

Furthermore there are a number of retrieval methods that return various geometrical data about the link, these are documented in the JavaDoc, delivered with the software.

4.1.3 ConstrainedLink

The sole purpose of this class is to guard the visual correctness of a ProjectedLink, therefore every action inherited from ProjectedLink that ConstrainedLink can perform is guarded by the following 3 constraints:

- No two non-equal points may be closer to each other than a certain distance
- No point and line may be closer to each other than a certain distance
- No line may pass through a location which is closer to two lines at the same time than a certain distance

These rules guarantee that a viewer can uniquely identify the visualized ConstrainedLink without any ambiguities as to which line crosses which when rendered. When an action that would violate the constraints is performed it is not executed and an exception, detailing what the objection is, is thrown. For example when a point is moved too close to a line the PointNearLineException is thrown.

4.1.4 UndoableLink

An UndoableLink adds the undo and redo functionality to a ConstrainedLink. Each editing action stores a new inverse action in a history list. Undos and redos scroll through this list performing the inverse and original actions respectively. Performing a new edit while not at the end of the history list throws away all edits currently undone but not redone. The design of the undo functionality is set up around the ‘Command’ design pattern [FFBS04].

Figure 4.4: A simplified version of the command pattern as used in UndoableLink
UndoableLink has a number of private inner classes that inherit from KnotAbstractUndoableEdit. These classes all contain the ability to execute, undo and redo their particular action. In Diagram 4.4 Undoable...Command is a place holder for all the different commands in UndoableLink (i.e. UndoableAddPoint, UndoableTranslateLine, UndoableInvertComponent,...)

4.2 User Interface

4.2.1 Overview

The user interface has to be flexible, extendable and informative. For instance, the system should not be limited to use a single KnotRenderer and a multitude of options has to be supported. However in order to preserve simplicity only one renderer may be displayed at one time and options should be shown only when needed. At any time one link can be drawn by a KnotRenderer on a DisplayPanel (See Figure 4.5), actions can be performed on that link by means of mouse interaction and an options panel from the active KnotPlugin. To structure the way the user browses through the options, the option window is divided into conceptual tabs, each containing a set of options that are related to the same conceptual step of editing or data retrieval. Much in the same way the DisplayPanels are ordered in a tabbed window where only one link is displayed at the same time, with the others directly accessible through tabs. This creates an intuitive interface for beginning users that still allows advanced control for expert users. On the GUI panel can be indicated what the resulting action is of a particular mouse action, in order to help beginning users use the mouse interface. The user interface functions as a bridge between these two subwindows allowing them to control each other when needed.

Figure 4.5: The user interface with all its components
4.2.2 UML Diagrams

Figure 4.6: Overview

Figure 4.6 shows the UML diagram connecting the user interface layer and the plugin layer. At the top is a JSplitPane which represents the GUI containing two tabbed panes, one DisplayPane to display all links and one PluginPane to display all plugin option panels.

Figure 4.7: The display side

Figure 4.7 shows the extended diagram connected to DisplayPane. The DisplayPane contains one or more DisplayPanels which in their turn contains one UndoableLink and one KnotSelection.

Figure 4.8: The plugin side

Figure 4.8 shows the extended diagram connected to PluginPane. The PluginPane is the only class, where KnotPlugin, KnotRenderer, PluginGuiPanel and KnotMouseHandler are interfaces which are to be implemented by plugin packages. Each plugin package should contain at least a KnotPlugin and a PluginGuiPanel. These classes are used to respectively provide the functionality and accessibility to this functionality for the user. The KnotRenderer and KnotMouseHandler need not be new classes implemented by the plugin package as they can be easily reused from the default or other plugin. The KnotMouseHandler handles prepacked events that originate from the active DisplayPanel (see Figure 4.7)
The *KnotPlugin* is the core of a plugin, it supplies the standard necessary functions for a plugin to function in the user interface. When creating a new plugin the final functionality ends up in a class that implements the *KnotPlugin* interface. In order to implement its visual functionality a plugin can use its own *KnotRenderer* to draw in the active *DisplayPanel*. More information about how to create new plugins can be found in Appendix C.

User interaction is provided through the *KnotMouseHandler* where the plugin can choose how to act on preprocessed user commands.

![Figure 4.9: The KnotMouseHandler interface](image)

Figure 4.9 shows the different mouse events that are generated by the active *DisplayPanel*. These mouse events are passed to the active *KnotPlugin* which decides what action to execute based on its KnotMouseHandler. This setup makes sure that mouse interaction is uniform over all *KnotPlugins* but that Knotplugins can still customize how they respond to that mouse interaction.
Chapter 5

Implementation

5.1 Platform

As described in the non-functional requirements the project is built in Java [Jay]. Java is an Object-Oriented programming language developed by Sun Microsystems [Sun] in the 1990’s. The basis of Java is an Objected-Oriented approach to programming and platform independence. The result of Java’s platform independence is that Java code needs to be compiled only once in order to produce an executable that can be run on multiple platforms. An added advantage is that Java programs program are easily ported to a web applet so users can access the software without having to have it installed on their computer system.

The Java version used for this project is J2SE 5.0 (1.5.0) and the Integrated Development Environment (IDE) is NetBeans 5.5 [Net].

5.2 Metrics

Source code metrics are measure on source code and are designed to detect general trends, programmer productivity, error probability, etc. The only catch is that although the data is objective, the interpretation is not. Software metrics have made their introduction into computer science with the idea that it would have the same beneficial effect as measurement had on other engineering sciences. “You can’t control what you can’t measure” [DeM86] is often quoted in this context. However criticism on the use of software metrics have been numerous, claiming them to be inaccurate or even unethical when used to steer programmers behavior. The metrics listed below were not used to guide or steer the project, they are an interesting curiosity for those who would like to get a general image of the size and complexity of the source code.

<table>
<thead>
<tr>
<th>Source Files</th>
<th>71</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directories</td>
<td>13</td>
</tr>
<tr>
<td>executable source lines of code (SLOC)</td>
<td>7593</td>
</tr>
<tr>
<td>Comment lines of code (CLOC)</td>
<td>1958</td>
</tr>
<tr>
<td>Comment Density (CLOC/SLOC)</td>
<td>25.7%</td>
</tr>
<tr>
<td>Test files</td>
<td>6</td>
</tr>
<tr>
<td>Test source lines of code (TSLOC)</td>
<td>815</td>
</tr>
<tr>
<td>Test comment lines of code (TCLOC)</td>
<td>196</td>
</tr>
<tr>
<td>Test Density (TSLOC/SLOC)</td>
<td>10.7%</td>
</tr>
</tbody>
</table>
The entire project is about 10,000 lines of source code of which 25% is not executable code but commented code, spread over 71 source files (which, because the project is written in Java is also the number of classes). There are in total 6 unit test files which take up about 1,000 lines of code, good for 1/10th of the source code.

5.3 Software Quality

5.3.1 Introduction

Compared to other engineering disciplines, computer science is a relatively new player in the field. Even though it has made remarkable progress over the last decades it is still often plagued by issues related to software quality. To end users this usually translates to programs with numerous bugs and for developers it means having to deal with complex, hard to modify legacy code. The rapid expansion of computer science and the gap it left behind in quality is now slowly being bridged by new research and the development of techniques that aid in the manufacturing of high quality software.

Software quality is sometimes split up in two dimensions, Internal vs External qualities and Process vs Product qualities. These are closely related to each other and it is hard to draw a line between them. But they can be used to quantify the importance and impact of the techniques that are tailored to handle them. Users usually do not care very much how a product is created and how its internal mechanics work, their main concern is the product itself and its external properties. On the other side developers should care a great deal about internal and process qualities because these create a base on which the external and product qualities are built. Figure 5.1 shows how the different software qualities are related by arrows indicating influences.[GJM91]

![Diagram of software quality dimensions](image)

**Figure 5.1**: The relations between the different dimensions of software quality

5.3.2 Process

The first way to improve software quality is to structure and improve the development process. A standard way of doing this is by applying the Waterfall model [Roy87] where requirement analysis, design, implementation and testing follow each other in a linear way. The biggest advantage of this approach is that problems can be identified early when the time-cost is still low. Though simple and disciplined the application of the waterfall model is mostly limited to stable projects where the requirements remain unchanged during its duration. Indeed problems arise when external factors come in to play and the requirements are changed or new insights on them are gained. In the waterfall model there is no way back to earlier stages of the development.

The Spiral model [Boe89] offers an answer to the problems with the waterfall model. By looping through the different phases of the waterfall model, earlier project phases are revisited, revaluated and adjusted to act upon unstable elements of the project. The main advantage is that the course of the project can be adjusted at different points in the project.

In the case of this project the spiral model is preferred over the waterfall model because of the
unstable nature of the project and the lack of reflection on the project. The idea is to create a gradual expansion of functionality when looping through the different phases of the waterfall model while user feedback can be incorporated into new requirements and new functionality.

5.3.3 Design by Concepts

When designing the internal structure of a computer program it is a good idea to re-use concepts that have proven themselves already, much like in constructions where not for every house a new kind of brick is invented. This ensures that the benefits of earlier projects and research can be reaped without having to reinvent every little piece over and over again.

Design Patterns

A design pattern is a generalized software solution to a particular reoccurring software design problem. It does not provide a finished design which translates directly to an implementation but rather gives a template on how to go about to solve a particular problem. Examples of used design patterns are the Command pattern and the Observer pattern which are respectively used to create undo/redo functionality and to create event handling in an updating user interface.[FFBS04][GHJV95]

Object-Oriented Programming

Object-oriented programming (OOP) is a style of programming that uses objects to capture particular conceptual functionality in order to improve modularity in a program. It uses a number of concepts which mimic the structure of concepts in the real world which makes it easier for a programmer to reason with the relationships between objects. A first advantage of OOP is that it allows for the separation of concerns, concepts which are only loosely related to each other can be separated into different objects in such a way that reasoning about them individually is possible, the result is that independent functionality can be achieved. A second advantage is that often Design Patterns are expressed in an OOP concepts and relations which makes it easier to translate one to the other, facilitating the implementation from design to code. This project is implemented in Java which leans very heavily on the OOP paradigm as one of its founding pillars.

Plugins

A plugin is an encapsulated piece of software that cannot run on its own but requires a host program to run in. It provides specific functionality through an interface specified in advance. The plugin paradigm requires the host program to supply an interface into which it can ‘plug in’ and provide its functionality. Plugins are usually written for a specific host program, but it is not unthinkable that multiple host programs that provide the same interface could make use of the same plugin. In this project plugins are used to provide built in functionality but also as a way to extend and maintain future functionality.

5.3.4 Code Testing

Even with a perfectly designed architecture and flawlessly proven algorithms, faults in software can still occur during the translation from design to implementation code. To filter out these
faults a procedure called *Unit Testing* is applied to the source code. The application is divided up into testable parts called units. These units are then subjected to a number of tests which compare the output from particular inputs to a reference output verified by the requirements. Because unit tests only depend on the requirements they can be written before the actual code is produced. The benefits of this testing procedure are two fold, on the one hand at a later time when programming code is refactored the same unit tests can be ran to confirm if the refactored code still confirms to the specifications of the requirements, facilitating adjustability of the code. On the other hand it creates confidence in the correctness of the implementation.

In this project code testing was used as code verification after it was written. The general procedure was to write test code after the source code was written and if there were any faults they would be fixed. If later bugs were detected during active execution that were not yet covered by the test code, then test code was added to cover the newly discovered case. Because the test code was written after the source code it was not used as a way to formalize input and output relations (which would be recommended).
Chapter 6

Conclusions and future work

6.1 Conclusions

At the end of this project a program has been realized that is a basis for an extensive knot theory program. The functionality of this program includes the interactive manipulation of knots under the constraints of Reidemeister moves and supplies Gauss code and Seifert circle information. A software engineering approach was taken in accord with the spiral model paradigm. First an early prototype of user interaction was developed. The next step was to create a well formed design that would best implement this interaction and fulfill the requirements of the program. From this design a second prototype was created. That second prototype was continuously expanded with functionality and the interaction was refined based on user feedback.

The current version provides part of the functionality as specified in the requirements (See the table below) but expanding work is still needed. This document provides the details of the used algorithms and the design of the program, which can be used to further develop KnotWeaver.
Input Requirements | Implemented
---|---
IR01 Manual Input | Yes
IR02 Gauss Code | No*
IR03 KnotPlot format | No
IR04 Braid notation | No

Output Requirements
---
OR01 Gauss Code | Yes
OR02 Visual Output | Yes
OR03 KnotPlot format | No
OR04 Braid notation | No

Editing Requirements
---
ER01 Manual editing that does not change the knot | Yes
ER01a Dragging of the knot limited to Reidemeister moves | Yes
ER01b Direct information about Reidemeister moves during editing | Yes
ER02 Manual editing that changes the knot | Yes
ER02a Full edit control over dragging | Yes
ER02b Swapping crossings | Yes
ER02c Change orientation of a component | Yes

Visualisation Requirements
---
VR01 Seifert Circles | Yes
VR02 Yamada-Vogel Algorithm | No**
VR03 Smoothening options for the link | No

*: A theoretical solution is given in the report, but is not implemented in KnotWeaver.
**: A non documented plugin that makes use of a GAP-server to retrieve braid words is imple-mented.

A similar analysis can be performed for the non-functional requirements.

**Interface:** KnotWeaver’s interface is simple, the editing functionality is available as much as possible through buttons on the option panel and the effect of mouse actions are listed on the option panel. Functionality is grouped per tab in on the PluginPane.

**Platform:** The application is written in Java and as such is platform independent.

**Expandability:** The plugin system makes it possible to add new functionality. Together with the interface documentation it is possible to create new plugins with relative ease.

**Documentation:** All essential classes in the KnotEngine are documented as well as all interface classes that are needed to create new plugins.

**Testing:** JUnit tests are created and run for the essential KnotEngine classes.

### 6.2 Future Work

Some work is still needed on both a research level and an implementation level.

Some research questions have to be answered, for example determining whether a force-directed graph layout algorithm that preserves planarity and allows different weights for individual points exists. Such an algorithm would be very helpful in the layout algorithm.

Another subject to study would be to allow crossings to play a less implicit role in the design, at the moment the crossings are derived implicitly from line locations in all classes from ProjectedLink and downwards. A more explicit role for crossings could have advantages in some applications such as creating a smooth representation of a link. In a sense crossings already play a superior role in the Link class, so this class could be used as a basis for such an implementation.

Concerning the implementation, the layout algorithm needs to be finalized in a working imple-
mentation. Some work is needed on filling in some details of this algorithm, for example on where specifically to add certain crossings in a knot diagram. Furthermore in the area of the user interface some improvements are welcome to make the user interface look better. Also currently the plugin system is not implemented to work with adding plugins without rebuilding the entire package. In order to allow proper extendability the plugin system needs to be expanded to simple adding of new plugins.
Appendix A

An Algorithm for Determining the Radii of Seifert Circles

The `determineOptimalRadii` algorithm takes a set of points in the plane with positive weights and determines a radius for each point making sure that an optimal radius ratio between points is achieved whenever possible in such a way none of the circles overlap. It does this by assigning a grow speed to each point equal to its weight. Next it takes the points and their grow speeds and calculates which two points would grow to touch each other first. Then it grows these points with their respective grow speeds until they touch. At that point the grow speeds of both points are set to 0. This growing phase is repeated until all points have grow speed 0. In this stage of the algorithm all points have grown to touch at least one of their neighboring points.

Algorithm `determineOptimalRadii(p : Vector<Points>; w: Vector<Integer>)`

1. $s_i ← w_i$
2. $r_i ← 0$
3. while $∃p_i : s_i ≠ 0$
4. do $t_{min} ← +∞$
5. $M ← ∅$
6. for each $p_i : s_i ≠ 0$ and $p_j$
7. do $g ← \frac{d(i,j)−r_i−r_j}{s_i+s_j}$
8. if $t < t_{min}$
9. then $t_{min} ← t$
10. $M ← \{p_i, p_j\}$
11. for each $p_i ∈ M$
12. do $r_i ← t * s_i$
13. $s_i ← 0$
14. return $r_i$
Figure A.1: The determineOptimalRadii algorithm on an example set of 4 points

Figure A.1 a) shows an initial state with 4 nodes each with a given weight. In this case $d(v,x)$ is the smallest between any two points.
b) shows the situation after the first growing phase, $v$ and $x$ have found their final radii and the next growing phase is prepared. In this case $z$ will be the first to hit $x$ so it is grown until it does.
c) displays the situation after $z$ has completed its expansion. $y$ is now the only remaining point left unexpanded, at its current growing speed it will reach the edge of $z$ first, so it is expanded until it hits $z$.
d) demonstrates the final state. All radii are grown to their maximum potential, their grow speeds are all 0 and the algorithm terminates.
Appendix B

Another Running Example of the Layout Algorithm

Gauss code:

\[ \alpha = \]
\[ [[+1, -2, +3, -4, +5, -1, +6, -3, +7, -5, +8, -9][+2, -6, +4, -7][+9, +10, -10, -8]/ - + - - - + + + -] \]

Step one: Determining Seifert circles

Seifert Circles:

- \( S_1 : 10 \)
- \( S_2 : 2, 3, 7 \)
- \( S_3 : 8, 9, 10 \)
- \( S_4 : 6, 4, 5, 8, 9, 1 \)
- \( S_5 : 4, 7, 5, 1, 2, 6, 3 \)
Step two: Construction of the Seifert graph

Figure B.1: The Seifert graph of Gauss code: $\alpha$

Step three: Collapsing the Seifert graph

The Seifert graph is collapsed into a collapsed Seifert graph by the \textit{collapseGraph} algorithm. Because the Seifert graph represented a linear path all nodes are collapsed into one node: $S_2 \subset S_5 \subset S_4 \subset S_3 \subset S_1$

Step four: Find a planar embedding

Because the collapsed Seifert graph exists of only one node the graph is already planar embedded.

Step five: create the knot layout

Figure B.2: The global layout of the knot diagram
Step six: Connecting the Seifert circles with crossings

Figure B.3: The final layout of the knot generated from $\alpha$
Appendix C

Creating a New Plugin

When some new functionality is desired a new plugin can be written to accommodate this desire. Creating a new plugin means creating a new package consisting of at least an implementation of KnotPlugin and PluginGuiPanel. The interfaces KnotRenderer and KnotMouseHandler can be implemented if needed, but using an already written KnotRenderer or KnotMouseHandler is allowed.

The KnotPlugin class is the heart of a plugin. It should contain the core functionality of the plugin. When writing a KnotPlugin the following methods need to be implemented:

- public KnotMouseHandler getMouseHandler()
- public void setKnotMouseHandler(KnotMouseHandler mouseHandler)
- public KnotRenderer getKnotRenderer()
- public void setKnotRenderer(KnotRenderer renderer)
- public String getVersion()
- public String getName()
- public JComponent getGUI()
- public void onActivate()
- public void onDeActivate()
- public void onLoad()
- public void onUnLoad()
- public void update()
- public String getTabName()

These methods allow all plugin functionality, further details and descriptions can be found in the JavaDoc, delivered with the software.

To allow the plugin to be loaded in the PluginPane, a PluginGuiPanel needs to be constructed and returned by the KnotPlugin. And PluginGuiPanel needs to inherit from JComponent in order to be loaded in the PluginPane. On the gui panel all user interface functionality can be placed so that the plugin functionality becomes available to the user. PluginGuiPanel has only one mandatory method that needs to be implemented:
public KnotPlugin getParentPlugin()

which returns the KnotPlugin which uses this PluginGuiPanel.

The two optional interfaces are KnotRenderer and KnotMouseHandler. New implementations for these can be created but it is also possible to reuse an existing implementation of an already implemented plugin. A KnotRenderer should implement the following methods:

- public void paint(Graphics g)
- public String getVersion();
- public String getName();

The key method here is paint which allows a visualization on a Graphics object. This method is called by the DisplayPanel and executed on its Graphics panel. This realizes the visualization of knots on the DisplayPane.

A KnotMouseHandler implements the methods (as listed in figure 4.9):

- public String getVersion();
- public String getName();
- public void onSelectPoint(Point2D label);
- public void onSelectLine(Point2D label);
- public void onSelectCrossing(Point2D upperLabel, Point2D underLabel);
- public void onDeselectAll();
- public void onMoveLine(Point2D label, int dx, int dy);
- public void onMovePoint(Point2D label, int dx, int dy);
- public void onAddPoint(Point2D location);
- public void onAddPointAfter(Point2D reference, Point2D location);
- public void onDragLine(Point2D label, int dx, int dy);
- public void onDragPoint(Point2D label, int dx, int dy);
- public void onInvertCrossing(Point2D labelx, Point2D labely);
- public void onDeletePoint(Point2D label);
- public void onCloseComponent(Point2D label);

These methods are called whenever a particular action is performed on the DisplayPanel with the mouse, the KnotMouseHandler of a KnotPlugin can then chose what action to perform in response.
List of Figures

1.1 Two knot diagrams of the two trefoil knots .......................... 6
1.2 A stick diagram of a trefoil knot ....................................... 6
1.3 a) a braid, and b) the knot it represents, From: [vWC06] ............... 7
1.4 a) crossingtype ’+’ b) crossingtype ’-’ .................................. 8
1.5 a) Transformation of the figure eight knot to its Seifert circles, b) Transformation of a knot crossing during the Seifert circle algorithm, From: [vWC06] ............... 8

3.1 a) R1 : a strand is twisted so that a loop is created or removed, b) R2 : two strands are are moved so that they overlap or that an overlap is removed c) R3: a strand that is under (or over) two crossing strands is pulled under (or over) that crossing 12
3.2 Detecting Reidemeister moves in Gauss codes ............................ 14
3.3 Executing an 'R3 shift' on x, y and z ..................................... 14
3.4 Two different Gauss codes representing the same link .................. 15
3.5 One complete Gauss code can lead to different knot diagrams, with different Seifert circle embeddings, of the same knot ....................... 25
3.6 The Seifert graph of α ................................................. 27
3.7 The collapsed Seifert graph of α ......................................... 28
3.8 The planar embedding of the collapsed Seifert graph of α ............... 29
3.9 The global layout of the knot ........................................... 30
3.10 The knot diagram of the Gauss code cgc as created by the layout algorithm .... 31

4.1 2-levelled design ...................................................... 32
4.2 A screenshot of how knots appear in KnotWeaver .......................... 33
4.3 a) The global KnotEngine structure b) the detailed view of the Link class with template E ........................................... 34
4.4 A simplified version of the command pattern as used in UndoableLink .... 35
4.5 The user interface with all its components ................................ 36
4.6 Overview ............................................................. 37
Bibliography


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, January 1995.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, January 1995.


