Summer 2021

Knot Theory in Virtual Reality

Donald Lee Price

KNOT THEORY IN VIRTUAL REALITY

A Thesis
Presented to
The Faculty of the Department of Mathematics
Western Kentucky University
Bowling Green, KY

In Partial Fulfillment
Of the Requirements for the Degree
Master of Science

By
Donald Lee Price Jr

August 2021

KNOT THEORY IN VIRTUAL REALITY

Date Recommended July 8, 2021

Dr. Claus Ernst, Director of Thesis

Dr. Uta Ziegler

Dr. Tom Richmond

Associate Provost for Research and Graduate Education

I dedicate this thesis to my 6^{th} grade math teacher, Mr. VanDyke. Through his encouragement and passion, I learned that I wanted to pursue mathematics. His kindness and leadership as a teacher will never be forgotten.

ACKNOWLEDGEMENTS

I would like to personally acknowledge and thank Dr. Rob Scharein. He met with me and my research group throughout this project to help with roadblocks during development of this software and to teach us how to use very useful programs such as KnotPlot and other command line tools used in this project. His patience and kindness are greatly appreciated. I also thank the research group that I was part of while working on this project and, in particular, Nathan Gillispie for conceiving this idea and for helping me with the graphics in the software.

PREFACE

I chose this area of research because problems in Knot Theory particularly interest me. Knot Theory is a field of math that is accessible to most people but also features plenty of very difficult mathematical and programming problems. I am constantly intrigued by the intersection of math and programming and so this field of study has been perfect for me. I enjoy developing software and building things that are of use to others. True to that ethos, the problems pursued by this thesis fulfilled many of the personal goals I have as a mathematician and software developer.

TABLE OF CONTENTS

Introduction and Overview 1

Overview 3

Background 4

Link Representation 10

Link Relaxation 12

Force Laws 12

Force Calculation 13

Force Application 15

Segment Distance Calculations 16

Link Identification 21

PD Code Construction 21

Line Segment Intersection 21

Multiple Crossings on a Segment 24

Crossing Construction 25

Gauss Code to PD Code Conversion 28

Gauss Code for Identification 31

Technical Details 33

Basic Setup 33

Oculus Controls 33

Web API 36

Notable Architecture 36

Bead and Link Component 37

Dependency Injection 40

Future Work 42

KNOT THEORY IN VIRTUAL REALITY

Donald Price August 2021 46 Pages

Directed by: Claus Ernst, Uta Ziegler, and Tom Richmond

Department of Mathematics Western Kentucky University

Throughout the study of Knot Theory, there have been several programmatic solutions to common problems or questions. These solutions have included software to draw knots, software to identify knots, or online databases to look up pre-computed data about knots. We introduce a novel prototype of software used to study knots and links by using Virtual Reality. This software can allow researchers to draw links in 3D, run physics simulations on them, and identify them. This technique has not yet been rigorously explored and we believe it will be of great interest to Knot Theory researchers. The computer code is written in C# and all code has been made publicly available.

CHAPTER 1

INTRODUCTION AND OVERVIEW

Knot Theory is a subfield of topology and has been a very active and interesting field of study for over a hundred years [1, 10, 27]. Knot theorists have long been interested in questions involving the representation of knots and the study of how to differentiate between knots. Researchers have developed several methods to classify knots and links into different families and to rigorously tabulate them. As in all areas of math, a plethora of mathematical and programmatic tools have been developed in order to enrich the study of knots and links [24, 26, 19].

Mathematical tools developed for knots have included several ways to identify and re-produce diagrams of knots such as the PD Code and the Gauss Code, which are explained in Chapter **??**. Other tools have included tables of knots and links that researchers may refer to in order to obtain pre-calculated data about those knots and links [24, 26, 19, 10, 2]. With increases in computing power, tabulations such as those in [10, 2] have become possible. Three of the most notable resources in this field are KnotPlot [23, 24], developed (and maintained) by Rob Scharein, the Knot Atlas [19], and Knotscape [26]. KnotPlot is a stellar piece of software that allows users to study a very wide variety of properties of knots and links and allows for beautiful animations and graphic representations of knots. The Knot Atlas servers as a repository of knowledge pertaining to Knot Theory (more specifically, it contains the values of numerous pre-computed knot and link invariants). It also makes available the KnotTheory package for Wolfram Alpha's [37] Mathematica software suite. Knotscape contains the complete knot table up to and including 16 crossings and can identify any knot by the Dowker-Thistlethwaite Code.

This thesis details the pursuit of a new utility that is aimed at helping researchers more easily and effectively study in Knot Theory. We make use of the Virtual Reality (VR) Headset known as the Oculus [35]. The headset allows the

user to interact with the space around them using hand-held controllers that come with the Oculus. VR has seen plenty of uses in the entertainment industry with video games and the capability to see interesting locations and spaces using the VR environment. For those unfamiliar, a VR headset, such as the Oculus, goes over one's eyes and rests on the head. The user is then made to feel, through ocular deception, that they are in a different environment. The headset will notice changes in the direction that one looks and can even track the position in 3D space of the user. Couple this with controllers, and one can interact with environments not typically available to humans.

We detail software that will allow Knot Theory researchers to draw knots and links in 3D and actually move around the knots and links to see them from different angles. This is a feat which, to our knowledge, has not been accomplished prior to this software. Of course, other software gives us the ability to see these objects as if they were in 3 dimensions, but all other implementations are simply utilizing 2D screens with mathematical transformations to change the perspective on an image as if it were in 3 dimensions. Our software differs in that it allows the user to see, move around, and actually be next to a knot or a link in 3D. Our software also allows users to run physics simulations on different knots and links. Once one has been drawn, the user can apply forces to what they drew to, hopefully, relax their drawing and show a comparatively simpler version of what they drew. Lastly, we also provide limited capability of actually identifying the knot or link that a user draws. The user must redraw every time they would like to study a different knot or link.

We draw heavily on work that has been done in the past. In particular, we model parts of our software using approaches pioneered by KnotPlot [24]. We would also like to personally thank Rob Scharein for not only his willingness to help in this endeavor, but also his kindness and patience with any questions that we had. We

hope that this software will serve to inspire further generations of mathematicians and computer scientists to build software that aids in the study of Knot Theory. All code pertaining to this project is available at `https://github.com/16dprice/oculus_knot_visualizer`. Furthermore, all code was developed in C# using the Unity framework [36].

1.1 OVERVIEW

The rest of this thesis first establishes relevant background for the reader in Chapter 2. Those familiar with the area of Knot Theory should not feel compelled to read the background supplied here, except for perhaps the section on link representation. We then provide a mathematical discussion of the physics simulations used in our software in Chapter 3. We also discuss several programmatic techniques and code usages in that same chapter. Chapter 4 discusses how we construct PD Codes, construct Gauss Codes, and identify knots and links in our software. We provide technical details of our software in Chapter 5 along with some specific code examples that are worth noting. Finally, we briefly discuss future work that can be done on this software in Chapter 6.

CHAPTER 2

BACKGROUND

In this section, we make several definitions and discuss several topics in order to setup a common language to be used later in this thesis. Some of these definitions and topics do not immediately relate to each other, but they are all relevant in order to wholly discuss all the ideas presented in this thesis.

A *polygonal knot* is the embedding of a polygon in 3 space that has a finite number of edges and does not intersect itself. This thesis will discuss knots and links in general, but the definition of polygonal knot suffices for our discussion. A *polygonal link* is a union of 1 or more polygonal knots such that none of the knots intersect each other. If there are n knots present in this way, we say the link has n *components*. To clarify, we use the work "link" to include knots (i.e. 1 component links), however the use of the word "knot" will always mean a 1 component link. We say that two links are equivalent if one link can be deformed into the other link. These deformations are limited; one may stretch, bend, and slide a polygonal arc of the link so long as the arc is never torn. Such a deformation of a link is called an *isotopy*. For the technical definition see [1, 3].

A *link diagram* D consists of a projection of the line segment(s) into the plane. To obtain a diagram in the plane, imagine shining a light onto the link and observing its shadow on the 2D plane. The place where this shadow intersects itself is called a *crossing* and we call D *regular* if these intersections, or crossings, occur between no more than two strands of the string and there are finitely many intersection points. When considering a crossing, the strand that was closer to the light source is called an *overstrand* and the strand that was further away is called an *understrand*. From now on, when a diagram is referenced, it is assumed that a regular diagram is being referenced. For an example of this so-called light shining process see Figure 1.

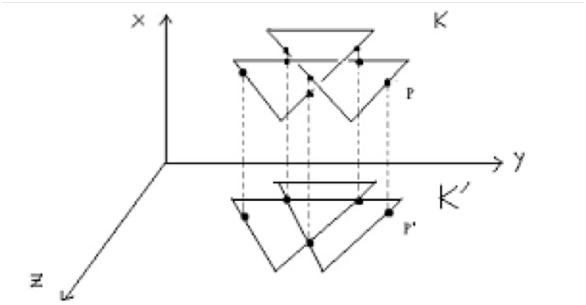

Figure 1: An example of projecting a knot to a plane. [9]

Note that in some of the following figures, we will draw links using smooth curves. However, the reader should keep in mind that if there are enough edges and these edges are short enough, the human eye cannot distinguish between a smooth curve and a polygon. Thus, we need not amend or modify our definition of a polygonal link.

A diagram of a link is considered *minimal* if there are no isotopic deformations that can be made that reduce the number of crossings in the corresponding diagram of the knot. Alternatively, a diagram D of a link L is minimal if there is no other diagram of the same link L with fewer crossings. Note that a link can have multiple distinct minimal diagrams. That is, the same link in 3-space can have more than one distinct minimal diagrams in the plane. We define *link type* (or *knot type*) as the equivalence class generated by a link (or knot) embedding under isotopy. A diagram is called *alternating* if an alternating pattern of over and under strands are encountered at crossings while tracing diagram D in a single direction. Otherwise, a diagram is called *non-alternating* (see Figure 2). A link is called *alternating* if it admits an alternating diagram. Otherwise, it is called *non-alternating*. Furthermore, we say a link is *oriented* if each component in the link has a direction associated with it in which one could traverse the link. We represent this in a link diagram with arrows, as in Figure 3.

Throughout this thesis, we will often refer to knots and links by using

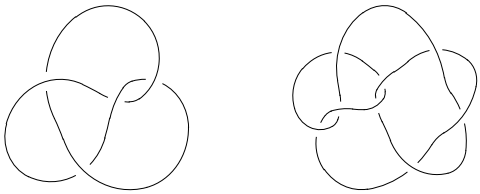

Figure 2: Left: A 4 crossing alternating knot diagram. Right: An 8 crossing non-alternating knot diagram.

Rolfsen notation [20]. There is a canonical ordering of knots and links. Knots are identified by their crossing number sub-scripted with their ordering in the knot table. For example, the only 3 crossing is referred to as 3_1 because it is the first (and only) 3 crossing knot. For links, we use the Thistlethwaite Link Table [21] table. These links are denoted differently. The third alternating 11 crossing link is referred to as $L11a3$ whereas the fifth non-alternating 12 crossing link, for example, is referred is $L12n5$. The L denotes that the object is a link, the number following it is the crossing number, the letter following is either "a" meaning alternating or "n" meaning non-alternating, and the last number denotes the ordering in the link table.

In our software, we analyze and use polygons in 3D to represent links. Since we defined links as a union of of polygons, a link may be represented as sequences of 3D points. These 3D points are what we use to manipulate and study links in our software.

The *Planar Diagram Code* (or *PD Code*, for short) is a numeric representation of a diagram of a link. The PD Code is obtained via numbering strands in a diagram and it is a representation of a diagram up to a re-numbering of those strands. We detail how to construct a PD Code by example next.

Consider Figure 3. This figure is a diagram of the knot 5_2 where the strands are labeled from 1 to 10. This labeling is constructed by beginning at the point drawn in the diagram and then following the arrow drawn beside that point. Once this direction is picked, the knot is also considered to be *oriented*. As the knot is

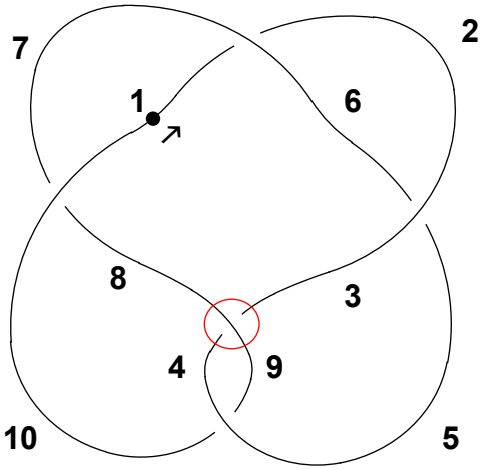

Figure 3: 5_2 with labeled strands

traversed in this direction and with the chosen starting point, each strand is labeled beginning with the number 1. When a crossing is encountered, the number used to label a strand is incremented by 1. Once each strand is labeled in this fashion, an oriented and numbered diagram is obtained. Note that a diagram with n crossings yields numbers 1 through $2n$.

To construct a PD Code from this now oriented and numbered diagram, consider each crossing in the diagram. For Figure 3, there are exactly 5 crossings. Each crossing is represented uniquely by 4 numbers. Consider the crossing that is circled in the figure. The *incoming underpass* is the strand that goes under in the crossing that is also going in to the crossing. For the circled crossing, this strand is labeled by the number 3. We proceed around the crossing in a counter-clockwise fashion to obtain the 4-tuple of $(3, 8, 4, 9)$. This is the ordered tuple of 4 numbers that corresponds to that crossing. Repeating this process with all of the crossings and writing it slightly differently gives a PD Code of
$PD[X[3, 8, 4, 9], X[9, 4, 10, 5], X[7, 10, 8, 1], X[1, 6, 2, 7], X[5, 2, 6, 3]]$ for the knot.

The process for generating a PD Code for a link with 2 or more components is precisely the same except that multiple arbitrary starting points and directions must be chosen for the different components in the link.

Crossings in an oriented diagram have a *sign*. The sign of a crossing is either positive or negative. Figure 4 shows the difference.

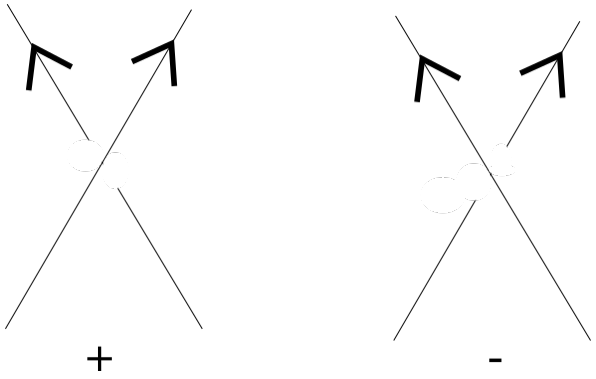

Figure 4: Positive (left) and Negative (right) crossing

The *Gauss Code* [17] is also a numeric representation of a diagram of a link, though it also uses some non-numeric characters. The Gauss Code is obtained in a similar but slightly different way from how the PD Code is obtained. There are several slightly different versions of the Gauss Code. We explain the version that is used in KnotPlot [24]. First, we label crossings in an arbitrary fashion. See Figure 5 for an example of this with a link.

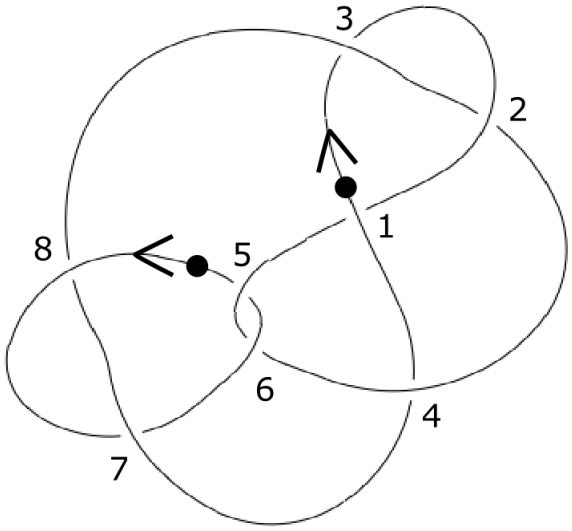

Figure 5: 8 crossing link with labeled crossings

Once the crossings are labeled, we pick an arbitrary starting point. Suppose we start on the black dot in Figure 5 closest to the crossing labeled with a 1. The strategy is then to record information about crossings as we encounter them. We store whether the crossing is over or under as we pass through it in addition to the number that we encounter, and the sign of the crossing. To indicate that we are passing over at a crossing, we record an "a". To record that it is under, we record a "b". For positive crossings, we record a "+". For negative crossings, we record a "-".

Thus, the first thing we record with our selected starting point in Figure 5 is b3-. Continuing this process, we end up with a code of b3-a2-b1+a5-b6-a4+b2-a3-b8+a7+b4+a1+. Now that we have looped back to our starting place, we go to our second component and repeat the process giving us "a8+b7+a6-b5-". We splice these together using the Unix pipe character "|" to obtain a Gauss Code of "b3-a2-b1+a5-b6-a4+b2-a3-b8+a7+b4+a1+|a8+b7+a6-b5-" for the diagram.

The *HOMFLY-PT Polynomial* [7] is an invariant of an oriented link and is calculated via a diagram. That is, the polynomial can be different for different orientations of a knot or link. This polynomial is a two variable Laurent polynomial with integer coefficients. The specifics of how to calculate this polynomial are not relevant for the discussion of this paper. Note that this polynomial is often used to distinguish different knots and links because, with a very high probability, different links have different HOMFLY-PT polynomials. As it is an invariant, if two links have a different polynomial, they are indeed different links. As an example, the HOMFLY-PT polynomial of the link in Figure 5 is given here:

$$H(a,z) = -\frac{1}{a^3 z} + \frac{2}{az} - \frac{2a}{z} + \frac{a^3}{z} - \frac{z}{a^3} + \frac{4z}{a} - 4az + a^3 z - \frac{z^3}{z^3} + \frac{3z^3}{a} - 2az^3 + \frac{z^5}{a}$$

As can be seen from this example, these polynomials are quite complicated and thus it is easy to envision their power in distinguishing different links. However, there are infinitely many distinct links with the same HOMFLY-PT polynomial [11]. A small example of this is the knot 5_1 and the knot 10_{132} in the Rolfsen table [20], which have the same polynomial.

Note that, as mentioned before, some links can have different polynomials when the orientation of one or more of the components is reversed. In the case of a knot, if there are two different polynomials for each of the orientations $H_1(a, z)$ and $H_2(a, z)$ then $H_1(a, z) = H_2(\frac{1}{a}, z)$. For example, for the knot 7_4, we obtain the following two polynomials depending on the orientation of the knot.

$$H_1(a, z) = -\frac{1}{a^8} + \frac{z^2}{a^6} + \frac{2}{a^4} + \frac{2z^2}{a^4} + \frac{z^2}{a^2}$$
$$H_2(a, z) = -a^8 + a^6z^2 + 2a^4 + 2a^4z^2 + a^2z^2$$

In the case of a link with more than one component, there is no easy formula that tells us how the polynomial changes when only one component is re-oriented.

For knots, in particular, when we re-orient it from canonical orientation, we refer to it using an overbar and we call this the *mirror* of a knot. For example, the knot 7_4 has a mirror that is referred to as $\overline{7_4}$. Note that in some cases, knots are actually isotopic to their mirrors. For example, 4_1 and $\overline{4_1}$ are the same knot.

2.1 LINK REPRESENTATION

We discuss here how links are represented in persistent storage and in memory in our software.

To represent links in a way that is accessible to computers, we store a series of 3D points which correspond to the coordinates of the vertices of a link. For "typical" links (i.e. knots and links in the Rolfsen Table [20] or other named links),

we store point data on the hard drive. We used "The Knot Server" [28] to obtain a list of vertices for prime knots up to 8 crossings as well as 2 and 3 component prime links up to 6 crossings. For knots and links that are drawn using the Oculus or that are obtained through manipulation via our software, we store the point data in memory using list structures. The specific structures we use are not relevant to the discussion in this thesis, though the terminology we use makes communicating other topics more straightforward.

Once point data has been loaded into memory from the hard drive or is created in our program, we refer to a point as a *bead*. A bead is simply a point in 3D with a few other properties. Since a bead is part of a link, it has a bead that occurs before it and a bead that occurs after it. These beads are called *adjacent*. Any distinct beads that are not adjacent are referred to as *non-adjacent*. A bead is neither adjacent nor non-adjacent to itself. In certain situations, we add more properties to a bead as needed to perform the tasks we detail in later sections.

CHAPTER 3

LINK RELAXATION

In this section, we describe the necessary algorithmic parts to run a physics simulation on a link. The intended purpose of this is to simplify a knot or link and make it smoother. We do not give a rigorous definition of what it means to be "simpler" or "smoother", though one can imagine a very tangled configuration of a knot. We would say that a "highly tangled" knot is not very simple. In this section, we lay out exactly what physical forces we use and how those forces are used in a discrete manner with a polygonal link. Note that throughout this discussion, there is nothing that necessarily implies that the forces presented are sufficient to simplify a link. The reliance on these methods for simplification is justified through experimental verification. That is, KnotPlot [24] has proven to be very effective over the years at relaxing knots and links. We rely on the sentiment that this software is good at relaxing links to justify our claim that our software is good at this as our implementation of link relaxing is based on the same principles.

Note that when we refer to *adjacent* line segments, we mean line segments that have a bead in common. *Non-adjacent* line segments do not have a bead in common. Similarly, *adjacent* beads form a line segment and *non-adjacent* beads are not on the same line segment.

3.1 FORCE LAWS

The physics simulations use two different kinds of forces. We utilize a "mechanical" force and an "electrical" force. The mechanical force operates on beads that are adjacent to one another and is an attractive force. That is, beads that are adjacent pull each other closer. The electrical force operates on beads that are non-adjacent and is a repelling force. That is, beads that are non-adjacent push each other apart. We apply these forces to each bead one at a time. For each bead, we find the beads that are adjacent and that are non-adjacent. We then construct a

3-dimensional force-vector for each bead corresponding to the direction in which the bead is being pushed.

For the forces used, we follow the same method that is used by knotplot [23]. The mechanical force is given as:

$$F_m = Hr^{1+\beta} \tag{1}$$

where r represents the distance between two beads, H is a constant, and $\beta = 0$ is the standard case for an ideal linear spring. The electrical force is obtained similarly:

$$F_e = Kr^{-(2+\alpha)} \tag{2}$$

where r again represents the distance between two beads, K is a constant, and $\alpha = 0$ represents the typical Coulomb's Law.

3.2 FORCE CALCULATION

We describe how forces are calculated and applied by an example.

Let L be an ordered list of n beads. That is, $L = \{b_0, b_1, \ldots, b_{n-2}, b_{n-1}\}$. To calculate the force acting on a given bead b_i, we first construct two separate lists containing the beads adjacent to b_i and the beads non-adjacent to b_i. If we knew ahead of time that L actually represented a knot, then it would immediately follow that the first and last bead are adjacent. However, this is not the case for a link. To remedy this, we do two things. First, we have a separate array, *componentStartIndices*, that stores the indices of L corresponding to when a new component starts in L. Second, we let each bead b_i store two additional properties. One is called *numBeadsInThisComponent* and the other is called *componentIndex*. *numBeadsInThisComponent* represents the total number of beads in the component that b_i is part of. *componentIndex* represents the index of the component that b_i is

part of. For notational convenience, we will write *componentStartIndices* as C, *numBeadsInThisComponent* for the i^{th} bead as n_i, and *componentIndex* for the i^{th} bead as c_i.

With this information, we can now directly calculate the beads indices that represent the beads adjacent to b_i. For a given b_i, define two offset variables o_1 and o_2 as follows:

$$o_1 = (i - C[c_i] + n_i - 1)\%n_i$$
$$o_2 = (i - C[c_i] + n_i + 1)\%n_i$$

where % is the typical operator used in programming languages to denote the remainder leftover after division and $C[c_i]$ is the c_i^{th} value in C. Then, define the adjacent indices a_1 and a_2 as:

$$a_1 = C[c_i] + o_1$$
$$a_2 = C[c_i] + o_2$$

which will give that b_{a_1} and b_{a_2} are adjacent to b_i. Every bead b_j such that $j \neq a_1, j \neq a_2$, and $j \neq i$ is non-adjacent to b_i.

Now, to calculate the forces on each bead, we have another ordered set $F = \{f_0, f_1, \ldots, f_{n-2}, f_{n-1}\}$ where f_i is a 3D vector and corresponds to the force that acts on the bead b_i. Initially, f_i is $\mathbf{0}$ for all i. We show a force calculation for a given bead b_i next. Note that the calculation is the same, in principle, for every bead.

Given b_i, let b_{a_1} and b_{a_2} be the beads adjacent to b_i and let $r(b_x, b_y)$ be the 3D vector from b_x to b_y. Using equation (1), the total mechanical force exhibited on b_i along with the direction of the force is:

$$F_{m,i} = H * \left(\frac{r(b_i, b_{a_1})}{\|r(b_i, b_{a_1})\|} * \|r(b_i, b_{a_1})\|^{\beta+1} + \frac{r(b_i, b_{a_2})}{\|r(b_i, b_{a_2})\|} * \|r(b_i, b_{a_2})\|^{\beta+1} \right)$$
$$= H * \left(r(b_i, b_{a_1}) * \|r(b_i, b_{a_1})\|^{\beta} + r(b_i, b_{a_2}) * \|r(b_i, b_{a_2})\|^{\beta} \right)$$

Similarly, the electrical force is calculated over all non-adjacent beads.

$$F_{e,i} = K * \sum_{\substack{j=0 \\ j \neq a_1, a_2, i}}^{n-1} \left(\frac{r(b_i, b_j)}{\|r(b_i, b_j)\|} * \|r(b_i, b_j)\|^{-(2+\alpha)} \right)$$
$$= K * \sum_{\substack{j=0 \\ j \neq a_1, a_2, i}}^{n-1} \left(r(b_i, b_j) * \|r(b_i, b_j)\|^{-(3+\alpha)} \right)$$

To calculate f_i, we simply add these two together giving that $f_i = F_{m,i} + F_{e,i}$.

3.3 FORCE APPLICATION

Now that forces can be calculated for a bead, the force must be applied to the bead to make it move in a certain direction. To do this, we can update the position of b_i by adding f_i to it directly. That is, we treat f_i to be a distance and direction for a bead to move. However, a blind addition like this makes it possible to change the knot type. That is, if we make an update by simply adding f_i to the position of b_i for all i, it is possible to have strands between beads pass through each other. Part of the remedy for this is to choose a number d_{max} such that beads are never moved more than d_{max} in a single update. To fully address the issue, we also rely on definitions stated and a theorem proved in [23]. A link-polygon is in a *safe position* if it is not *stuck*. The polygon is stuck if all the beads are *stuck*. For the full definition of these terms, refer to [23]. Though, intuitively, we can imagine that a bead is stuck if it is too close to any other beads or line segments. That is, a bead can become stuck if moving it could risk changing the link type.

The choice of d_{max} relies on our choice of another variable, d_{close}. This parameter is defined as being the smallest distance that two non-adjacent line segments may be from each other while still considering the beads on each end to be in safe positions. That is, if the two line segments from a bead b_i are more than d_{close} units away from all other non-adjacent line segments, then b_i is in a safe position. Otherwise, b_i is not and it is stuck. d_{close} is chosen to be relatively small (i.e. ≈ 0.05) and d_{max} is chosen such that $d_{max} < d_{close}$.

We define a simple force limiting procedure as follows:

$$f_i = \begin{cases} f_i & \|f_i\| \leq d_{max} \\ d_{max} * \frac{f_i}{\|f_i\|} & \|f_i\| > d_{max} \end{cases}$$

Forces are calculated for all beads simultaneously before any movements are made. Once these forces are calculated, we iterate over all beads and apply the corresponding force one at a time. If the application of a force to a bead would put the polygon in an unsafe position, the application of the force is not done. Once this procedure has been completed for all beads, forces are re-calculated with the new bead positions and the same thing is done again.

Thus, we state the following theorem from [23] (which is Theorem 2 in [23]):

Theorem 1. *Using this procedure, the knot type of a knot or link is not changed during force application.*

3.4 SEGMENT DISTANCE CALCULATIONS

We devote an entire section to discussing the process of calculating distances between line segments. This process is repeated many, many times during just one iteration of a link relaxation procedure, thus it must be heavily optimized so as to speed up the physics simulations as much as possible.

For each bead, the distance between the two segments it is attached to and every other segment must be calculated before it can be moved. That is, for each of

the two segments, we must calculate the distance between it and every other non-adjacent segment to check if at least one of those distances is less than d_{close}. If it is, the bead is stuck and cannot be moved. If there are N beads, then there are N line segments. Therefore, we must calculate $N-3$ distances for each line segment (we do not calculate the distance from the segment to itself or to its two neighbors). Since this happens twice for every bead (as each bead is attached to two segments), this results in $2*N*(N-3) = 2N^2 - 6N$ total calculations per one update of bead positions. The time complexity of this is then $O(N^2)$. Of course, this does not take into account the time complexity of actually performing the calculation, but merely shows how many times this calculation must be performed. As algorithms of this time complexity can be notoriously slow, it is imperative to optimize this portion of the physical simulation as much as possible.

To do this, we use the algorithm presented in [25] for the dist3D_Segment_to_Segment function in the source code provided. We discuss only the mathematical formulation (and not the code) given at [4]. The mathematical formulation presented by [4] and translated into optimized code by [25] is the code that we use as it is a more robust implementation of the algorithm. We also discuss what it means to be a more robust implementation and why this is critical to the performance of our software.

Let P_0 and P_1 be the endpoints of the first line segment. Likewise, let Q_0 and Q_1 be the endpoints of the second. We parameterize the line segments given by these points using the formulas $l_1(t) = (1-t)P_0 + tP_1$ and $l_2(s) = (1-s)Q_0 + sQ_1$ for $t, s \in [0,1]$. We define the following variables:

$$a = (P_1 - P_0)\cdot(P_1 - P_0), b = (P_1 - P_0)\cdot(Q_1 - Q_0), c = (Q_1 - Q_0)\cdot(Q_1 - Q_0)$$
$$d = (P_1 - P_0)\cdot(P_0 - Q_0), e = (Q_1 - Q_0)\cdot(P_0 - Q_0), f = (P_0 - Q_0)\cdot(P_0 - Q_0)$$

where the points are treated as vectors for the purpose of subtraction and taking the dot product. Thus, a, b, c, d, e, and f are all real valued numbers. We can then define the squared distance function between the two segments as follows:

$$R(t,s) = \|l_1(t) - l_2(s)\|^2 = at^2 - 2bts + cs^2 + 2dt - 2es + f \tag{3}$$

The details of this formulation are left to the reader.

This gives us the relatively simpler problem of calculating the minimum of a paraboloid constrained by the condition that $(t,s) \in [0,1]^2$.

First, we observe that the cross product $(P_1 - P_0) \times (Q_1 - Q_0)$ is equal to $\mathbf{0}$ if and only if the line segments are parallel. In our software, we assume that line segments are never parallel as the likelihood of this is exceedingly low (basically zero). Recall the following identities from linear algebra given two vectors u and v and the angle θ between them:

$$\|u \times v\|^2 = (\|u\| * \|v\| * |sin(\theta)|)^2$$
$$\frac{u \cdot v}{\|u\| * \|v\|} = cos(\theta)$$

Let $u = P_1 - P_0$ and $v = Q_1 - Q_0$. Let θ be the angle between u and v and observe the following sequence of equivalences:

$$\begin{aligned}
&\|u \times v\|^2 \\
=&(\|u\| * \|v\| * |sin(\theta)|)^2 \\
=&u \cdot u * v \cdot v * (1 - cos(\theta))^2 \\
=&u \cdot u * v \cdot v - cos(\theta)^2 (u \cdot u * v \cdot v)
\end{aligned}$$

$$
\begin{aligned}
&= a * c - cos(\theta)^2 \, (u \cdot u * v \cdot v) \\
&= a * c - (u \cdot u * v \cdot v) * \frac{(u \cdot v)^2}{(u \cdot u * v \cdot v)} \\
&= a * c - (u \cdot v)^2 \\
&= a * c - b^2
\end{aligned}
$$

Now, we have that the line segments are parallel if and only if $a * c - b^2 = 0$. Let $\Delta = a * c - b^2$. As we assume the line segments are never parallel, we assume that $\Delta \neq 0$.

Recall that to minimize a function over a region, we must simply check the boundaries of the region and the inside of the region for minima. As $R(t, s)$ is a paraboloid, it will have exactly one minimum in the unit square. There are 9 total candidates for the location of the minimum of the function in the unit square. The minimum is either at one of the 4 corners of the square, along one of the 4 edges of the square, or inside of the square. The corners of the square are simply the points $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$. We denote the possible minimization points on the boundaries as $(\hat{t_0}, 0)$, $(\hat{t_1}, 1)$, $(0, \hat{s_0})$, and $(1, \hat{s_1})$. We denote the possible inner minimum as $(\bar{t}, \bar{s})$. The task now becomes to calculate the five variables introduced here.

Notice that $\nabla R(t, s) = 2\langle at - bs + d, -bt + cs - e \rangle$. Define $F(t, s) = at - bs + d$ and $G(t, s) = -bt + cs - e$. We can then calculate the unknown variables by solving the following equations:

$$F(\hat{t_0}, 0) = 0, F(\hat{t_1}, 1) = 0$$

$$G(0, \hat{s_0}) = 0, G(1, \hat{s_1}) = 0$$

$$F(\bar{t}, \bar{s}) = G(\bar{t}, \bar{s}) = 0$$

The details of these calculations are very straightforward and so we give the values of these variables once calculated:

$$\hat{t_0} = \frac{-d}{a}, \hat{t_1} = \frac{b-d}{a}$$
$$\hat{s_0} = \frac{e}{c}, \hat{s_1} = \frac{b+e}{c}$$
$$\bar{t} = \frac{be-cd}{\Delta}, \bar{s} = \frac{ae-bd}{\Delta}$$

The task is now to evaluate $R(t,s)$ at all of these points and take the minimum over all the values. Note that not all of the variables calculated here are necessarily between 0 and 1. If they are not, we ignore them as we are only interested in a minimum on or in the unit square. The assumption that $\Delta \neq 0$ is crucial for this calculation. This implementation, as noted in [4], it is quite slow. Furthermore, there are several instances of division of numbers that are possibly very small. These divisions introduce error during floating point arithmetic and therefore a better implementation is desired.

For an optimized version of this procedure, we use an implementation of the code given in [25]. This implementation is optimized for speed and requires a minimal number of operations while also only requiring one division operation. A single division operation is desirable as division with (very small) floating point numbers is notorious for introducing error into calculations.

CHAPTER 4

LINK IDENTIFICATION

In this section, we discuss the process that we use to identify links. We use several concepts discussed in the background to eventually obtain the link type of a link drawn in our software.

4.1 PD CODE CONSTRUCTION

Given a set of beads in our software, it is necessary to calculate a PD Code corresponding to the link represented by those beads. The PD Code that we construct can then be displayed to the user as well as be used to construct a Gauss Code of the link. This Gauss Code is then used with 3^{rd} party software to calculate a HOMFLY polynomial. The resulting polynomial is then used to identify the link.

To construct a PD Code from a set of beads, we assume that we are projecting the link as if we are viewing it from an infinitely long way away in the positive z-direction. This is the classic way to project knots and links. We then treat each bead as if it is part of a strand. In particular, we assume that no beads have the exact same x and y values. Since bead positions are represented using the float primitive in C#, it is, in a practical sense, impossible for this to occur unless points are specifically constructed in such a way to make that happen. Thus, as each bead is part of a strand, we can simply assign a strand number to each bead. A crossing occurs when the strand number changes as we traverse the beads in order.

To detect crossings, we consider the line segments in a given polygonal link. We ignore the z-component and only consider the x and y-components of beads. This way of viewing the beads is analogous to viewing the link from an infinitely long way away in the z-direction. We then find pairs of bead pairs such that the line segments between each of the pairs intersect. The next section discusses determining if two line segments intersect.

4.1.1 LINE SEGMENT INTERSECTION

Given four points, P_0, P_1, Q_0, Q_1, define the line segments l_1 and l_2 such that l_1 connects P_0 and P_1 and l_2 connects Q_0 and Q_1. To determine if these line segments intersect, we construct slope-intercept equations first to find the intersection point of l_1 and l_2 as if they were lines. The slopes m_1 and m_2 of l_1 and l_2, respectively, are as follows:

$$m_1 = \frac{P_{1,y} - P_{0,y}}{P_{1,x} - P_{0,x}}$$
$$m_2 = \frac{Q_{1,y} - Q_{0,y}}{Q_{1,x} - Q_{0,x}}$$

where $P_{1,x}$ and $P_{1,y}$ are the x and y values of P_1. The y-intercepts b_1 and b_2 of l_1 and l_2, respectively, may be calculated as:

$$b_1 = P_{1,y} - m_1 * P_{1,x}$$
$$b_2 = Q_{1,y} - m_2 * Q_{1,x}$$

and we have the following equations of l_1 and l_2:

$$l_1 : y = m_1 * x + b_1$$
$$l_2 : y = m_2 * x + b_2$$

Before proceeding, note that we do not address the case of what happens when l_1 and l_2 are parallel. This would mean that $m_1 = m_2$. As these values will be floating point numbers in C#, it is, practically speaking, impossible for this to occur. The same logic applies as to why we do not consider the two lines being exactly the same line. From here, we proceed by calculating the intersection point

of these two lines and then using parameterized versions of these lines to determine if the line segments intersect. In fact, to do this, we need only the x-value x_I of the intersection point which is given by:

$$x_I = \frac{b_2 - b_1}{m_1 - m_2}$$

We then define parametric forms for the x-values of l_1 and l_2 as follows:

$$l_{1,x}(t) = (P_{1,x} - P_{0,x}) * t + P_{0,x}$$

$$l_{2,x}(s) = (Q_{1,x} - Q_{0,x}) * s + Q_{0,x}$$

Additionally, note that t values of 0 and 1 correspond directly to the points P_0 and P_1. Similarly, s values of 0 and 1 correspond to Q_0 and Q_1.

With these definitions, we may then calculate the corresponding t and s values for the point x_I as follows:

$$t_I = \frac{x_I - P_{0,x}}{P_{1,x} - P_{0,x}} \tag{4}$$

$$s_I = \frac{x_I - Q_{0,x}}{Q_{1,x} - Q_{0,x}} \tag{5}$$

Now, if t_I and s_I are both between 0 and 1, then the line segments represented by P_0, P_1, Q_0, and Q_1 intersect. Thus, we may identify line segments in a polygonal link that intersect. We define these intersections as being our crossings.

Note that this approach has a fundamental flaw. It assumes that for any segment, there is no more than 1 line segment that crosses it. If we assume that it is true that there is at most 1 other line segment that crosses a line segment, this approach works. The case where there is more than 1 line segment does occur and

our solution to this is addressed in the next section. Following that, we discuss how to take these calculations of line segment intersections to produce strand labels and a PD Code for a polygonal link.

4.1.2 MULTIPLE CROSSINGS ON A SEGMENT

Our procedure for determining strand numbers for a polygonal link is to treat each bead as if it were simply part of a strand. With this way of thinking, each bead can be assigned a strand number and then if a crossing occurs over a segment between two beads, the strand number is incremented. If we assume that any line segment has at most 1 crossing that occurs across it, then we could straightforwardly go through the beads of a link and label them with strand numbers and then later extract a PD Code from that labeling. However, it sometimes happens that a line segment has 2 or more crossings occurring across it. Our fix to this is conceptually simple.

Suppose there is a line segment l such that there are k crossings that occur across it. That is, there are k other line segments that cross over l if the z-coordinates were ignored for all segments. Call these line segments $L_c = \{l_1, \ldots, l_k\}$. Our strategy is to simply subdivide l into many smaller segments until each subdivision then only has 1 crossing occurring across it. To accomplish this, we perform a bead add procedure. This procedure computes all the occurrences of a crossing across l and then determines the first occurrence. It then adds a bead between that occurrence and the next.

Let P_0 and P_1 be the endpoints of l similar to before. Define, similar to before, the following parametric equations:

$$l_x(t) = (P_{1,x} - P_{0,x}) * t + P_{0,x}$$
$$l_y(t) = (P_{1,y} - P_{0,y}) * t + P_{0,y}$$

$$l_z(t) = (P_{1,z} - P_{0,z}) * t + P_{0,z}$$

Let t_i be the t-value such that the crossing between l and l_i occurs at t_i determined by the same manner as before. We then have the set $T_I = \{t_1, \ldots, t_k\}$. Suppose, without loss of generality, that $t_i < t_j$ for all $i < j$. Then, the bead that we add to our list of beads is found by defining similar parametric equations for the y and z values of our line segment l. The t-value to determine the position of our new bead is

$$t_b = \frac{t_1 + t_2}{2}$$

We then add a new bead at the position $(l_x(t_b), l_y(t_b), l_z(t_b))$. The procedure is then finished. Notice that this only adds one bead to the link. Thus, this procedure is then repeated until all segments have 1 or fewer crossings occurring over them. As our polygon has finitely many segments, this procedure must stop.

Note that this procedure could be improved upon by subdividing a line segment into as many segments as needed instead of only adding one bead and then repeating the procedure. However, it is a simpler programming task to define a procedure to add one bead and then to repeat this procedure until we are done. Solving the problem in this manner proved to be very quick, so we did not feel the need to optimize this solution.

4.1.3 CROSSING CONSTRUCTION

Once the procedure in the previous section has been performed, we may then assume that all line segments contain at most 1 crossing. We then go through all of the beads and label them with a strand number. As crossings occur between beads, we may treat the beads themselves as being strands of a link diagram. Thus, beads have a strand number associated with them and multiple beads could be associated with a particular strand. We start at whatever bead happens to be first in the list

that the beads are stored in and incrementally label the beads. Once we encounter the first bead again, we move on to any other components and continue this numbering process with the next strand number available.

Next, we go through this list and form crossing pairs of bead pairs. To do this, we iterate through the list of beads and find pairs of beads such that the strand number is different between them. This means that a crossing occurs between the beads. Next, we find the corresponding unique other bead pair that form that crossing with the first bead pair. To do this, we iterate over all other pairs of beads until we find a pair that intersects the current pair. These four beads, or two sets of bead pairs, form a crossing pair.

In each crossing pair, label the 3D points corresponding to these beads as P_0, P_1, Q_0, and Q_1 where one pair has the labels P_0 and P_1 and the other pair has labels Q_0 and Q_1. Note that we assume that P_0 occurs before P_1 with respect to the orientation of the link. Likewise, assume Q_0 occurs before Q_1. To determine the pair which constitutes the understrand of this crossing, we first calculate the values of t_I and s_I in the same manner as was used in equations (4) and (5). We then define parametric equations for the z-components of the lines represented by these points as we have done before:

$$l_{1,z}(t) = (P_{1,z} - P_{0,z}) * t + P_{0,z}$$
$$l_{2,z}(s) = (Q_{1,z} - Q_{0,z}) * s + Q_{0,z}$$

Now, either $l_{1,z}(t_I) < l_{2,z}(s_I)$ or $l_{2,z}(s_I) < l_{1,z}(t_I)$. In the former case, the bead pair corresponding to P_0 and P_1 forms the understrand. In the latter case, the bead pair corresponding to Q_0 and Q_1 forms the understrand. This is because, as noted earlier, we produce a PD Code as if we were looking at the link from an

infinitely long way away in the positive z-direction. With this information, we may then construct a 4-tuple for a crossing as we did in the introductory example of the PD Code.

Let b_{P_0} and b_{P_1} be the beads corresponding to the points P_0 and P_1, respectively. Likewise, let b_{Q_0} and b_{Q_1} be the beads corresponding to the points Q_0 and Q_1, respectively. Without loss of generality, suppose that b_{P_0} and b_{P_1} form the understrand of the crossing. As b_{P_0} occurs before b_{P_1}, we know that b_{P_0} is the incoming understrand. Thus, we may partially construct a crossing code for this. Let s_{P_0} be the strand number of b_{P_0} and s_{P_1} be the strand number of b_{P_1}. The partial crossing code is then $X[s_{P_0}, ?, s_{P_1}, ?]$ where question marks are inserted to indicate that we do not yet know what strand number are in those positions.

The problem now becomes to determine how to place the strand numbers from b_{Q_0} and b_{Q_1}. To do this, we must determine which bead occurs first if we are to traverse all 4 beads in a counter-clockwise manner. Using 2D vector math, this is quite straightforward.

Let u be the projection of the 3D vector $\boldsymbol{P_0P_1}$ onto the x-y plane. That is, we ignore the z-coordinate.. Then, let v and w be the projections of the 3D vectors $\boldsymbol{P_0Q_0}$ and $\boldsymbol{P_0Q_1}$ onto the x-y plane, respectively. We calculate the following angles:

$$\theta_1 = \arccos\left(\frac{u \cdot v}{\|u\| * \|v\|}\right)$$
$$\theta_2 = \arccos\left(\frac{u \cdot w}{\|u\| * \|w\|}\right)$$

θ_1 is the angle between the projections of $\boldsymbol{P_0P_1}$ and $\boldsymbol{P_0Q_0}$. θ_2 is the angle between the projections of $\boldsymbol{P_0P_1}$ and $\boldsymbol{P_0Q_1}$. We know ahead of time that Q_0 and Q_1 occur on either side of the line segment between P_0 and P_1 because otherwise there would not be a crossing here in the first place. Thus, exactly one of θ_1 and θ_2

must be positive and exactly one must be negative. The negative angle corresponds to the point that is immediately counterclockwise of P_0. Thus, if $\theta_1 < 0$, the crossing is $X[s_{P_0}, s_{Q_0}, s_{P_1}, s_{Q_1}]$. Otherwise, it is $X[s_{P_0}, s_{Q_1}, s_{P_1}, s_{Q_0}]$ where s_{Q_0} is the strand number of b_{Q_0} and s_{Q_1} is the strand number of b_{Q_1}. We do this for every crossing pair, and we return the resulting PD Code.

4.2 GAUSS CODE TO PD CODE CONVERSION

As mentioned in the beginning of this section, we use 3^{rd} party software to identify links. The software that we use needs the Gauss Code of a link to identify the link. Thus, given that we can calculate the PD Code of a link, it is then necessary to convert between the PD Code and the Gauss Code. Note that we could simply devise an algorithm to calculate the Gauss Code from the bead representation of a link. However, it is much simpler to design an algorithm to convert a PD Code to a Gauss Code and simply calculate the PD Code from the bead representation since we have already designed that algorithm. As we have done before, we illustrate the algorithm for building a Gauss Code by example.

We consider the first 5 crossing alternating link. This link is listed as 5_1^2 in the Rolfsen table [20]. A PD Code of that link is $PD[X[6,1,7,2], X[10,7,5,8], X[4,5,1,6], X[2,10,3,9], X[8,4,9,3]]$. The first step of our algorithm is to translate the crossings to tuples of 4 numbers and then arbitrarily label them with a number between 1 and n, where n is the number of crossings. For this example, we perform the following translation:

$$X_1 : (6,1,7,2)$$

$$X_2 : (10,7,5,8)$$

$$X_3 : (4,5,1,6)$$

$$X_4 : (2,10,3,9)$$

$$X_5 : (8, 4, 9, 3)$$

We define the function $sign : (s_1, s_2, s_3, s_4) \to \{\text{“}+\text{”}, \text{“}-\text{”}\}$ to find the sign of a crossing as follows:

$$sign(s_1, s_2, s_3, s_4) = \begin{cases} \text{“}-\text{”} & s_4 - s_2 = 1 \; or \; s_2 - s_4 < -1 \\ \text{“}+\text{”} & otherwise \end{cases} \tag{6}$$

Thus, $sign(X_1) = \text{“}-\text{”}$, $sign(X_2) = \text{“}-\text{”}$, $sign(X_3) = \text{“}-\text{”}$, $sign(X_4) = \text{“}+\text{”}$, and $sign(X_5) = \text{“}+\text{”}$. We then translate each crossing into four separate 4-tuples. The first entry of the 4 tuple is one of the strand numbers from the crossing, the second entry is the crossing number, the third entry is an “a” if the strand is an overpass or a “b” if it is an underpass, and the fourth entry is a “+” if the crossing is positive and “-” if it is negative. For example, X_1 would be transformed into the following 4-tuples:

$$(6, 1, \text{“}b\text{”}, \text{“}-\text{”})$$
$$(1, 1, \text{“}a\text{”}, \text{“}-\text{”})$$
$$(7, 1, \text{“}b\text{”}, \text{“}-\text{”})$$
$$(2, 1, \text{“}a\text{”}, \text{“}-\text{”})$$

Once this has been done for every crossing, we will have a list of 4-tuples that is has four times as many elements as there are crossings. We do not write out all the tuples for this particular example, but we do write out key tuples as we go along.

The procedure we perform is to pick a tuple at random. We then record the third, the second, and then the fourth entry of the tuple. For example, for $(6, 1, \text{“}b\text{”}, \text{“}-\text{”})$, we would record “b1-”. This is the beginning of our Gauss Code.

Once we have used this tuple, we delete it from our overall list of tuples. We then find the next tuple by looking for a tuple in our list that has the same crossing number and strand marker (the "a" or the "b") as the tuple we just used. In our example, this would be the tuple $(7, 1, \text{“}b\text{”}, \text{“}-\text{”})$. We delete this tuple as well, but we use the strand number to find the next tuple. As each strand number shows up exactly twice in our tuples, we will find exactly one tuple. In our example, the tuple we find would be $(7, 2, \text{“}a\text{”}, \text{“}-\text{”})$. We append to our Gauss Code the same information we did from our very first tuple giving us a Gauss Code resulting in "b1-a2-".

This procedure will terminate when we look for a tuple that has been deleted. This will occur either because i) we are out of tuples or ii) we have finished the Gauss Code for one component in a multi-component link. If condition i) occurs, we return the Gauss Code and we are done. If condition ii) occurs, we add the pipe character to our list and call the procedure again with our now smaller tuple list. We append whatever that returns to the list we had before. At the end of this procedure, we will have the Gauss Code for the link diagram represented by the PD Code given.

Note that this procedure does not necessarily preserve the orientation of a link. That is, if we had chosen to start with $(7, 1, \text{“}b\text{”}, \text{“}-\text{”})$ in our example above, we would have found $(6, 1, \text{“}b\text{”}, \text{“}-\text{”})$ as our second tuple. This would have resulted in a Gauss Code that has a component whose orientation was opposite of the same component in the PD Code. To be clear, this procedure can result in a Gauss Code that is not the same as the input PD Code as orientations can be reversed. However, we will be using this Gauss Code to pass into another program that will calculate the HOMFLY-PT polynomial. Using this polynomial, we use another program to identify the *unoriented* link type of the polynomial. Thus, we acknowledge that the Gauss Code we produce is not necessarily exactly equivalent

to the PD Code that we input, but we also claim that this does not matter for the purpose that we use it for as identifying the unoriented link type means the orientations of the components of the link do not matter.

4.3 GAUSS CODE FOR IDENTIFICATION

We use a 3^{rd} party program to calculate the HOMFLY-PT polynomial given a Gauss Code. We then pass this polynomial into another 3^{rd} party program that we will refer to as *gidknot* (short for get id of knot or link). It identifies the unoriented link type by doing a dictionary lookup on the polynomial given. We received both of these programs through personal communication with Rob Scharein, though the program for the HOMFLY-PT polynomial generation came from work similar to the work done in [8]. As noted before, the HOMFLY-PT polynomial is an invariant for an oriented link type. That is, no matter the configuration of a link, as long as it is oriented in the same way, it will yield the same polynomial. Additionally, it is a nice property of this polynomial that, for relatively low minimal crossing numbers, the polynomials are all distinct. This means that we can very reliably identify most knots and links using just their HOMFLY-PT polynomial. The gidknot program does have some limitations because it is possible for a very small number of links to have the same polynomial. gidknot can identify all links with 2 or fewer components up to 11 crossings. It works by looking up links in a pre-computed table of polynomials.

As the crossing number increases, the number of distinct links with the same HOMFLY-PT polynomial increases and the identification of links via this polynomial becomes less and less reliable. However, for link types with fewer than 11 crossings, this technique works with very few exceptions. Two notable examples, though, are the knots 5_1 and 10_{132} and the knots $\overline{8_8}$ and 10_{129}. The polynomial for both 5_1 and 10_{132} is

$$H(a,z) = 3a^4 - 2a^6 + 4a^4z^2 - a^6z^2 + a^4z^4$$

The polynomial for both $\overline{8_8}$ and 10_{129} is

$$H(a,z) = 2 - \frac{1}{a^4} + \frac{1}{a^2} - a^2 + 2z^2 - \frac{z^2}{a^4} + \frac{2z^2}{a^2} - a^2z^2 + z^4 + \frac{z^4}{a^2}$$

While this is a potential pitfall of our software, we also expect the user to be aware that the link identified could potentially be different than the one that is drawn with our software. In most cases, this can be remedied by simply running the physics simulation until a near minimal representation of the link is found. Once this has been, if the link you see has a perspective such that there are only 8 crossing points, it would be clear in the example of $\overline{8_8}$ and 10_{129} that the link is actually $\overline{8_8}$.

We have two different versions of each binary file for these programs. One version is compiled for MacOS and the other is compiled for Linux distributions. The Oculus is based on an Android operating system. As such, these programs do not run as we expect on the Oculus. There are several possible solutions to this. The solution that was decided to be the most straightforward and worthwhile is to put these programs on a remote server and require that any users who wish to identify links be connected to the internet with their Oculus. This accomplishes two tasks: it offloads the tasks of running these programs to another machine and it allows development of the application to progress while still maintaining a high degree of automation in the processes that we run in our software. The exact setup of this solution is discussed in the next chapter.

CHAPTER 5

TECHNICAL DETAILS

In this chapter, we give specific details regarding different implementations of code in our software. While we do not go into detail on everything that was written, we give a selection of code and methodologies to better represent the work done on this thesis.

5.1 BASIC SETUP

We use version 2019.4.17f1 of Unity [36] for this project and the Oculus Quest 2 [35]. As the Oculus headset is based on the Android operating system, we use a tool called Android Debug Bridge (adb) [29] to upload builds to the Oculus. This tool allows us to build in Unity and simply install the build onto the Oculus once it is connected to a computer with the adb program installed on it. For details on how to install this program, see [29].

To straightforwardly add a build to the Oculus in this manner, the Oculus must be set to Developer Mode. For information on how to do this, see [30]. To do make a build in Unity, one must first have the code for the project. As mentioned in the introduction, the code is available at `https://github.com/16dprice/oculus_knot_visualizer`. Once Unity the code has been cloned onto a machine and the Unity environment is running, one needs to change a few settings in the Unity environment. For information on how to do this, see [31]. One should then be able to make builds with Unity and upload them to the Oculus using the tools and methods cited.

Alternatively, a build file can be provided upon request.

5.2 OCULUS CONTROLS

The Oculus headset comes with two controllers. Each controller has several inputs. For a detailed picture of these inputs, see [32]. The controls used for our software are the A and B buttons on the right hand controller, both the left and

right hand triggers, the hand grip on the right hand controller, and the joysticks on both sides. The triggers are the lever-like controls on the back of the controllers and the hand grip is the lever-like control where the middle finger typically rests on the controller.

To draw a component, we use the position of the right hand controller. When the user pulls down the right trigger, the application enters drawing mode. The user can then trace out a path in 3-space while beads are dropped at evenly spaced intervals as this occurs. Alternatively, a user may place their hand in a position and pull the trigger once to place beads manually. Once the user has completed the component, pressing the A button will connect the last bead to the first bead and will switch from a bead display to a tube display. In this display mode, the user does not see the beads but instead sees a smooth component (where the relative level of "smoothness" depends on the particular configuration of the beads). Figure 6 shows the final product of drawing the knot 7_3. The points used to draw this knot were obtained from the Knot Server [28].

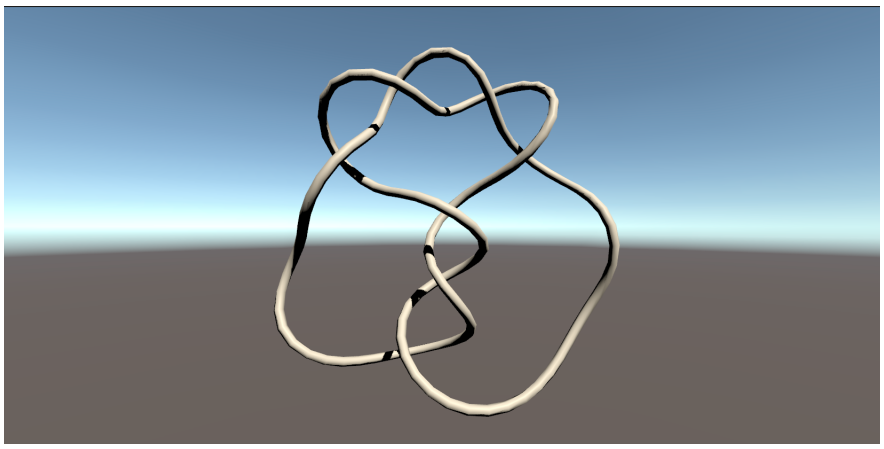

Figure 6: 7_3 as a tube display

The user can then add more components in this same manner. The user can delete all of the components by pulling the left trigger. Pushing down the right hand grip starts the physics simulation process on whatever components have been

drawn and completed. The left joystick controls the relative magnitude of the mechanical and electrical forces. To increase the electrical force, the user pushes the left joystick upwards. To decrease it, the left joystick is pushed downwards. Similarly, to increase the mechanical force, the left joystick is pushed to the right and to decrease it is pushed to the left. The values that these movements control are the K and H constants from equations 2 and 1 in Section **??**.

To identify the link, the user presses the B button once all drawings have been completed. The Oculus then displays the PD Code that it has calculated on a text pane on the right hand side of the user's display. The Gauss Code is displayed below that. Once the API request has returned, the result of that is displayed below the Gauss Code. Thus, even if the identification of the link that has been drawn is inconclusive, the user still has access to a Gauss Code and a PD Code of the link. Figure 7 shows what this is supposed to look like in the Oculus headset. The figure is simply a mock up of what the panel looks like in the headset because we cannot take a screenshot of the view while the headset is on. The same layout is shown while using the software.

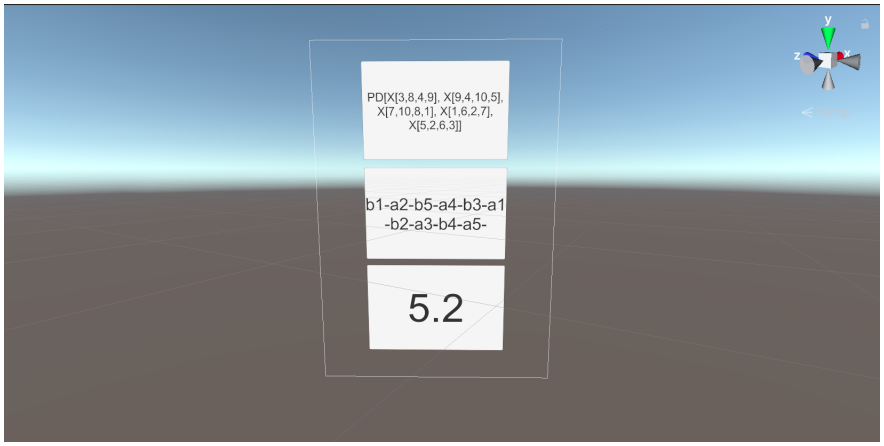

Figure 7: PD Code, Gauss Code, and Identification

Lastly, the user may push the right joystick up to move the link away from them or down to pull it back towards them. When drawing links, it is easy enough

to draw the link some distance away from the headset and to inspect. However, after running physics simulations, the link tends to spread around the user's head and thus it is nice push the link back if this occurs.

5.3 WEB API

To setup our remote server, we use a NodeJS [34] backend environment. We utilize the Express middleware [33] to handle API requests to the server. Thus, our software makes HTTP GET requests over the internet. The Gauss Code is calculated on the Oculus itself in the C# code. This Gauss Code is then sent to our server via the internet. The server receives this code and uses the programs we have for HOMFLY-PT polynomial calculation and link identification to identify the link type corresponding to the Gauss Code that was sent. The server then responds with a string indicating which knot or link the Gauss Code represents. This text is received by the Oculus and is then displayed in the application for the user to see. The API endpoint for this is
`http://107.170.2.119:5000/identify?gaussCode=GAUSS` where `GAUSS` is replaced by a Gauss Code. Notice that this endpoint can actually be used completely independently of the Oculus. If one wishes to make a request to this endpoint, one need only to use any software that will make a GET request and then format the URL as shown here. This endpoint will be available indefinitely so that researchers may continue to use it (with the exception of possible downtime for maintenance or updates).

Note that anyone who wishes to use this endpoint should ensure that all "+" symbols in the Gauss Code are replaced by "%2B" as the "+" symbol is a special reserved character in URLs. The string "%2B" is the special code recognized by browsers and API frameworks to mean "+".

5.4 NOTABLE ARCHITECTURE

In the following subsections, we detail a few specific software architecture

decisions that were made along the way of developing this software. While some of the code examples given may be relatively small, there can often be substantial ramifications in software design if even the smallest portions of code are not given a high degree of attention. The author of this thesis would like to highlight, specifically, the book *Clean Code* [18] by Robert Martin (also known, affectionately, as Uncle Bob). This book gives several examples of good software design principles. While not all of the advice is to always be followed precisely, it provides a wealth of knowledge that is of very practical use.

5.4.1 BEAD AND LINK COMPONENT

There are two classes, in particular, that set the fundamental basis for much of the other code in our software. These classes are the Bead and LinkComponent class. The most simplistic way to view a bead is as a point in 3 dimensions. In essence, that is all that it is. With that definition, a component of a link in our software is simply a list of beads. This easily fits with our definition of a polygonal link as we treat the list of beads to be ordered. The polygonal link is then the set of line segments obtained by connecting successive beads and, notably, connecting the last bead in the list to the first. We display the code for the two classes here:

```
public class Bead
{
    public Vector3 position;
    public Bead(Vector3 position)
    {
        this.position = position;
    }
}
public class LinkComponent
```

```
{
    public readonly List<Bead> BeadList;
    public LinkComponent(List<Bead> beadList)
    {
        BeadList = beadList;
    }
}
```

The `Vector3` object is an object specific to the Unity framework and has functionality as one would expect an object mirroring mathematical vectors would have (i.e. ability to add, subtract, take dot product, cross product, etc.). The `List` object is specific to C# itself. It also acts as one would expect (elements can be appended, elements can be accessed with an index, etc.).

Now, notice that the Bead class is essentially just a wrapper for a 3D point. That is, it holds no functionality other than to contain a `Vector3`. The reasoning for this is quite subtle but has widespread ramifications for our software and for our communication among researchers. When one reads code, one would expect that the code reads specifically as to what it does. For example, consider the next two snippets of code (that do the same thing):

```
 int totalPoints = 0;
for(int i = 0; i < points.Count; i++) {
    totalPoints++;
}
 int totalBeads = 0;
for(int beadIndex = 0; beadIndex < BeadList.Count; beadIndex++)
{
    totalBeads++;
}
```

Clearly, this code is very contrived. One should expect that the value of `totalPoints` will be the exact same as `totalBeads` assuming that `points` and `BeadList` have the same length. The thing to notice here, though, is that the first `for` loop, out of context, is substantially harder to understand than the second. Points can represent anything. Beads, however, are a *specific* thing. Most of a programmer's time is spent on reading code. If we can make reading code more efficient, then the entire software development process becomes more efficient. Thus, proper naming of very small parts of code can have very wide ranging consequences.

Thus, the objective by illustrating these very small classes is to make the reader aware that even though these classes are small and do not do much besides store data, they make the entire software development process much more understandable. It is far easier to read code that deals with Beads than it is to read code that deals with points. When we see an instance of a `Bead` in a piece of code, we know exactly what that code is doing. It is doing some sort of process involving the beads of a link. Moreover, when we see an instance of a `LinkComponent`, we know that the code is doing some sort of process involving a component of a polygonal link. These assumptions would be far less straightforward if one only sees instances of `Vector3` or `List<Vector3>`.

One other thing to notice is how we have, throughout this thesis, noted that different algorithms require a bead to take on different properties. For example, in the description of the force calculations and applications, it was necessary for a bead to know which component it was part of, the number of beads in its component, and which bead in the order of beads it was. For the algorithm that calculates the PD Code of a link, it was necessary for a bead to know which strand it belonged to. Notably, the force calculations need not know anything about strands. PD Code calculations need not know anything about component indices, number of beads in a component, or which bead in the order of beads that it is. Therefore, it is also a

nice property of building such simple classes that we can use them and expand upon them in other portions of code without the need to bloat a single class. For the sake of brevity, the code representing beads specific to the force calculation and PD Code calculation processes are not shown. However, the reader should note that, following the construction of the `Bead` class, we are able to use that class to construct other classes of more specific classes that should hold very specific information.

5.4.2 DEPENDENCY INJECTION

In software development, some parts of code will inevitably be dependent on other parts of code. It is imperative, in many cases, to try to make sure that two seemingly unrelated pieces of code are not too tightly coupled. That is, one should be able to change a piece of code, within reason, without breaking another part of the code. One very effective way to accomplish this is by using interfaces and dependency injection.

Dependency injection refers to the method by which one piece of code receives another piece of code that it is dependent on. In our software specifically, we exhibit an example where we utilize an interface in C# to inject a dependency. Consider the following snippets of code:

```
 public interface ILinkBeadsProvider
{
    List<LinkComponent> GetLinkComponents();
}
 public class LinkStickModel
{
    private readonly List<LinkComponent> _componentList;
    public LinkStickModel(ILinkBeadsProvider beadsProvider)
    {
        _componentList = beadsProvider.GetLinkComponents();
```

```
    }
...
 public class DefaultFileBeadsProvider :  ILinkBeadsProvider
{
...
```

There are three things shown here. First, an interface that enforces anything that implements it must have a method called `GetLinkComponents`. Second, a class that has a constructor which expects an object that implements that interface. And, third, a class that implements the interface.

The important thing to notice here is that the class `LinkStickModel` is only expecting something that has a `GetLinkComponents` method. This class is *dependent* on a class that has that method. Furthermore, the `DefaultFileBeadsProvider` class must have that method. By simply reading the declaration of the class, we can tell that it has implemented that interface and thus must have that method in it. The significance of this may be quite subtle.

The `DefaultFileBeadsProvider` class reads the files that were mentioned earlier in this thesis. It reads the files that contain many 3D points corresponding to different knots and links. The specifics of how it reads these files is not important. What is important is that it somehow constructs a list of `LinkComponent` objects that it can return via calling the `GetLinkComponents` method. The `LinkStickModel` uses this method and constructs a stick model of a link that we use to display the link to a user. Notice how these two tasks are related. However, they do two separate things. The significance of breaking down responsibilities in this way and of using dependency injection in this manner is that we can very safely and easily add more classes later that implement that same interface. Upon doing this, we need not modify the `LinkStickModel` class at all in order to handle these new objects. We simply implement the interface and construct the `LinkStickModel`

class as we normally would. Thus, in the future when perhaps other file formats are considered or other ways of generating points for a polygonal link are introduced, all that needs to be done in order to display those links is to build a new class. No code needs to be edited or deleted other than to pass a different object into the constructor of the `LinkStickModel` class wherever it is used.

Note that there is no way to access these files in our software. We used these files as a way to test our code and to ensure that certain features, such as knot identification, worked properly.

CHAPTER 6

FUTURE WORK

The author would like to continue to work on this project and implement more features into the Oculus. As the code is publicly hosted, it is possible for anyone to contribute to development of this software. It is also possible for anyone to use this software or to suggest changes for it.

It would be beneficial to add functionality to do strand passages (also known as band surgeries) and different nullification moves. In general, we are not allowed to cut or tear a link in any way. However, strand passages and nullification moves are procedures that are known as *unknotting moves*. We do not explicitly define what exactly these are, but they are procedures that can be used to simplify links by decreasing the minimal crossing number of a link. Such procedures are widely studied and are of interest to many Knot Theory researchers [12, 13, 14, 15, 5, 6, 16].

We would also like to add functionality to calculate the PD Code from any perspective of the link that is drawn in our software. As it was not necessary for identifying the link, we did not undertake doing this. It would also be of use to be able to calculate the PD Code from the perspective of the headset. Currently, as mentioned in the sections about constructing the PD Code, the PD Code is calculated as if viewing the link from an infinitely long way away. It could

potentially be more useful to calculate the PD Code from the perspective of the user.

One more long term goal is to make this code accessible on other VR headset devices. The Oculus is quite an expensive piece of technology (ranging in the hundreds of dollars) and we do not necessarily expect those who do not have an Oculus to purchase one solely to use our software. It would be nice to run the same software we have developed for the Oculus on other, cheaper alternatives that researchers may even already have access to.

References

[1] Adams, Colin *The Knot Book*, American Mathematical Soc., (2004)

[2] Burton, Benjamin A. (2020), The Next 350 Million Knots, 36th International Symposium on Computational Geometry, 25:1-17, 10.4230/LIPIcs.SoCG.2020.25

[3] Cromwell, Peter *Knots and Links*, Cambridge University Press, (2004).

[4] Eberly, D., `https://www.geometrictools.com/Documentation/DistanceLine3Line3.pdf`, Last Accessed: July 8, 2021

[5] Ernst, C. and Montemayor, A., Nullification of torus knots and links, The Journal of Knot Theory and its Ramifications, 23. 11. (2014) `https://doi.org/10.1142/S0218216514500588`.

[6] Ernst, C. and Montemayor, A., Nullification numbers of knots with up to 10 crossings, The Journal of Knot Theory and its Ramifications, 25. 7. (2016) `https://doi.org/10.1142/S0218216516500371`.

[7] Freyd P, Yetter D, Hoste J, Lickorish W B R, Millett K and Ocneanu A, A new polynomial invariant of knots and links Bull. Am. Math. Soc. 12 (1985) 239–247

[8] Gouesbet, G. and Meunier-Guttin-Cluzel, S., Computer evaluation of Kauffman polynomials by using Gauss codes, with a skein-template algorithm, Applied Mathematics and Computation, 122. (2001) 229-252

[9] Ho Hon Leung. (2019). Knot Theory III. Fundamental concepts of knot theory (continued). `http://pi.math.cornell.edu/~mec/2008-2009/HoHonLeung/page3_knots.htm`

[10] Hoste, Jim, Thistlethwaite, Morwen, and Weeks, Jeff, The first 1,701,936 knots, The Mathematical Intelligencer, 20. 4. (1998) 33-38, doi:10.1007/BF03025227

[11] Kanenobu, T. Infinitely Many Knots with the Same Polynomial. Proc. Amer. Math. Soc. 97, 158-161, 1986

[12] Kanenobu, T., Band surgery on knots and links, The Journal of Knot Theory and its Ramifications, 19. 12. (2010) 1535-1547.

[13] Kanenobu, T., H(2)-Gordian distance of knots, The Journal of Knot Theory and its Ramifications, 20. 6. (2011) 813-835.

[14] Kanenobu, T., Band surgery on knots and links II, The Journal of Knot Theory and its Ramifications, 21. 9. (2012) 1250086.

[15] Kanenobu, T. and Miyazawa, Y., H(2)-unknotting number of a knot, Commun. Math. Res. 25. (2009) 433-460.

[16] Lickorish, W. B. R., Unknotting by adding a twisted band, Bull. Lond. Math. Soc. 18. 6. (1986) 613-615.

[17] Livingston, C. and Moore, A. H., KnotInfo: Table of Knot Invariants, `knotinfo.math.indiana.edu`, Last Accessed: July 8, 2021.

[18] Martin, R. C. (2009). *Clean code: A handbook of agile software craftsmanship.* Upper Saddle River, NJ: Prentice Hall.

[19] Morrison, Scott and Bar-Natan, Dror, `http://katlas.org/wiki/Main_Page`, Last Accessed: July 8, 2021

[20] Morrison, Scott and Bar-Natan, Dror, `http://katlas.org/wiki/The_Rolfsen_Knot_Table`, Last Accessed: July 8, 2021

[21] Morrison, Scott and Bar-Natan, Dror, `http://katlas.org/wiki/The_Thistlethwaite_Link_Table`, Last Accessed: July 8, 2021

[22] Nakagawa, Yoshiyuki, Introduction of Knotscape, Interdisciplinary Information Sciences, Vol. 9, No. 1, (2003) pp. 89–94

[23] Scharein, Rob, Interactive Topological Drawing, `https://knotplot.com/thesis/thesis_letter.pdf`, Last Accessed: July 8, 2021

[24] Scharein, Rob, https://www.knotplot.com/, Last Accessed: July 8, 2021

[25] Sunday, D., http://www.geomalgorithms.com/code.html, Last Accessed: July 8, 2021

[26] Thistlethwaite, M., http://www.math.utk.edu/~morwen/knotscape.html, Last Accessed: July 8, 2021

[27] Thomson, William, On vortex atoms, Proceedings of the Royal Society of Edinburgh, 6:94-105 (1869) doi:10.1017/s0370164600045430

[28] http://www.colab.sfu.ca/KnotPlot/KnotServer/, Last Accessed: July 8, 2021

[29] https://developer.android.com/studio/command-line/adb, Last Accessed: July 8, 2021

[30] https://developer.oculus.com/documentation/native/android/mobile-device-setup/, Last Accessed: July 8, 2021

[31] https://developer.oculus.com/documentation/unity/unity-conf-settings/, Last Accessed: July 8, 2021

[32] https://developer.oculus.com/documentation/unity/unity-ovrinput/, Last Accessed: July 8, 2021

[33] https://expressjs.com/, Last Accessed: July 8, 2021

[34] https://nodejs.org/, Last Accessed: July 8, 2021

[35] https://www.oculus.com/, Last Accessed: July 8, 2021

[36] https://unity.com/, Last Accessed: July 8, 2021

[37] https://www.wolframalpha.com/, Last Accessed: July 8, 2021