AFRL-RY-WP-TR-2022-0032

ALGORITHMS FOR SKEIN MANIPULATION AND AUTOMATION OF SKEIN COMPUTATIONS (Preprint)

Rachel Harris
Decision Sciences Branch
Multi-Domain Sensing Autonomy Division

FEBRUARY 2022
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE February 2022	2. REPORT TYPE Dissertation	3. DATES COVERED START DATE 17 December 2021	END DATE 17 December 2021

4. TITLE AND SUBTITLE
ALGORITHMS FOR SKEIN MANIPULATION AND AUTOMATION OF SKEIN COMPUTATIONS (Preprint)

5a. CONTRACT NUMBER N/A	5b. GRANT NUMBER N/A	5c. PROGRAM ELEMENT NUMBER N/A
5d. PROJECT NUMBER N/A	5e. TASK NUMBER N/A	5f. WORK UNIT NUMBER N/A

6. AUTHOR(S)
Rachel Harris

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Decision Sciences Branch (AFRL/RYAT) Multi-Domain Sensing Autonomy Division Air Force Research Laboratory, Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command, United States Air Force	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory Sensors Directorate Wright-Patterson Air Force Base, OH 45433-7320 Air Force Materiel Command United States Air Force	10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RYAT	11. SPONSOR/MONITOR'SREPORT NUMBER(S) AFRL-RY-WP-TR-2022-0032

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
PAO case number AFRL-2021-4479, Clearance Date 17 December 2021. A dissertation in Mathematics and Statistics submitted to the graduate faculty of Texas Tech University in partial fulfillment of the requirements for the degree of Doctor of Philosophy. The U.S. Government is joint author of this work and has the right to use, modify, reproduce, release, perform, display, or disclose the work. Report contains color.

14. ABSTRACT
Skein manipulations prove to be computationally intensive due to the exponential nature of skein relations. Resolving each crossing in a knot diagram produces 2 new knot diagrams; knot diagrams with over 5 crossings become increasingly difficult to work with. In this work, I introduce a method for automating these computations using algorithms developed to perform computations in the knot complement. This method is developed for all 2-bridge knots, particularly twist knots and (2,2p+1)-torus knots, but can be extended to other families with modification. After showing these algorithms produce the desired result, I demonstrate their implementation in a Python program. This program is used to to compute several known examples, demonstrating how results obtained through several months of work can be can now be obtained in less than 5 minutes. This program will be used to for testing various hypotheses in SU(2) Chern-Simons theory.

15. SUBJECT TERMS
math, mathematics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT Unclassified	b. ABSTRACT Unclassified	C. THIS PAGE Unclassified	SAR	188

19a. NAME OF RESPONSIBLE PERSON Rachel Kinard	19b. PHONE NUMBER *(Include area code)* (937) 713-8313

Algorithms for Skein Manipulation and Automation of Skein Computations

by

Rachel Harris, E.I.T., B.S., B.S., M.S.

A Dissertation

In

Mathematics and Statistics

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for the Degree of

Doctor of Philosophy

Approved

Dr. Razvan Gelca
Chair of Committee

Dr. Dmitri Pavlov
Co-Chair of Committee

Dr. Alastair Hamilton
Co-Chair of Committee

Dr. Mark Sheridan

August 2021

ACKNOWLEDGMENTS

This work encompasses all I have discovered over the past two years, and is dedicated to Dr. Rodica Gelca, my mentor and friend. She believed in me when I did not believe in myself. Starting graduate school, I felt inadequate and overwhelmed; she gave me encouragement and confidence in a critical moment of self-doubt. I can certainly say that, without her, I would have stopped before I had even started. Without her, I certainly would never have found Razvan. Without Razvan, I would not be a mathematician.

To Razvan Gelca, whom I consider a true genius! For his contagious enthusiasm, constant support, and excellent advice. Never have I found someone with a stronger spirit, a greater love for life, or a desire to live life to the absolute fullest. He taught me that life is best spent invested in projects and mental rest is best found outdoors.

To Dr. Dmitri Pavlov, for opening up my world of mathematics! His patience with new mathematicians is unparalleled. Thank you for "not feeding us second rate formalisms" and for refusing to grade our homework until we made corrections. This was the turning point at which I started to find my feet in higher mathematics. Thanks for all the summer seminars and winter seminars, and the love of mathematics!

To Dr. Xinyun Zhu, my mentor, who took me in as an undergraduate, for first teaching me Topology on the chalkboard in the hallway. For giving me the foundation on which to build this dream! To Dr. Essam Ibrahim, for introducing me to research as an undergraduate. Everything you told me about research turned out to be true - it is just as fulfilling as you made it out to be!

To Christi Hoerster, my best friend, my personal assistant, for the support necessary to succeed. Only thanks to her did I have the courage to pursue this dream! Thank you for not letting me drop Dr. Pavlov's Algorithms class! How could I have known it would lead here?

To Emily Or'El Hoerster, the best Lubbock Buddy ever! For her patience, companionship, and support of my graduate studies! To Richard Hoerster, for Shabbos study time, which is the fuel for weekday studies.

To my fiance, Brandon Kinard, for his enduring patience, bedrock stability, and hot chocolate exactly when I needed it!

To Dr. Tom O'Hara, the only instructor to have known me throughout my entire academic journey. I stubbornly professed I would never go to grad school. He laughed and didn't believe me.

I would also like to thank Connie Sanchez, Alma Brannan, Linda Penny, Joseph Severino, Dr. Chris Hiatt, Dr. Paul Feit, Dr. Warren Koepp, Dr. Forrest Flocker, Dr. Ramiro Bravo, Jim McPherson, Dr. Alex Wang, Dr. Lars Christensen, Dr. David Weinberg, Dr. Dan Grady, and Dr. Magdelena Toda.

L'Hashem Elokeinu, Abba sheli v'elohi. I started this degree with the intention to serve - May I complete what I started!

TABLE OF CONTENTS

ABSTRACT

Skein manipulations prove to be computationally intensive due to the exponential nature of skein relations. Resolving each crossing in a knot diagram produces 2 new knot diagrams; knot diagrams with over 5 crossings become increasingly difficult to work with. In this work, I introduce a method for automating these computations using algorithms developed to perform computations in the knot complement. This method is developed for all 2-bridge knots, particularly twist knots and (2,2p+1)-torus knots, but can be extended to other families with modification. After showing these algorithms produce the desired result, I demonstrate their implementation in a Python program. This program is used to to compute several known examples, demonstrating how results obtained through several months of work can be can now be obtained in less than 5 minutes. This program will be used to for testing various hypotheses in $SU(2)$ Chern-Simons theory.

LIST OF FIGURES

CHAPTER 1 **INTRODUCTION**

Skeins arise naturally in the study of knot and link invariants. As such, skeins are linear combinations of embedded (framed) circles, or more generally graphs, in a 3-dimensional manifold which are identified modulo a set of locally defined relations, called "skein relations". Of all skein theories, the most studied is the one that arises in the Chern-Simons theory with gauge group $SU(2)$, and this particular theory is associated with the skeins studied in this paper. We focus on the version of this theory that is based on the Kauffman bracket, which can be easily related to skeins constructed via the quantum group associated to $SU(2)$.

Skein computations prove to be computationally intensive due to their exponential complexity. Try computing the Jones polynomial of a knot with 5 or more crossings, and you will see why automating such computations is so tempting. In this work, I develop an automated method for performing skein computations in a family of manifolds that is sufficiently restrictive so as to make automation possible, but also sufficiently rich so that insights can be gained from these computations. Following the advice of my mentor, Razvan Gelca, I will strive to make this work both readable and engaging. However, given its highly technical nature, and keeping in mind that this is a dissertation, I sacrifice some readability for accuracy and concision.

It is important to stress out that the skein computations themselves are not difficult to perform at any single stage. Given a fixed set of skein relations, one need only make the proper replacement, splitting one term into two, each with a different coefficient. In fact, performing such a computation is not beyond the reach of even a talented high-school student, so why should it be considered so impactful as to constitute my entire dissertation? This lingering question requires no better answer than one particular example, after which you will agree that it is not the computation itself, but rather the method and the computational effort necessary to produce meaningful results that demands the procedure outlined in this paper.

The scale at which we wish to compute the primary motivation for this work. At each stage, the number of terms in the computation is doubled, rapidly increasing the complexity of the computation. The organization required to perform such a computation by hand, carrying terms and signs correctly at each stage, is beyond what can be expected of most students. Additionally, the diagrammatic nature of these computations is particularly demanding on the human brain, as they require mental manipulation of 3-dimensional geometric objects.

It is natural to ask whether it is worthwhile to perform such computations. The answer to this question comes from experience; within skein theory exist surprisingly simple patterns and formulas awaiting discovery. It often happens that a skein computation involving numerous terms simplifies to a relation with very few terms. Further, in the case of trefoil and 3-twist knots, the polynomial results always appear as Chebyshev polynomials, implying a deeper underlying structure is at work. In the performance of such computations, some of this structure is revealed, but further invetigation is crucial to forming conjectures which will lead to further discovery of these structures. Our automation methods promote such discovery, allowing us to generate many examples in a reasonable timeframe.

1.1 Outline

Let me provide an outline. In Chapter 2, I will address background, starting from the birth of the subject from the works of Vaughn Jones and Edward Witten, to the impact of my grand-advisor, Charlie Frohman. Included are the contributions of my own advisor, Razvan Gelca, as well as Razvan's former students Jeremy Sain, Hongwei Wang, and Shamon Almeida. Next, I will introduce my own assignment and the surprising output that came from its fulfillment.

In Chapter 3, I will introduce the unfamiliar reader to the fundamentals of skein modules and skein computations, providing several examples and how the action of the Kauffman Bracket Skein Module of a knot on the Kauffman bracket skein algebra of its boundary produces information about both, and how this so closely resembles the familiar computation of the Kauffman Bracket and Jones Polynomial.

Chapter 4, by far the longest section, will outline all the details regarding a novel method for the automation of skein computations in the genus 2 handlebody as well as in the complement of 2-bridge knots. I prove why such a method produces the desired result in an efficient manner. In short, a multicurve in the complement of a knot is projected to a specified handlebody embedded in the knot complement and then manipulated there based on an index assigned at the beginning of the computation: strands which cross over are moved to lower indices, while strands that cross under are moved to higher indices. The resolution of crossings introduced in this way produces "basis elements" with no more effort than brute force, although the steps are shown to be close to minimum thanks to thoughtful reduction after each stage of resolution. The final result is reduced, like terms are combined, and it is then written in terms of Chebyshev polynomials, which correspond to interpreting the skein components to be colored by the irreducible representations of $SU(2)$.

In Chapter 5, I discuss the algorithms and their convergence. Chapter 6 is devoted to the results obtained by use of the algorithms presented, and their comparison to our previous computations performed by hand. As one might expect, the results are replicated, even corrected in some cases. I conclude with Chapter 7, which outlines future direction, expanding these methods to fit other situations and alternate skein relations, and explaining the necessary modifications of our methods for use on a much larger family of curves.

1.2 Comments

I would also like to note the implementation of two specific writing techniques. First, I will strive to make the preliminary information given in Chapter 3 sufficient to introduce the beginner to the fundamentals of skein computations, much like a section of class notes. I will include as much good reference material as I have found useful in my own study, including some examples. My hope is that this material would serve as a starting point for other graduate students, regardless of their use of my specific contribution. Second, I have included many examples throughout the text as well as a chapter dedicated to the detailed application of this program to several known examples. It is my sincerest request that future papers include more illustrative examples.

Finally, I have included many, many pictures, generated using the vector graphics editor, Xfig. While I typically prefer to draw diagrams by hand, I have been converted by the ease of use Xfig offers, as it produces professional drawings while retaining the character of the author.

CHAPTER 2
BACKGROUND

2.1 History

Skein theory began with John Conway's discovery of a skein relation for the Alexander polynomial, but it really became a fundamental theory after Vaughn Jones defined his knot polynomial in 1984 [10]. Unlike the Alexander polynomial, the Jones polynomial could not be defined or computed by other means than skein relations. Immediately after Jones, Louis Kauffman defined a related knot and link invariant, known as the Kauffman bracket, which is also computed via skein relations [11]. In 1989 Edward Witten [17] gave an intrinsic definition of the Jones polynomial by means of quantum field theory. However, since Witten's constructs lack mathematical rigor, skein relations remain to date the building blocks of the theory of the Jones polynomial.

It is worth pointing out that Witten has build a rich theory of knot and manifold invariants with the tools of a quantum field theory based on the Chern-Simons functional, and that this theory has been developed rigorously by Nikolai Reshetikhin and Vladimir Turaev [16]. The Reshetikhin-Turaev theory can be constructed entirely with the tools of skein theory as it was shown in [12] and [8]. There is a parallel theory constructed by Christian Blanchet, Nathan Habegger, Gregor Masbaum, and Pierre Vogel which makes use of the Kauffman bracket [1] (see also [14]).

Skein relations yield an algebraic topological construction, the skein module of a manifold, which was introduced in its full generality by Jozef Przytycki [15], following some particular constructs of Vladimir Turaev. Ch. Frohman, D. Bullock, and Adam Sikora have noticed that the skein modules of the Kauffman bracket are deformations of the rings of functions on the $SL(2, \mathbb{C})$ character variety of the fundamental group of the corresponding manifold, more precisely of the ring of affine characters of the fundamental group. This lead to a particular interest in the multiplicative structures of the skein algebras of cylinders over surfaces as well as of the skein modules of knot complements. The relationship between skein modules and character varieties has led to a non-commutative generalization of the A-polynomial of a knot, see [4]. The skein computations that arise in this setting are the target of the program developed in this dissertation.

In particluar, we replicate the skein computations in the genus 2 handlebody of J. Przytycki, and in complements of 2-bridge knots as outlined by T.T.Q. Le [13], and, in particular, those in the complement of (2,2p+1)-torus knots of D. Bullock [2].

2.2 Recent Work

In [5], R. Gelca computed the A-ideal and the A-polynomial of the trefoil knot by first understanding the action of the skein algebra of the cylinder over the boundary of the trefoil knot on the skein module of the knot complement. To compute this action, Gelca developed relations for the action on the family of (p, q)-curves inductively by first computing the action on the curves $(1, 0)$ and $(1, -1)$, and then using the product to sum formula to determine the action on $(1, q)$ and (p, q). This action was used to compute a generator of the A-ideal. Using a similar method, Gelca and Sain computed the A-ideal for the figure-eight knot in [7]. In [9], Gelca and Wang performed computations for the 3-twist knot following the work of Gelca and Nagasato [6]. In the dissertation work of A. Almeida, these computations were replicated using a different set of skein relations. The computations necessary to derive these results in even a single example consume several pages, as each particular case requires a separate and lengthy calculation. Sufficient examples are required to test conjectures and recognize the general form of recurrence relations and higher structure.

2.3 A Motivational Example

To motivate our current work, consider a skein in the complement of the trefoil similar to the computation of the $(1, 0)$ curve in [5]. As in [5], we are required to resolve only 3 crossings resulting in 8 skein diagrams. In [5], these skeins were written in terms of basis elements of the Kauffman Bracket Skein Module of the complement of the trefoil. Only some of the skeins produced from the resolution of the existing crossings were basis elements, and the others were cleverly isotoped and realized as linear combinations of skeins which were easier to resolve. The additional computations required for such skeins resulted in a polynomial with over 30 terms, most of which canceled to produce the relation given by Gelca in Lemma 3 of [5]. In a similar computation, Figure 2.1 shows a skein computation in which three crossings are resolved. Although the input skein is not dissimilar from that of the $(1, 0)$-curve in [5], differing only in the way it intersects, the resulting skeins are more complicated than previoulsy obtained.

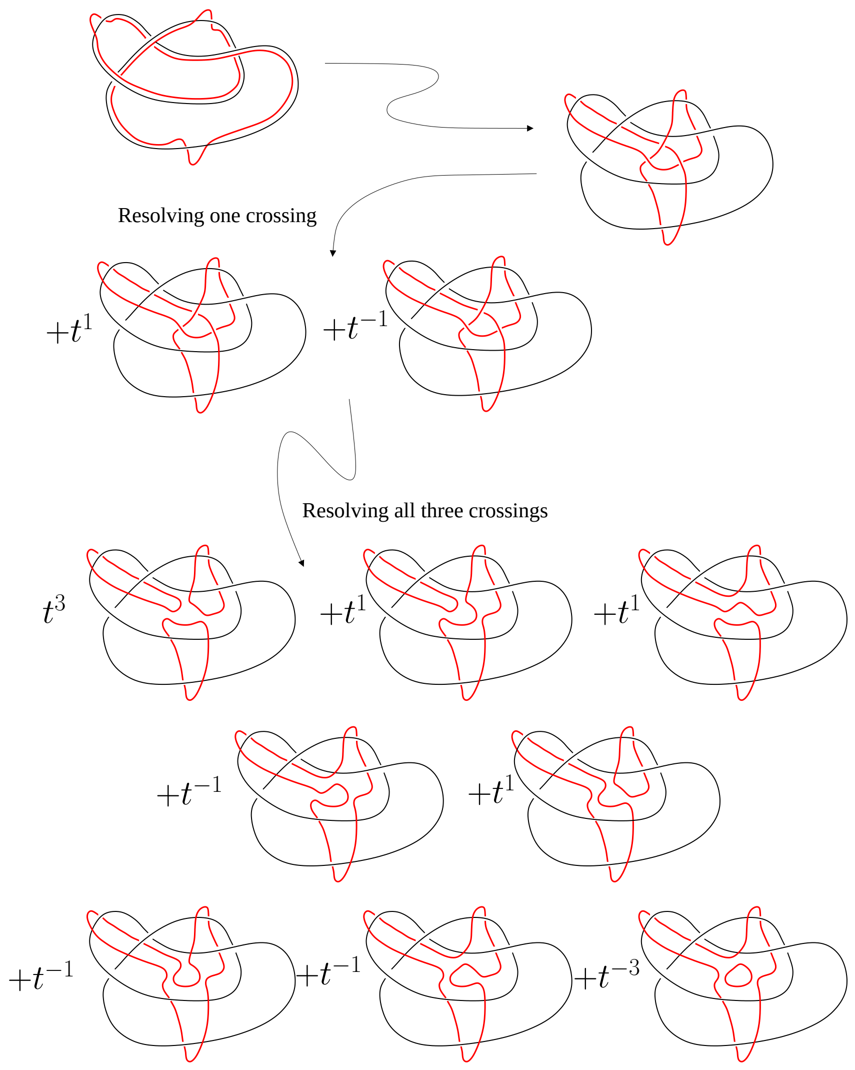


Figure 2.1. A sample computation

At each stage, two skeins are produced, and so the resulting linear combination contains $2^3 = 8$ terms. Some of the resulting skeins can be identified by isotopy, hence combined, and others can be reduced by isotopy. To write the result as the linear combination of a family of desired skeins (such as basis elements of the Kauffman Bracket Skein Module), we must now determine relations for each term individually, necessitating a new skein computation for at least 2 of the 8 terms.

Such a computation, and the additional computations which result from it, is required for every skein we want to resolve. Each skein we resolve is a particular case of a more general result, and a several such examples are necessary to develop or test various conjectures in the skein theory of a particular setting. Similar computations on knots of higher order yield exponentially more terms; performing such computations in the complement of more complicated knots could require months, if not years, invested in producing very few examples. The amount of time and manual effort required to perform skein computations for higher knots exceeds reason.

As my own assignment, rather than spending several years engaged in the lengthy computation of yet another example, I have dedicated my efforts over the past several years to the discovery of a standardized procedure that will suffice for performing these skein computations. The notion that general procedures naturally beget automation, and with assurance from my co-advisor Dmitri Pavlov that it was achievable, I set out to develop algorithms for skein computations. The software which resulted from the development of these algorithms is really only a proof of concept. I felt it important to prove that the algorithms described in this dissertation could replicate previous results in a timely manner. The program, which took the better part of a year to create, took less time to develop than it would have taken to compute the next example. In developing these algorithms, I address two major setbacks in the performance of skein computations. The first is that only experience dictates how a skein is isotoped. If the skein is incorrectly isotoped, the resulting skein may require additional computations on one of the many terms produced. In some instances, skeins isotoped to the wrong location eventually produce the original skein as a term after resolution. The second setback is that, even when handled with expert care, the complexity of such computations is overwhelming when the number of crossings exceeds 5 in the original skein.

Now automation requires two conditions – a generalized procedure and standardized input – and these two conditions happen to perfectly address both of the above setbacks. A generalized procedure, in the form of the algorithms presented in this paper, removes the need for cleverly realizing one skein as a linear combination of other easily resolvable skeins. Additionally, the program, performing all computations with thoughtful reductions at every iteration, handles the overabundance of terms increasing in computational complexity very efficiently. As we have discovered in performing such computations by hand, the final results of such computations are in general shockingly simple, with most computations resulting in mass cancellations at the last step. The program expands these terms and eventually combines and cancels to produce a pleasant output.

CHAPTER 3
PRELIMINARY FACTS

3.1 Knots and Knot Diagrams

A knot is the embedding of a circle S^1 in $\mathbb{R}^3$ or S^3. More generally, a link is a disjoint union of several knots. Knots and links are usually identified up to isotopy. A knot or link can be represented as planar diagrams by taking a projection of the knot or link to the plane, as shown in Figure 3.1. Essentially, this diagram is a shadow of the knot or link, except that it produces apparent self-intersections that must be interpreted. The self-intersections are not intersections at all, but rather represent one strand crossing over or under the other strand in 3-space. If the knot or link is oriented, then we can associate to each crossing a sign, positive (+) or negative (-), using the right-hand rule. For a knot, then signs of the self-crossings do not depend on the orientation of the knot, and from these signs we can deduce if that is an overcrossing or an undercrossing. Just as a knot or link may have several isotopic representations in the 3-dimensional space, a knot or link may also have several equivalent signed planar diagrams. Figure 3.2 shows three diagrams which represent the same link (with one component drawn with black and the other with red).

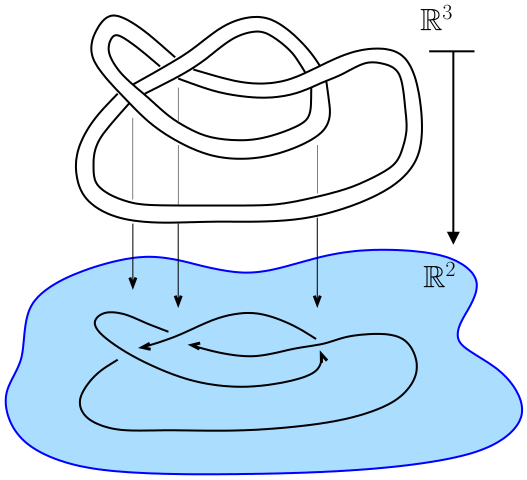


Figure 3.1. Projecting a 3-dimensional knot to the plane, we retain information on the crossings.

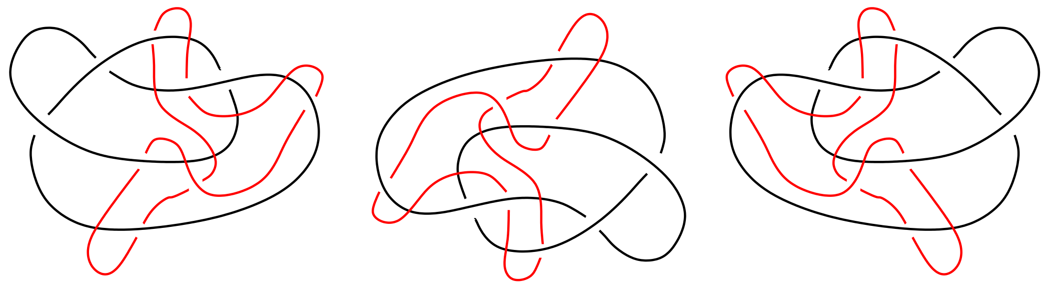


Figure 3.2. Three planar diagrams representing the same link.

A framed knot is the embedding of an annulus. It is customary to represent a framed knot by the same type of diagram as a usual knot, with the convention that the framing is parallel to the plane of the diagram – this is called the blackboard framing. Since the information related to the framing of a knot is important in Chern-Simons theory, all computations involving knots and links must take into account framing. In our diagrams, we draw the knots and links as curves with no apparent thickness, and work in the blackboard framing. This is, of course, only for convenience, although some diagramatic manipulations apply to knots with no framing as well.

3.2 Reidemeister Moves

A theorem of Reidemeister proves that two diagrams represent the same knot or link if and only if they can be transformed into each other by an application of some of the three following moves: The first Reidemeister move (R_1) allows us to twist and untwist the curve in either direction. The second Reidemeister move (R_2) allows two strands to cross or uncross as shown in Figure 3.3. The third Reidemeister move (R_3) allows a strand to pass over or under a crossing in the diagram.

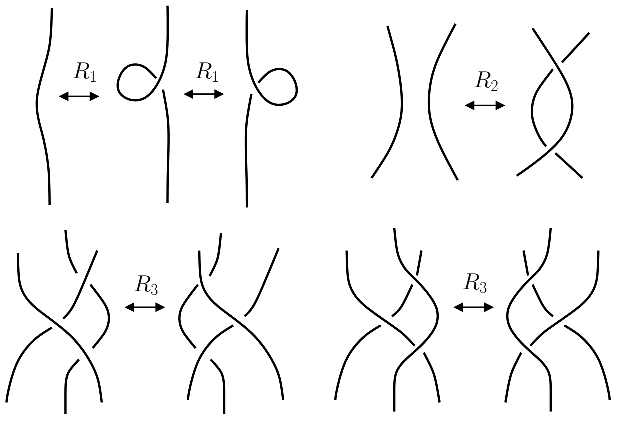


Figure 3.3. The Reidemeister Moves

3.3 Skein Relations and the Kauffman Bracket Skein Module of a Manifold

Let M be a 3-dimensional manifold. Consider the free $\mathbb{C}[t,t^{-1}]$ module $\mathcal{L}(M)$, with basis the isotopy classes of framed knots and links in M, including the empty link. We factor this module by the submodule generated by the elements described in Figure 3.4, where, in each of the linear combinations, the diagrams represents framed links that are identical except in an embedded ball, where they look as shown. The resulting module is called the *Kauffman Bracket Skein Module of M* and is denoted by $K_t(M)$. It is important to point out that since a ball can be embedded in many different ways in a manifold, the resulting module is significantly smaller than the free module spanned by the isotopy classes of knots and links.

The factorization amounts to setting the elements from Figure 3.4 equal to zero. The two relations that you obtain this way, as in Figure 3.4, are called *skein relations*. It is important to point out that adding a positive or negative twist to the framing of one component amounts to multiplying the skein by $-t^3$ and $-t^{-3}$, respectively (a consequence of the two skein relations, shown in Figure 3.5).

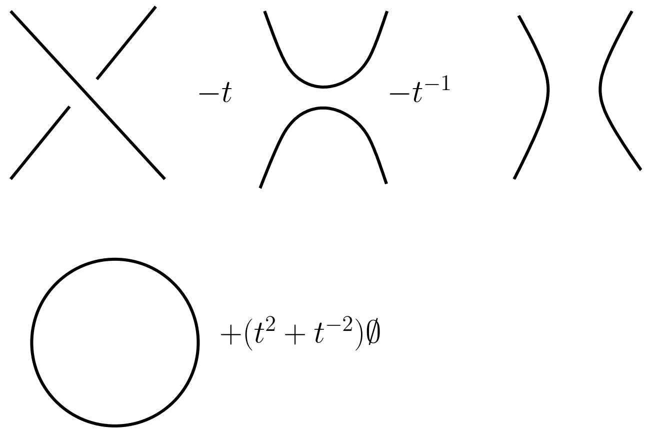
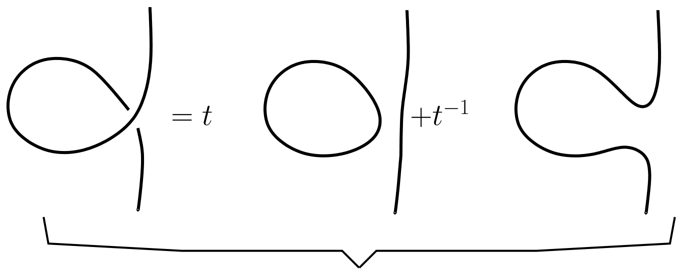
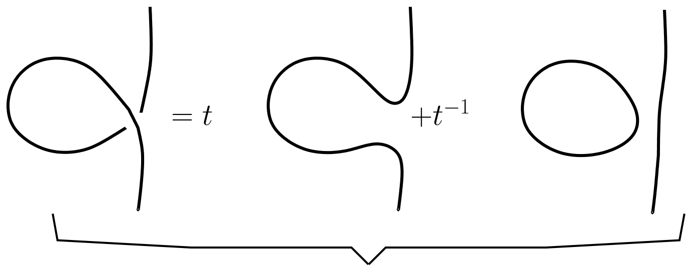


Figure 3.4. The Kauffman Bracket skein relations



$$K_+ = t(-t^2 - t^{-2})K + t^{-1}K = -t^3K$$



$$K_- = tK + t^{-1}(-t^2 - t^{-2})K = -t^{-3}K$$

Figure 3.5. Twisting relations

Two types of manifolds are relevant for this dissertation: genus g handlebodies and complements of knots. The genus g handlebody, H_g, is the cylinder over a disk with g holes, and it is known from the work of J. Przytycki [citation] that $K_t(H_g)$ is a free module with basis all non-selfintersecting multicurves in the disk with g holes. For knot complements the situation is more difficult, the skein module has been shown to be free and a basis has been identified only for the 2-bridge knots [Thang Le] and torus knots [J. Marche] (one should point out also to the early results of D. Bullock on $(2, 2p+1)$ torus knots and twist knots).

When M is the cylinder over a surface, that is $M = F \times [0,1]$ for some surface F, the Kauffman bracket skein module $K_t(F \times [0,1])$ has the structure of an algebra induced by gluing one cylinder on top of another. We denote this algebra shortly by $K_t(F)$, the skein algebra of F. In the case where $F = \mathbb{T}^2$, the 2-dimensional torus, the multiplication in the Kauffman bracket skein algebra has been explicated in [3]. When M is a manifold with boundary, the operation of gluing the cylinder over the boundary to the manifold gives rise to a $K_t(\partial M)$-module structure of $K_t(M)$.

The example that gave rise to the need of developing our algorithm is the one where $M = S^3 \setminus N(K)$, where $N(K)$ is a tubular neighborhood of a knot K in S^3, so where $K_t(M)$ is the Kauffman bracket skein module of a knot complement. Note that it is also a module over the Kauffman bracket skein algebra of the torus $K_t(T^2)$. The study of the $K_t(T^2)$-module structure of $K_t(S^3 \setminus N(K))$ is extremely computationally demanding, and this is the motivation for our work.

3.4 Skein Computations

Given a manifold M and a knot K in M, we can apply a skein relation by embedding a ball in M so that an apparent crossing is created inside this ball. As an example, the framed knot from Figure 3.6 becomes, after the application of the Kauffman bracket skein relation, the linear combination of framed knots in 3.7. So the two represent the same skein in $K_t(M)$.

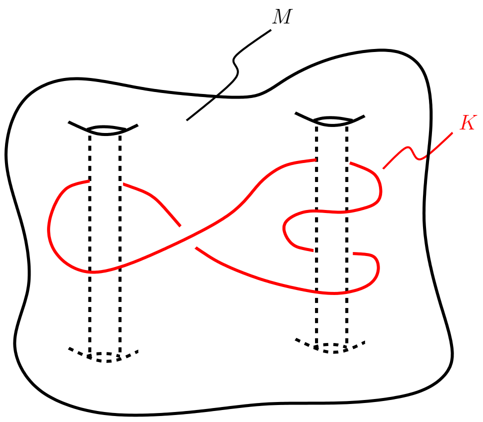


Figure 3.6. A knot in a manifold M

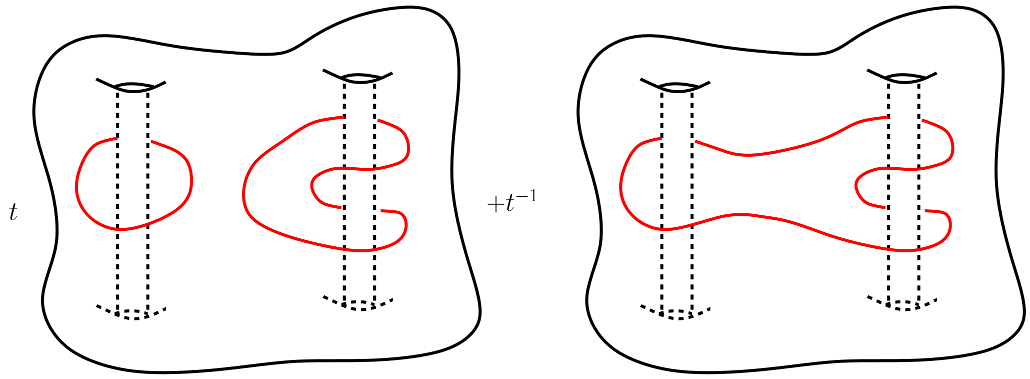


Figure 3.7. Resolving the crossing in M

As an example of interest to us, let B be a fixed knot in S^3 and let $M = S^3 \setminus B$. We will consider skeins in M, such as the one shown in Figure 3.8. You can see that this skein has an apparent crossing, which can be resolved using the Kauffman bracket skein relation.

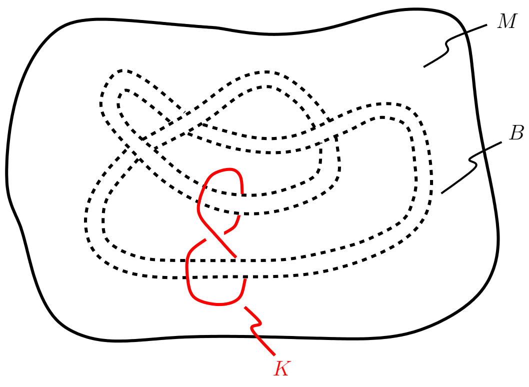


Figure 3.8. Resolving the crossing in M

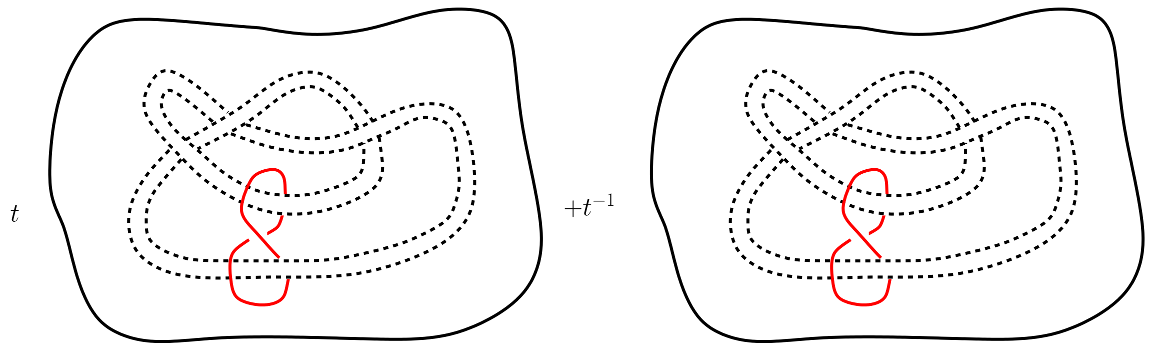


Figure 3.9. Resolving the crossing in M

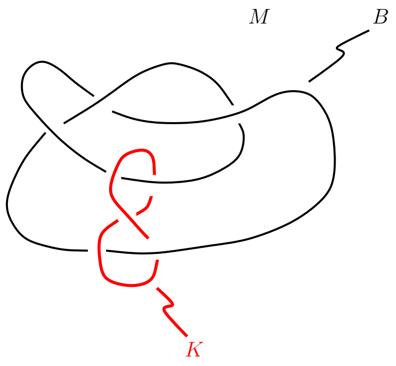


Figure 3.10. Resolving the crossing in M

For convenience, we can view this system as a knot diagram. The complement of the curve B is the manifold M in which our computation is performed, and appears as whitespace in Figure 3.10. The curve K is resolved about the curve B, where B is the vacancy left in $\mathbb{S}^3$ by the removal of the knot B. Throughout this paper, we work in diagrams formatted as in Figure 3.10.

3.5 Basis Elements

We use the power notation to represent parallel copies of framed knots as shown in Figure 3.11. For example, the x^2 indicates two parallel copies of x, while x^3 indicates three parallel copies of x, etc.

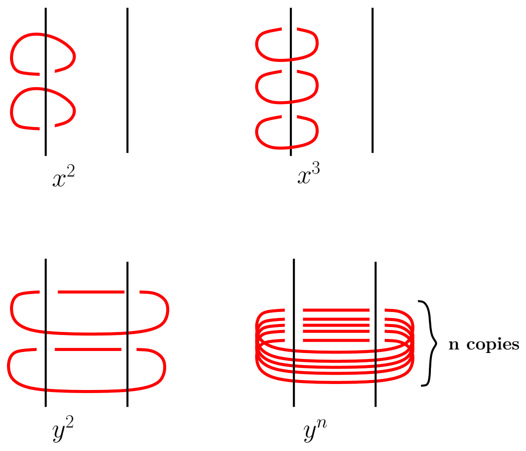


Figure 3.11. The power of a framed knot indicates multiple parallel copies.

Of particular interest to us is the problem of representing a skein in the skein module $K_t(H_2)$ of the genus 2 handlebody as a linear combination of the standard basis elements of this module. In each case, skein computations resolve curves in the handlebody as a linear combinations of the skeins $x^m y^n z^k$, where x, y, z are the framed curves depicted in Figure 3.12 (of course, with the blackboard framing).

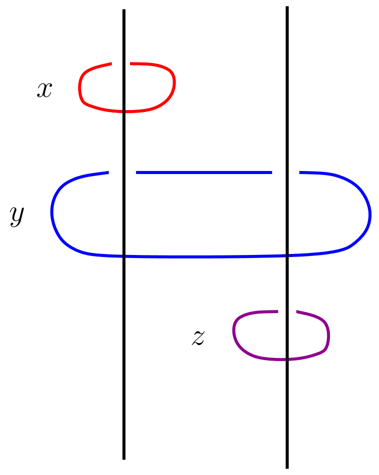


Figure 3.12. Basis elements x, y, and z

For all 2-bridge knots, skeins in the knot complement can be written as a linear combinations of $x^m y^n$, with m any nonnegative integer but with n ranging in a finite set. The curves x, y do indeed come from the basis of an embedded handlebody, and since the knot is one continuous strand, the basis elements x and z of the handlebody are identified.

3.6 Hindrances to Progress

Number of terms produced from crossings

The problem with brute force (creative moves required, like the brain twister curve!)

Ways that computations can go wrong and human errors

The length of time required to perform computations

CHAPTER 4
PROCEDURE AND ALGORITHM

4.1 Computational Setting

Due to constraints determined by our limited understanding of the structure of the Kauffman bracket skein module of the complement of a general knot, we are forced to focus on a family of knots that are sufficiently well understood to be able to perform meaningful computations, and sufficienly large to allow discoveries and conjectures. This family consists of 2-bridge knots.

Thang Le has shown [cite Le here] that the Kauffman bracket skein module of the complement of a 2-bridge knot is free, and he has determined a basis for it. His work has been performed in a genus 2 handlebody, and in fact any skein computation in the knot complement can be performed in this handlebody, and this significantly reduces computational complications. This is because all computations can ultimately be performed in a twice punctured disk.

This is not the case for knot complements in general. While we will show in Lemma 4.1 that curves in the complement of any knot can always be isotoped to lie within a genus g handlebody, listing the basis elements for the skein module of handlebodies of genus $g > 2$ is quite difficult. So results of computations performed in the complement of a handlebody of genus $g > 2$ do not yet produce meaningful formulas. Hence, we content ourselves, for the time being, by modeling only 2-bridge knots.

Thus, our computations take place either in the complement of a 2-bridge knot or in a genus 2 handlebody (the complement of a 2-braid), with some parts of the computations happening in a genus 4 handlebody (seen as the complement of a 4-braid). In the case of a genus 2 handlebody, our goal is to develop an algorithm that would write any skein represented by a linear combination of multicurves as a linear combination of the basis elements of the Kauffman bracket skein module of the handlebody. In the case of a 2-bridge knot, we identify a genus 2 handlebody inside the knot complement, and our goal is to represent any skein that is given as a linear combination of framed multicurves in the knot complement in terms of the basis of the skein module of that handlebody. Combining this with handle slides will allow us to represent any skein in a basis of the skein module of the 2-bridge knot complement.

Every 2-bridge knot can be realized by taking a 4-braid and adding two strands both at the top and at the bottom of the 4-braid. Each of these pairs of strands identifies two of the four strands of the 4-braid. The strands attached at the top form the upper "bridges" of the 2-bridge knot, and the ones at the bottom form the lower "bridges". Alternatively, the knot complement is obtained by adding two 2-handles to the genus 4 handlebody.

4.2 Terminology

Before we proceed, we must establish some conventions and a standardized terminology, that will be used throughout the dissertation.

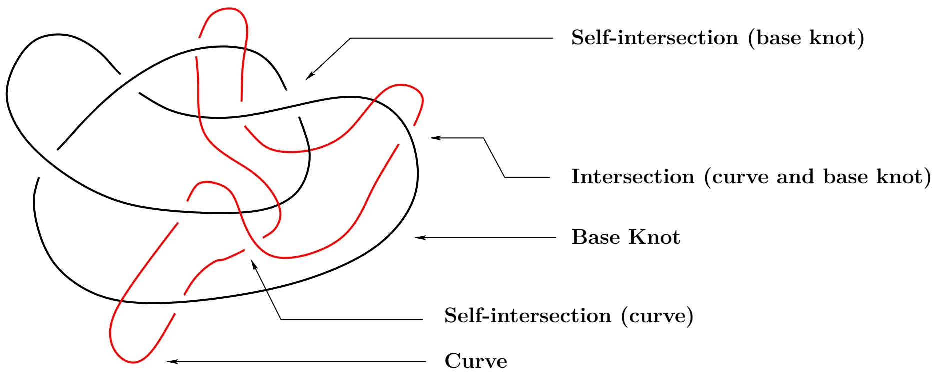

Self-intersection (base knot)
Intersection (curve and base knot)
Base Knot
Self-intersection (curve)
Curve

Figure 4.1. Labeling points of interest in a knot diagram

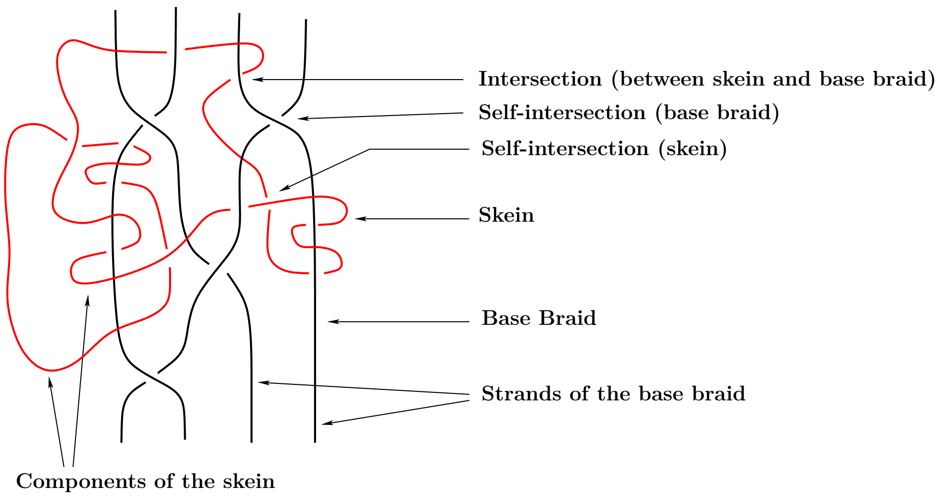

Intersection (between skein and base braid)
Self-intersection (base braid)
Self-intersection (skein)
Skein
Base Braid
Strands of the base braid
Components of the skein

Figure 4.2. Labeling points of interest in a braid diagram

Base Knot and Base Braid

Our computations take place in either a knot complement or in the complement of a braid. These are the **base knot** and **base braid**, and are denoted by B. In diagrams they are drawn with a black line, and are usually held rigid. The crossings of this base knot or braid are not resolved.

Skeins

The **skeins** to be resolved are embedded in the manifold we consider, which is either a knot complement or a handlebody. A skein is an equivalence class, and we represent it by choosing a representative of that class, which in our examples will be a framed link or a linear combination of framed links. In diagrams, the skeins are drawn in red; to them we apply skein relations and isotopies. A skein may have several components, which either intersect or are disjoint.

Figure 4.1 shows a skein in the complement of the trefoil knot. The base knot is therefore the trefoil knot, and the skein is the red curve. We point out that, in 2-dimensional planar diagrams, apparent self-intersections are places where one strand passes either over or under another strand in the 3-dimensional diagram. These intersections are points of interest in such planar diagrams and include places where the curve (skein) intersects the base knot, places where the curve intersects itself, and places where the base knot intersects itself. Figure 4.2 shows a skein in the complement of a 4-braid. The base braid is the 4-braid and the skein is the red curve. In this case the skein has two components which intersect.

Strand

When the base knot, B, is the entire 2-bridge knot, it is made up of one continuous **strand**. A base braid, which can be part of the 2-bridge knot, has multiple **strands**, which can be realized as part of the same knot by recording the connecting information at the top and bottom of the braid. The strands of the braid are indexed sequentially, counting at the bottom of the braid from left to right.

Over/Undercrossing

The terms **overcrossing**, **undercrossing** are applied both to strands of the base knot or braid in the diagram and to curves representing skeins in the manifold. It may be that one strand of the base knot or braid crosses over (or under) another strand, the curve representing the skein crosses over (or under) a strand of the base knot/braid, or this curve crosses over (or under) itself. In all cases, we refer to the crossings using these terms.

Signs of Crossings

The **sign** of a crossing is determined by the right hand rule. A crossing is made up of an overstrand and an understrand; each strand is oriented, and the orientation determines the sign. Imagining the over/undercrossing pair in 3-space, the sign can be determined by the right hand rule is follows: run the thumb of your right hand along the overstrand in the direction of travel and curl your fingers along the understrand as shown in Figure 4.3. If your fingers agree with the direction of the understrand, the sign of the crossing is positive. If your fingers curve in the other direction, the sign of the crossing is negative. This convention is inherited from the convention used in electricity and magnetism to determine current flow in wires and the direction of magnetic fields. This convention is not arbitrary; it was Gauss who produced the first ever link invariant using the Biot-Savart law in electro-magnetism.

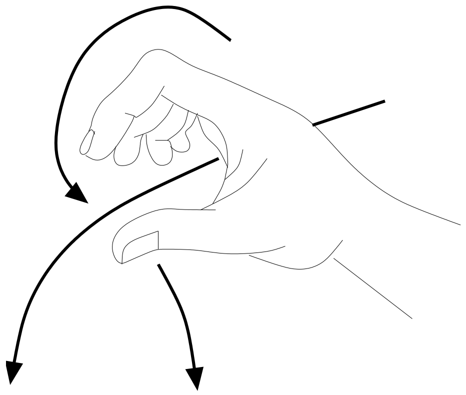


Figure 4.3. Convention for determining if crossing is positive or negative

4.3 Two Special Skeins

When working in the skein module of the genus 2 handlebody, besides the basis elements $x^m y^n z^k$, we introduce two other skeins, y'_{pos} and y'_{neg}, shown in Figure 4.4. We rely on these skeins, writing all other skeins as linear combinations of basis elements and the following two skeins, to reduce computational complexity.

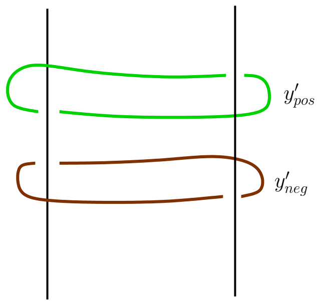


Figure 4.4. Two simple curves, y'_{pos} and y'_{neg}.

After performing the computation, these two skeins can be written in terms of x, y, z as

$$y'_{pos} = -t^{-2}xz - t^{-4}y \text{ and } y'_{neg} = -t^{2}xz - t^{4}y.$$

These relations are obtained by inducing a crossing and resolving it as shown in Figure 4.5. Note that these computations are the same regardless of the orientation of the picture; a y'_{pos} rotated by 180 degrees is still a y'_{pos}. This resolution is independent of other computations made in the knot complement, hence it does not affect the final answer to resolve these curves after resolving the rest of the curve as basis elements. The use of y'_{pos} and y'_{neg} is not of necessity, but of convenience.

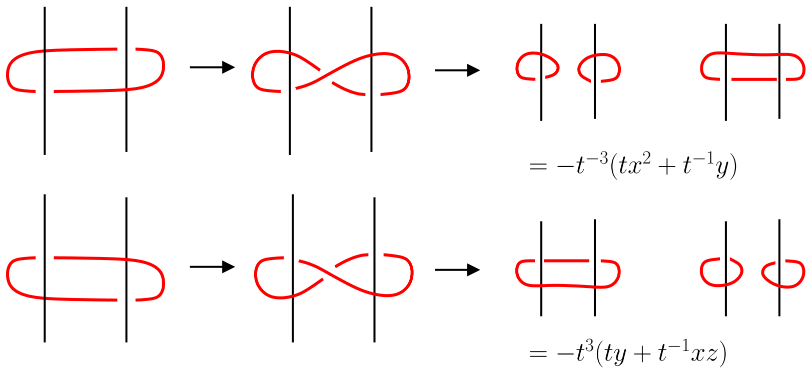


Figure 4.5. The special skeins y'_{pos} and y'_{neg}.

4.4 Conversion from Diagram to Array

The first step to automation is proper input. The difficulty we face is that the input of our computations is diagrammatic. In order to convert from a diagram to an input array, we must record all necessary information, such as intersections between the skeins and base knot/braid, relative position of components of the skeins and strands of the base knot/braid with respect to each other, and the orientation of each component of the skein. Our challenge is that this visual information is non-numerical, and so we establish notation to describe the skein knot/braid system. This notation borrows some of the familiar format of braid representations, but includes more information.

Computations are performed by manipulating skeins in the complement of a knot or braid, sometimes reducing very complicated curves to simple ones by pushing them over and around the strands of the base knot/braid. These simplifications can be difficult to recognize, as often isotoping curves generally makes the situation worse before it gets better.

4.5 Procedure for Performing Skein Computations

Automation will actually take place in the complement of a braid, so if we work in a knot complement we need an algorithm for mapping the skein in the knot complement to a skein in the complement of a braid. The idea is to identify a submanifold N of the knot complement $M = S^3 \backslash N(B)$ such that skeins can be pushed into N and such that $M \backslash N$ is the complement of the braid. This idea is illustrated in Figure 4.6. So from now on we only work with base braids.

Lemma 4.1. *Let $M = S^3 \backslash N(B)$ be the complement of a neighborhood of a knot, B. We can identify a submanifold N of M such that N is a handlebody of genus g, and any skein in M can be isotoped to lie completely inside of N.*

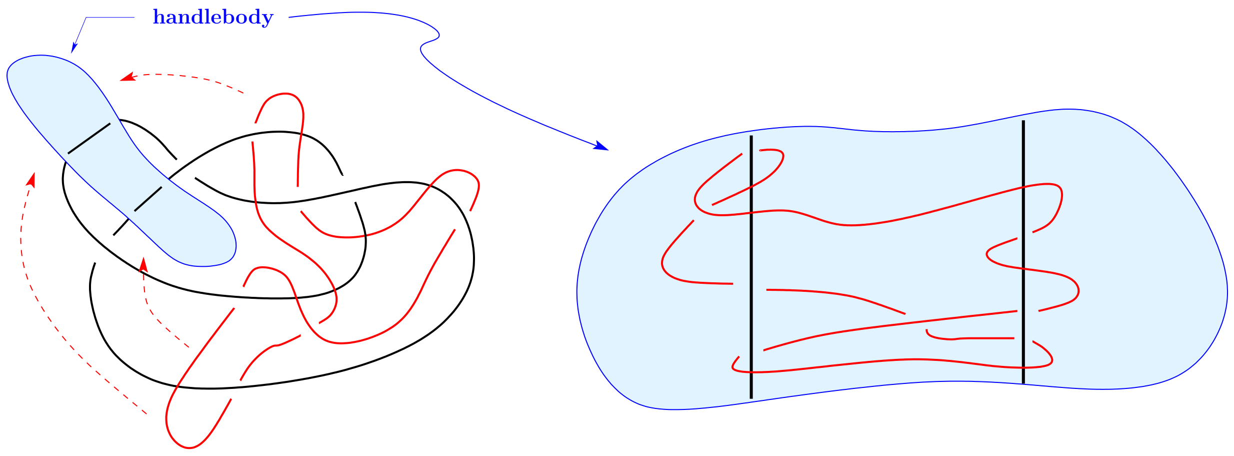


Figure 4.6. Representing a skein in the knot complement as a skein in a handlebody.

Choosing a section of the base knot as described, we record the connecting information of the strands at the boundary of this section. With the section, which corresponds to a handlebody of genus g, oriented so that strands of the braid flow from bottom to top, we label the strands sequentially from left to right. For twist knots and torus knots, there is a preferred location for this handlebody. For twist knots, consider a section of the knot containing 2 strands, one of which is an exterior strand, labeled strand 1 and strand 2. The curve is projected through a handlebody of genus 2 and the computations performed are performed in a handle body of genus 2. For (2,2p+1)-torus knots, the section we choose contains four strands (labeled strand 1, strand 2, strand 3, and strand 4). The curve is projected through a handlebody of genus 4, and the computations are performed in a handlebody of genus 2. We consider the knot as if proceeding from the lower boundary of the section and terminating at the upper boundary of the section. Fixing the strands of the base knot, we now proceed to isotope the curve along the strands, projecting it through the crossings of the base knot, into the portion of the diagram between the highest crossing and the upper boundary of the section. This projection from "bottom" to "top" manipulates the curve through the handlebody of genus g, where g is the number of strands of the base knot in that section.

To accomplish this projection, we must input all information about the base knot: the number of strands, the number of crossings, and the sign of each crossing. The information is always recorded from the lower boundary of the section to the upper boundary, that is, from bottom to top. Next, we input all the information about the multicurve in the complement of the base knot: the strand information (over and undercrossings), the index information, and the orientation information (relative to the position of the strands of the base knot). The procedure for recording this information can be found in Chapter 4.

I will now outline the program for performing computations.

Prepare

The first step is to choose a section of the knot corresponding to a genus g handlebody. We then manually isotope the curve to lie entirely within this handlebody. Viewing the curves within this handlebody as braids, we record the connecting information at the top and bottom of the braid. We will recall this information before proceeding to the next step. Recall that, for ease of entry, there is a left-sided bias to the way a curve is entered, sometimes necessitating an isotopy done by hand so that the curve begins and ends each section on the left side of the base braid. Also, it is occasionally convenient to separate a single connected strand into multiple components. The connecting information at the top and bottom of the section determines the number of actual components in the curve, and will be recalled before the curve is used as the input of the next step.

Step 1: Project

The curve is then projected through the handlebody, from the bottom of the braid to the top, by passing the curve section by section through the crossings of the base knot. The curve is projected to the section above the highest crossing, between that crossing and the upper boundary of the braid; this section is still a handlebody of genus g, but does not contain any crossings in the base braid when viewed as a planar diagram. Also worth noting is that the index and orientation are unnecessary at this step, as the relative position of strands does not change until crossings are resolved. However, the index is used to determine if simplification of over-over and under-under crossings can be made before proceeding. When entries are added before and after the existing strand values, to account for the strand passing through a crossing, their orientations are completely dependent on the orientation of the original value.

Intermediate Step: Reduce

The index information is used to reduce the curve, canceling consecutive over-over and under-under crossing pairs. To reduce computational complexity, we will perform reductions at each stage of the next step; after each crossing is resolved, we check for reductions in the curve and instances of the unknot.

Step 2: Resolve

We now work in a genus 2 handlebody. We resolve all existing crossings, using the skein relations. This results in a linear combination of simple curves with coefficients in powers of the variable t. Since this resolution takes place in a handlebody of genus 2, rather than in a 2-punctured disk, the resulting curves are not guaranteed to be basis elements after all crossings are resolved.

Step 3: Induce

To guarantee the resolution of the skein as basis elements of the skein module, we must project from the genus 2 handlebody to the 2-punctured disk. To do this, we implement an algorithm which rearranges strands by inducing crossings and resolving them. The result of this algorithm is a linear combination of basis elements of the skein module with coefficients in powers of the variable t.

Post-Processing: Combine

Like terms in this linear combination are combined and canceled. In this stage, we also substitute any given relations.

4.6 How to Record Input Information

The procedure above can be divided into two parts: the first half of the procedure (Step 1: Project) involves working with a skein in the complement of a braid (a genus g handlebody) and projecting it to the top section of the base braid, past any crossings. The second half of the Program (Step 2: Resolve and Step 3: Induce) works with a skein in the complement of a genus 2 handlebody whose underlying braid has no self intersections. Our input depends on our starting situation. The rest of the program (Reducing and Combining) formats our output, and can vary depending on the application. If we start in the complement of an n-braid, we begin with Program Project, proceed to Program Resolve, and end with Program Induce. If we begin in a genus 2 handlebody, we begin with Program Resolve and proceed to Program Induce. Our starting point dictates our input.

4.7 Input for Program Project

In Program Project, we work with a skein in the complement of an n-braid, that is, in a genus n handlebody. In the diagram in which we work, the braid B has c crossings (labeled 1 through c) and n strands (labeled 1 through n). The c crossings segment the braid into $c+1$ sections (labeled 1 through $c+1$). The skein K has k components (labeled K^0 through K^{k-1}). For each component K^l, for $0 \leq l \leq k-1$, the component consists of $c+1$ sections (labeled S_0^l through S_{c+1}^l). Each section S_d^l for $0 \leq d \leq c+1$ contains p points of interest, which include intersections between the skein and the base braid, intersections between the skein and itself (self-intersections), and intersections between other components in the skein. We now outline how to record this input information.

4.7.1 Base Braid Input Array

Considering an n-braid, B, with c crossings, we index the strands of the braid, from 1 to c, sequentially from left to right. As with the notation used in conventional braid representations, the labels for strands are exchanged every time two strands cross. For example, when strand 1 and strand 2 cross, strand 1 becomes strand 2 in the section following the crossing and strand 2 becomes strand 1. Hence, braids are naturally sectioned by their crossings. Following this observation, we section the base braid by its crossings. A base braid with 1 crossing will have two sections, one above the crossing and one below the crossing; a base braid with 2 crossings will have 3 sections, etc. as shown in Figure 4.7. Hence we label the sections 1 through $c+1$. The strands of the base braid in each section are labeled sequentially, from left to right, with strand 1 the farthest left strand in each section. We use this labeling scheme to record information about the parts of the skein K that intersect that section.

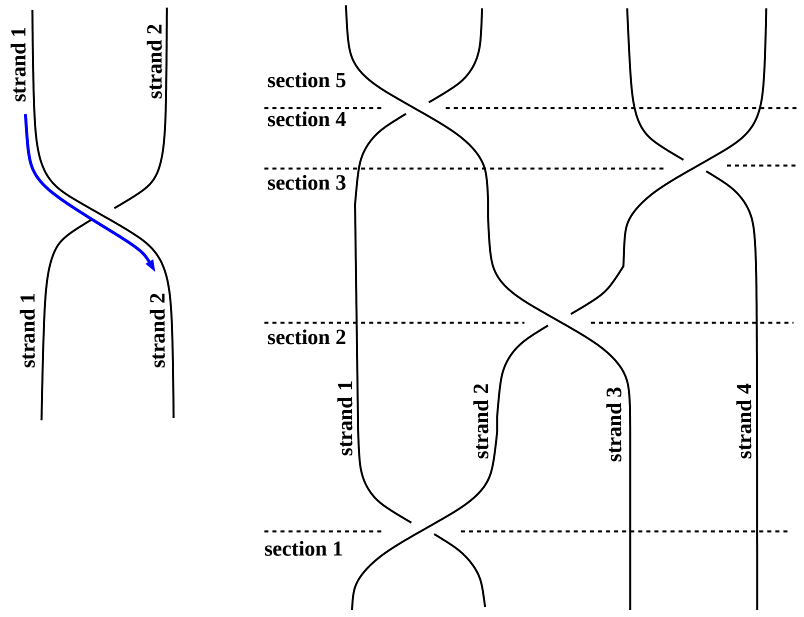


Figure 4.7. Strand labels exchange through a crossing; braids are naturally sectioned by their crossings. Strands are labeled left to right in each section

We record n, the number of strands in the base braid, and proceeding from the bottom of the braid to the top, we record two pieces of information at each of the c crossings in the braid. First, we record which two strands cross in each crossing. Since the consecutive strands will exchange labels, and this labeling depends on which side of the crossing we are on, we need only take the smaller of these two numbers. So, if the crossing is between strands m and $m+1$, we need only record m. Second, we record the sign of the crossing using the following convention: if strand m crosses over strand $m+1$ the sign is positive and if strand m crosses under strand $m+1$ the sign is negative.

Thus, the input array for the base braid is as follows:

n: the number of strands in the base braid, and,

$$\boxed{B = [[b_0, ..., b_{c-1}], [v_0, ..., v_{c-1}]]}$$

where c is the number of crossings in the base knot, labeled 1 through c. For $0 \leq c' \leq c$, the value $b_{c'-1} = m$ where $0 \leq m < n$ and the crossing c' is between strands m and $m+1$. The entry $v_{c'-1} = \pm 1$ is the sign of the crossing c' and corresponds to the entry $b_{c'-1}$. This is the only information required from the base braid, B.

4.7.2 Skein Section Input Array

Now we determine the input array for the skein in the complement of the base braid. The skein is more difficult to describe, as it has no fixed starting point or ending point and moves freely throughout the complement of the base braid. Thus to record the information about arbitrary skeins we require a standardized format and a common point of reference. We bring the curve entirely to the left side of the braid, identify a starting point in each component, and manually isotope the skein, K, to have to form of a braid, beginning and ending at the top and bottom of the base braid. From here, the components of the skein can be recorded.

The base braid is divided into sections based on its crossings, and we record the portion of the skein in each section which intersects the base braid. In most cases, the skein will intersect the base braid in multiple sections, and this is not an issue except that in moving from section to section we require a fixed point of reference to connect the portion of the skeins between sections. Thus, we require that, within each section, the curve starts to the left of strand 1 and comes back to the left of strand 1 before moving to the next section. This may require some manual isotoping, albeit in a very predictable way. This prevents us from having to retain additional information (such as between which two strands the skein connects from section to section) If the curve does not begin or end to the left of strand 1 in the section, we manually pull the curve to the left side, resulting in either additional undercrossings or overcrossings. This is shown in Figure 4.8. Note that in some instances, moving the curve to the left results in an over-over or under-under crossing pair that can be then simplified, as exemplified in the bottom diagram from the figure.

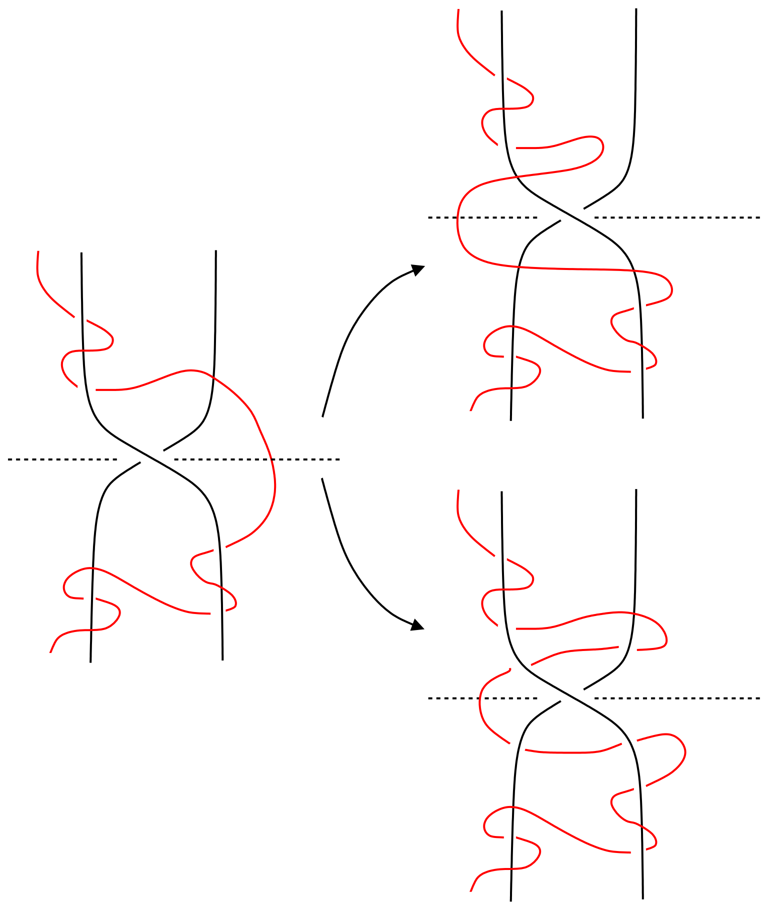


Figure 4.8. The skein is brought to the left side of the diagram; the skein starts and ends to the left of strand 1 in each section.

In most cases, there is a natural way to write the skein so that each component flows from the bottom of the braid to the top, proceeding from a lower section of the base braid to a higher section of the base braid until it reaches the top section in the base braid. If this is not the case, it is generally easy to manually isotope the curve such that this condition is fulfilled. The reason for such a condition is pure convenience; we record the information about the skein from bottom to top, connecting each section to the previous section and recording the input of each section only once. If it is not immediately apparent how to write the skein in such a way, we can begin in the first component at the point where the skein intersects the strand of the base knot closest to the bottom boundary of the base braid. From this point, we follow the skein as long as the section number (the number of the section we consider) increases, hence, we record information about the skein in each section from bottom to top. At the point in which the section number decreases, that is, we visit a section of the base braid which we have already visited, we establish a point R_1. We break the skein at R_1, recording the connecting information by labeling the endpoint of each part of the skein with the same label. This point is brought to the top of the diagram, which connects back to the bottom of the diagram, and we continue recording information from bottom to top. We continue in this fashion for each section of the skein, sometimes splitting a single component into multiple components by breaking the skein at such a point. However, each time we do this, we must retain the connecting information at the top and bottom of the braid to reassemble the skein after projection, so the resulting skein will end up with the same number of components regardless of the number of times a component is split.

In many cases, the skein can be formatted as a braid with fewer intersections by manual isotopy. The skein in Figure 4.9 is already in the desired format. Beginning at the lowest point of intersection between the skein and the braid, we continue until we reach the top of the braid, where we break the skein, bringing the skein to continue as a new component at the bottom.

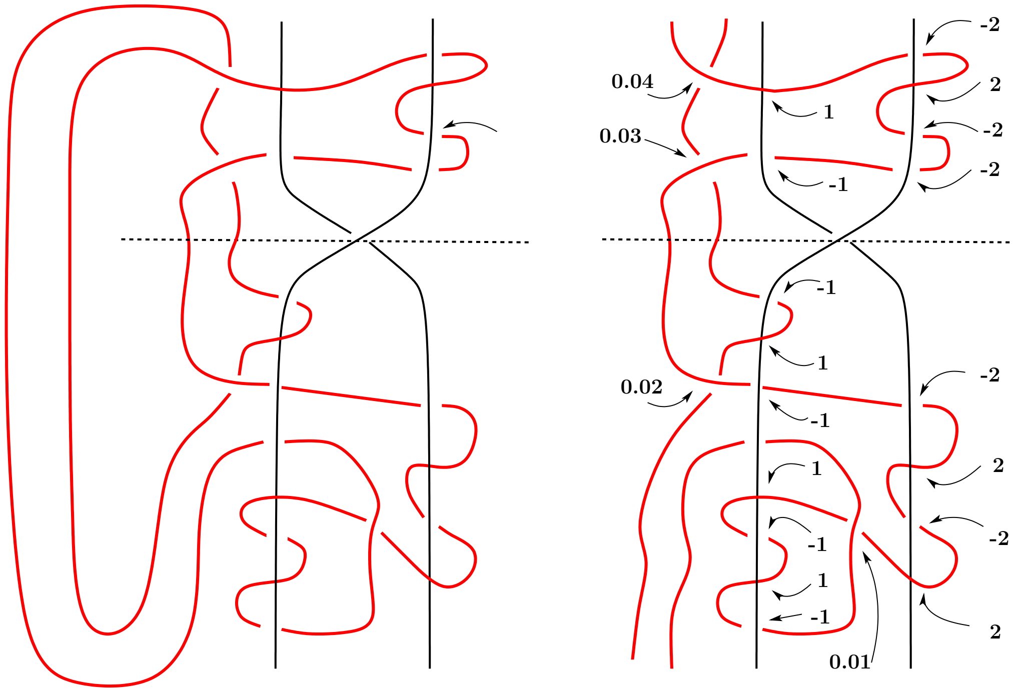


Figure 4.9. The skein is formatted such that the components of the skein start at the bottom of the section and end at the top of the section.

After such a procedure, the skein is formatted such that the components of the skein in each section start at the bottom of the section and end at the top of the section. We record the information for each component of the skein section by section, starting from the bottom section (the section below the lowest crossings). First, we label all crossings, distinguishing self-intersections of the skein and intersections between the skein and the base braid. Self-intersections of the skein (including intersections between one component of the skein and another) are labeled $1\times10^{-k_r}, 2\times10^{-k_r}, ..., r\times10^{-k_r}$, where k_r is the magnitude of the number of crossings, and the labels are assigned to these self-intersections in any order. If there are less than 100 crossings, $k_r = 2$, and so the crossings are labeled $0.01, 0.02, ...0.r$; if there are 100-999 crossings, $k_r = 3$, and the crossings are labeled $0.001, 0.002, ...0.r$, etc. Indeed, the order on these crossings does not matter, as the order only dictates which crossing will be resolved first, second, etc. Since the skein is always pulled to the left side of the diagram, the curve either begins by intersecting the first strand of the base braid, passing through a self-intersection labeled $r' \times 10^{k_r}$ for some $0 \leq r' \leq r$, or else does not intersect the base braid in that section. After this, the curve passes back and forth between strands, crossing over and under strands of the base braid, until it again returns to the left side of the diagram. It may pass through a self-intersection, or multiple self-intersections, before moving to the next section of the base braid.

In each section S^l_d, of component l of the skein, we form an array

$$\boxed{S^l_d = [s^l_{d,0}, s^l_{d,1}, ..., s^l_{d,p-1}]}$$

where p is the number of points of interest in the component, and $s^{p'}_d$, $0 \leq p' < p$, is determined by the following chart:

Table 1: Skein Section Input Information (for a base braid with n strands)
If the curve crosses over strand 1 of the base braid, we record a "1", so $s^k_{d,p'} = 1$.
If the curve crosses under strand 1 of the base braid, we record a "-1", so $s^k_{d,p'} = -1$
If the curve crosses over strand 2 of the base braid, we record a "2", so $s^k_{d,p'} = 2$.
If the curve crosses under strand 2 of the base braid, we record a "-2", so $s^k_{d,p'} = -2$.
...
If the curve crosses over strand n of the base braid, we record a "n", so $s^k_{d,p'} = n$.
If the curve crosses under strand n of the base braid, we record a "$-n$", so $s^k_{d,p'} = -n$.
If the curve crosses over an intersection $0.r'$, we record a "$0.r'$", so $s^k_{d,p'} = 0.r'$.
If the curve crosses under an intersection $0.r'$, we record a "$-0.r'$", so $s^k_{d,p'} = -0.r'$.

When the skein has multiple components (i.e. is represented by a framed link), which may or may not be attached via the connecting information retained at the top and bottom of the diagram, we record the input array for each component. Each component K^l intersects the base braid B in several sections and may intersect other components of the skein in several places. Each component K^l may also intersect itself in several locations. Thus, the input array for the lth component of the skein, section by section is

$$\boxed{K^l = [S^l_0, S^l_1, ..., S^l_c]}$$

where S^l_d is defined above for each section $1 \leq d \leq c+1$ with c the number of crossings in the base braid (thus $c+1$ sections). The input for the first section, S_0^l, is the portion of component l of the skein below crossing 1. The input for the next section, S_1^l, is between crossing 1 and crossing 2. In general, the input for section $d+1$, S_d^l, is the portion of component l between crossing d and $d+1$.

Thus the skein is given by

$$\boxed{K = [K^0, K^1, ..., K^{k-1}]}$$

with K^l defined above for $0 \leq l \leq k-1$ (k components of the skein).

4.8 Program Project Algorithm

From this input, the skein is projected from the bottom of the braid to the top of the braid in the following way: for c crossings in the base braid, separate the curve into $c+1$ sections, and for each component l of the skein, each section S_d^l is moved from section d to $d+1$, starting with section 0 and ending with section $c+1$. At each stage, the result from moving the input of S_d^l from section d to $d+1$ is appended to the input S_{d+1}^{l+1}. The result is the skein projected past all crossings in the base braid to the $c+1$ section of the braid, between the crossing c and the top boundary of the braid.

For each component l, and each section d, consider $S_d^l = [s_{d,0}^l, s_{d,1}^l, ..., s_{d,p-1}^l]$. By Table 1, each entry $s_{d,p'}^l$ for $0 \leq p' \leq p-1$, is either ± 1, ± 2, or t for some $0 < t \leq r$.

We push section S_d^l past crossing d, and append the result to S_{d+1}^l. This is accomplished by the algorithm given in Algorithm 1. Passing this portion of the skein through the crossing d amounts to introducing new elements, overcrossings and undercrossings, at specific locations in S_d^l. The first step in Algorithm 1 is to determine the location of the insertion, and the second is to insert the overcrossings and undercrossings as necessary. The sign of the crossing determines which values are inserted. For each crossing, we pass from the section of the diagram below the crossing to the section of the diagram above the crossing.

The crossing d is made up of two consecutive strands of the base knot, strand m and strand $m+1$. In the section below the crossing, the skein intersects these two strands of the base braid. If we think of the intersections between the skein and the base braid (overcrossings and undercrossings) in groups on each strand, we get a sequence of intersections on strand m followed by a sequence of intersections on strand $m + 1$. This pattern repeats, alternating groupings of intersections on m and $m + 1$, until we get to the location where strand m and strand $m + 1$ cross. To move the curve past this crossing, we can move the groups of intersections on each strand separately, starting with the group closest to the crossing. Intersections between the skein and any strand of the base braid other than m and $m + 1$ are unaffected by the crossing, and are moved to the next section with no changes.

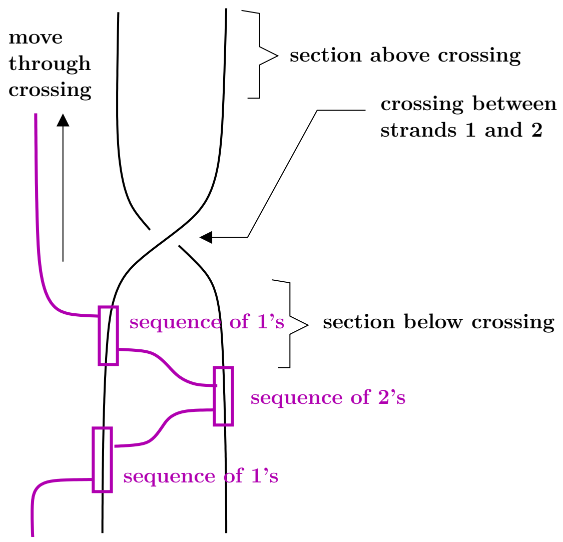


Figure 4.10. moving curve past crossing between strands 1 and 2 in the base knot

Without loss of generality, let the crossing be between strands 1 and 2 of the base braid. In the section below this crossing, we have a grouping of intersections which are over/undercrossings on strand 1 (a sequence of entries ± 1), and a grouping of intersections which are over/undercrossings on strand 2 (a sequence of entries ± 2). When we move groupings on strand 1 through a positive crossing, they move from strand 1 below the crossing to strand 2 above the crossing (as the strand labels switch when the strands are crossed). In doing so, we must introduce an undercrossing on strand 1 before this grouping and after this grouping as shown in Figure 4.11. We insert a -1 before the grouping and we insert a -1 after the grouping. When we move groupings on strand 2 through a positive crossing, they move from strand 2 below the crossing to strand 1 above the crossing. In this case, no additional crossings are added before or after the sequence, as shown in Figure 4.12. In this manner, we move groupings of crossings on strand 1 and groupings of crossings on strand 2, group by group, starting with the group closest to the crossing and proceeding until all parts of the curve appear above the crossing. This result is appended to the beginning of the next section, and the procedure is repeated.

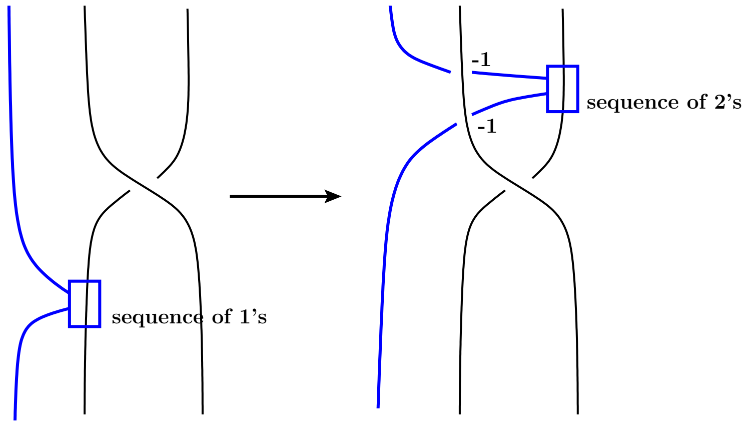


Figure 4.11. crossing c' with sign $v_{c'-1} = 1$, values on strand 1 change to values on strand 2, undercrossings on strand 1 are added before and after the sequence.

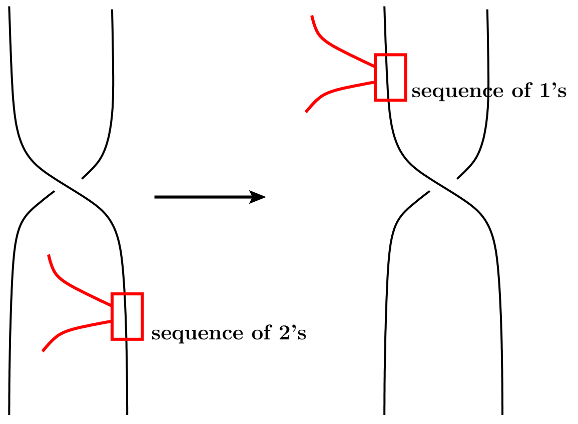


Figure 4.12. crossing c' with sign $v_{c'-1} = 1$, values on strand 2 change to values on strand 1 and no additional crossings are added

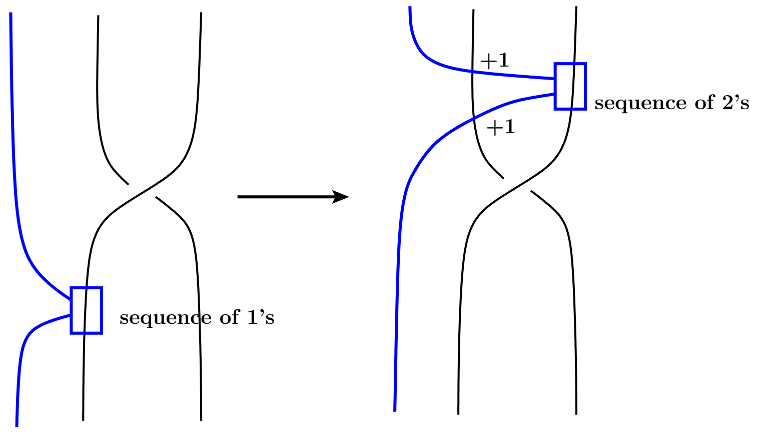


Figure 4.13. crossing c' with sign $v_{c'-1} = -1$, values on strand 1 change to values on strand 2, undercrossings on strand 1 are added before and after the sequence.

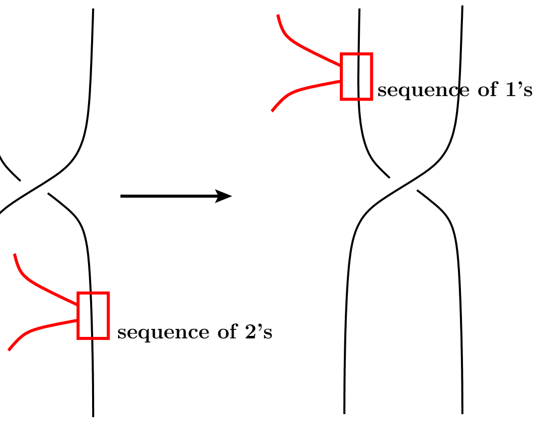


Figure 4.14. crossing c' with sign $v_{c'-1} = -1$, values on strand 2 change to values on strand 1 and no additional crossings are added

4.9 Identifying a Genus 2 Handlebody

The result of Program project is a skein in the complement of a genus n handlebody, where n was the number of strands in the underlying braid. When $n = 2$, we have a skein in the complement of a braid with two non-intersecting strands. For $n = 4$, we address how to pass Program project (a skein in the complement of a 4-braid projected to a section of the handlebody with 4 non-intersecting strands) to the input of Program Resolve and Program Induce (a skein in the complement of a 2-braid, with 2 non-intersecting strands) in the case of all 2-bridge knots. This is the family of knots we consider in this paper. We discuss the use of this program on other families of knots in Chapter 7.

All 2-bridge knots are of the form shown in Figure 4.15, which can be thought of as a 4-braid with two 2-braids attached. The upper 2-braid is called the "upper bridge", and the lower 2-braid is called the "lower bridge". The 2-braids identify pairs of strands of the 4-braid at the top and bottom. We identify a genus 4 handlebody, corresponding to the complement of the 4-braid. The Projection algorithm is applied to skeins in this handlebody, and the result is projected to the upper section of the handlebody, where the 4 strands of the braid are non-intersecting. Next, we use the upper bridge to identify strands 1 and 4 and strands 2 and 3. This produces a skein in a genus 2 handlebody, and to this result we first apply several reduction algorithms (to reduce under-under and over-overcrossing pairs). Then, we use the curve in the genus 2 handlebody as the input for the next phase of the program.

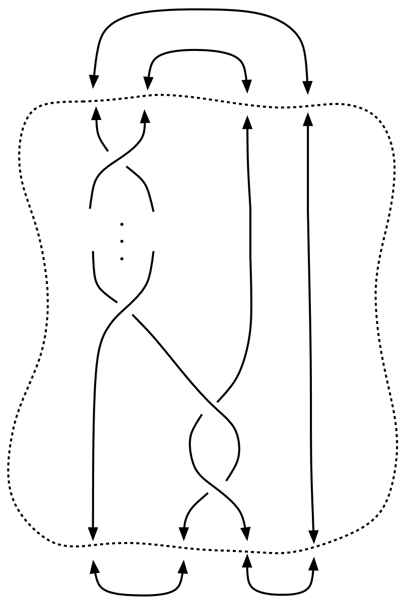


Figure 4.15. 2-bridge knot

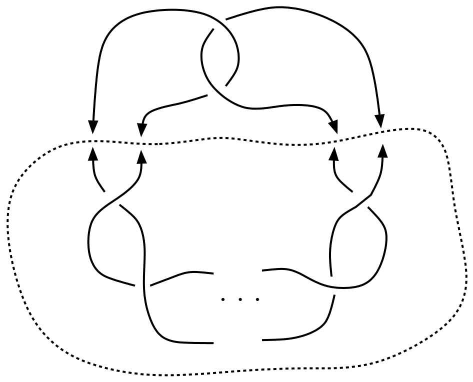


Figure 4.16. twist knot

In our case, a genus 2 handlebody is of particular interest, as every genus 2 handlebody is the cylinder over a 2-holed disk. Resolution in the 2-holed disk produces a specified set of basis elements. This can be easily seen from the following diagram. Any curve in the 2-holed disk either encompasses one hole (the elements x and z), both holes (the element y), or does not encompass either hole at all (the unknot).

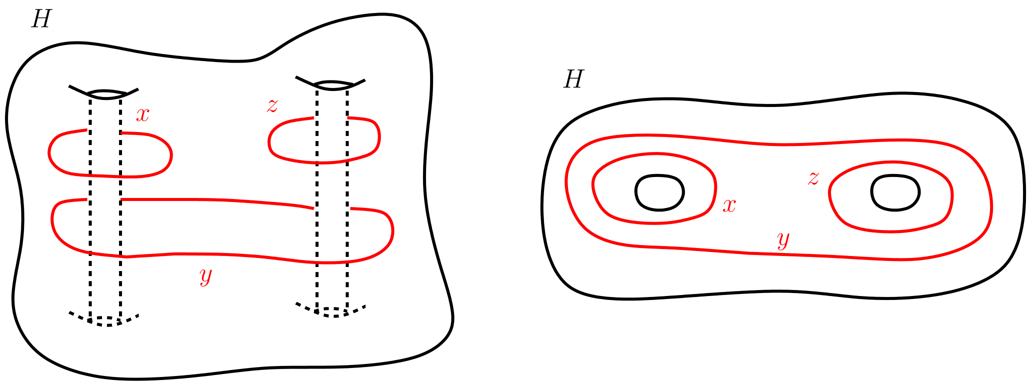


Figure 4.17. Framed curves x, y, and z

With this is mind, realizing a curve contained in the 2-handlebody as a curve contained in a 2-holed disk and resolving all existing crossings produces a polynomial in the basis of the skein module. The question is, how do we accomplish such a projection from given a skein in a genus 2 handlebody? This is a spatial question equivalent to the question: how do we change our view of the skein from a side view to a top view? From the side view, two strands might not seem to cross, but cross when viewed from the top. Similarly, two strands which appear to cross in the side view may not cross in top view. We will address this in the next section.

Before we concern ourselves further, we must first isotope the curve so as to be completely contained in a genus 2 handlebody which as a planar diagram has no intersections. In Lemma 4.2, we show this can be done for every two bridge knot. As a corollary, for the families of knots we consider, twist knots and $(2, 2p+1)$-torus knots, we have a preferred location for this handlebody. For example, all twist knots can be written in the form shown in Figure 4.18; there is a hooked section at the top, and a twisted section at the bottom, with k twists proceeding in the same direction. The preferred location of the handlebody is indicated. Similarly, for 2-bridge knots, shown in Figure 4.19, the preferred location of the handlebody is indicated.

Lemma 4.2. *For every 2-bridge knot, there is a genus 2 handlebody embedded in the knot complement such that every framed curve in the knot complement can be transformed by an isotopy into a framed curve in that handlebody.*

Proof. The complement M of the 2-bridge knot is obtained by adding a 2-handle to a genus 2 handlebody H. View the 2-handle as $D \times [0, 1]$ where D is the unit disk. There is an embedded open disk $D_0 \subset D$ such that $D_0 \times [0, 1]$ does not intersect the framed curve. Then H is a strong deformation retract of $M \backslash D_0 \times [0, 1]$, and consequently the original curve can be isotoped to a curve in H. □

Corollary 4.3. *For every twist knot, every framed curve in the knot complement can be transformed by isotopy so as to lie completely in the genus 2 handlebody shown in Figure 4.18.*

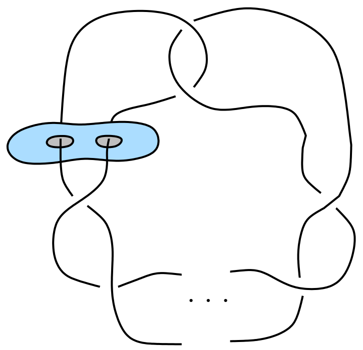


Figure 4.18. Location of the genus 2 handlebody in the particular case of a twist knot

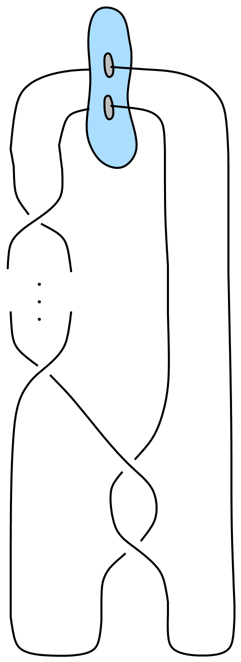


Figure 4.19. Location of the genus 2 handlebody in a 2-bridge knot

4.10 Reducing Curves After Every Stage

Before proceeding to the next step, the complexity of the following calculations can be drastically reduced by performing simplifications, canceling consecutive pairs of overcrossings or undercrossings. This is equivalent to recoginizing instances of Reidemeister moves, in particular, the Reidemeister move R_2. A skein can be reduced when it has consecutive overcrossings or undercrossings as shown in Figure 4.20. In this case, making one such reduction may produce another overcrossing or undercrossing pair, which can be further reduced. While reducing the curve is not necessary to performing the algorithm, thoughtful simplifications in the skein will reduce the number of steps required. The skein can be reduced before the algorithm begins, at each stage, and as a final step. Reduction at each stage drastically improves computation time.

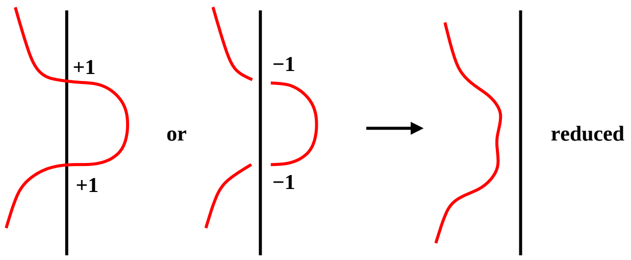


Figure 4.20. Canceling consecutive pairs of crossings (shown here on strand 1)

4.10.1 Reduction Method 1

In a diagram, we recognize a reduction by noting an over-over or under-under pair. In the array, this is a pair of the same numbers with the same sign. For example, the entries $[...1, 1...]$ indicate that the curve passes over strand 1 and then over strand 1 again, undoing each other, and so we cancel them. In another example, the entries $[... - 2, 1, 1, -2, ...]$ indicate under strand 2, over strand 1, over strand 1, under strand 2; canceling the over-over pair results in an under-under pair, which can also be canceled. There is, however, one caveat to this identification: the pairs we identify must be consecutive along a strand. It is possible to have an over-over pair that will not cancel, as shown in Figure 4.21. Hence, we must first identify an over-over or under-under pair, and then check the indices of the pair to determine if they are consecutive along a strand. If the indices are consecutive numbers, then they are certainly consecutive. Yet there is a case where the indices might not be consecutive (compared to all indices), but are still consecutive when compared to all indices along the same strand. Once we determine if such a pair can cancel based on their indices, we cancel the pair from curves array. We must also remember to cancel the corresponding elements from the index array and the orientation array.

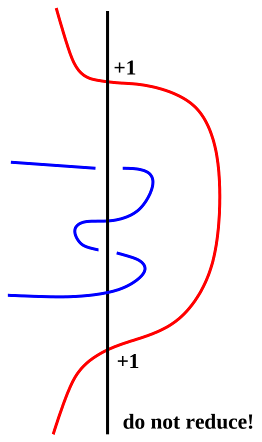


Figure 4.21. Consecutive pairs of crossings which do not cancel

4.10.2 Reduction Method 2

There is one other instance in which we can identify an over-over or under-under pair, but it is a result of our notation (and actually performs the same type of reduction as the first method). In our diagram, the curve is connected, having no fixed starting point. When converting to an array, however, we were required to fix a starting point along each component of the curve, choose an orientation, and list the crossings of the skein beginning and ending at this point. If the first crossing we listed was part of an over-over or under-under pair with the last crossing listed, they would not appear consecutive in the array. Thus we use a separate method to identify these end pairs. For example, the entries $[1, ..., 1]$ form such a pair. Canceling this pair might reveal another pair, as in $[1, -2, ..., -2, 1]$, and so we must continue to check the pairs until the first and last entries are different. Again, we check the index of this pair to determine if they can cancel, and when canceled we cancel them from the intersection array E and cancel their corresponding elements in the index array I and orientation array Q.

4.10.3 Implementing Both Reduction Methods Simultaneously

When we cancel one instance of an over-over pair, we might reveal the other instance. For example, in the curve $[1, 1, -2, ... - 2]$ we must first cancel the over-over pair (using the first reduction method) and then the under-under pair (using the second reduction method). This leads us to recognize that in order to achieve full reduction at any stage, we must first implement the first method to cancel consecutive pairs, and then the second method to cancel end pairs. If we were to implement reductions in the opposite order (using the second reduction method before the first), using the second method would insure that the end pair can not be canceled, but then using the first method might cause the end pair to cancel, as in the example above (canceling the 1's leads to canceling the 2's). There is, however, no instance in which the second method will lead to cancellations in the first method, as canceling the end pairs will not reveal a cancellation that was not already present. Hence, we apply the methods in the given order, applying the first reduction method and following with the second.

We begin by scanning the array, front to back, for over-over or under-under pairs using the first method. Starting with $r = 0$, we check the curve array for the condition that $Z^1_{z,r} = Z^1_{z,r+1}$ for all r from 0 to $len(Z^1_z) - 2$, since the last index of Z^1_z is $len(Z^1_z) - 2 + 1 = len(Z^1_z) - 1$. When we find a pair, we check their indices. Let s be the index of the first crossing in the pair. Then the indices of the pair are $Z^4_{z,s}$ and $Z^4_{z,s+1}$. We determine which of these is the smaller index of the two, indicating it is lower (in terms of height) than the other. Without loss of generality, let $Z^4_{z,s}$ be the smaller of the two. We then need only check for the indices between $Z^4_{z,s}$ and $Z^4_{z,s+1}$. If $Z^4_{z,s}$ and $Z^4_{z,s+1}$ are consecutive, there are no indices between them, and we are done. If not, let S be the index between them, so $Z^4_{z,s} < S < Z^4_{z,s+1}$. Then we identify t such that $Z^4_{z,t} = S$. If $|Z^4_{z,t}| = |Z^4_{z,s}|$, then this is an instance of a crossing on the same strand with an index between the over-over (or under-under) pair, hence, they are not consecutive. If no t can be found such that $|Z^4_{z,t}| = |Z^4_{z,s}|$, then the pair is consecutive relative to the strand, and can thus be canceled.

Implementing this reduction between steps in the program and between iterations in each step drastically reduces the overall computation time.

4.11 Input for Program Resolve and Induce

In Program Resolve and Program Induce, we work with a skein in the complement of an 2-braid, that is, in a genus 2 handlebody. In the diagram in which we work, the braid B has two non-intersecting strands, strand 1 and strand 2. The skein K is a framed link with k components. For each component, K^l, $0 \leq l \leq k-1$, we record three pieces of input information: intersection information (E^l), index information (I^l), and orientation information (Q^l). We now outline how to record this input information.

4.11.1 Skein Crossing Input

We begin by labeling all intersections in the skein (self-intersections and intersections between components of the skein). We distinguish self-intersections of the skein and intersections between the skein and the base braid. Self-intersections of the skein (or intersections between one component of the skein and another) are labeled $1 \times 10^{-k_r}, 2 \times 10^{-k_r}, ..., r \times 10^{-k_r}$, where k_r is the magnitude of the number of crossings, and the labels are assigned to these self-intersections in any order as shown in Table 4.11.1. Indeed, the order on these crossings does not matter, as the order only dictates which crossing will be resolved first, second, etc.

We include these labels in an array, S:

$$\boxed{S = [1 \times 10^{-k_r}, 2 \times 10^{-k_r}, ..., r \times 10^{-k_r}]}$$

where k_r is the magnitude of the number of crossings. For clarity, this value is tabulated as follows:

Skein Crossing Information
For a skein with 1-99 crossings, $k_r = 2$. The crossings are labeled $S = [0.01, 0.02, ..., 0.99]$
For a skein with 100-999 crossings, $k_r = 3$. The crossings are labeled $S = [0.001, 0.002, ..., 0.999]]$
For a skein with 1000-9999 crossings, $k_r = 4$. The crossings are labeled $S = [0.0001, 0.0002, ..., 0.9999]$

Next, we assign an orientation to each component of the skein. The orientation on each component affects the sign of the crossings within that component. For the r crossings to be resolved, we record an array containing the signs of the crossings based on the assigned orientations:

$$\boxed{U = [\pm 1, \pm 1, ..., \pm 1]}$$

where $dim(U) = r$. As each crossing is resolved, we delete the corresponding element of U. Since crossings are resolved in order, the first element will always be deleted, leaving an array of dimension 1 less that the previous stage. The computation terminates when all crossings have been resolved, and $dim(U) = 0$.

4.11.2 Skein Intersection Input

It is worth noting that, among all components of the skein, if there are s intersection points between the skein and the base braid and r intersection points between the skein and itself, then there are $s+r$ points of interest we consider. We will record $s+2r$ points, among all components, as the label on each crossing is recorded twice, once for the overcrossing $(0.r)$ and once for the undercrossing $(-0.r)$. In each component, l, we say there are p_l points of interest.

Let k be the number of components of the link that represents the skein, labeled 0 through $k-1$. For each component, K^l, $0 \leq l \leq k-1$, we form an array

$$\boxed{E^l = [e_0^l, e_1^l, ..., e_{(p_l)-1}^l]}$$

where e_j^l, $0 \leq j \leq (p_l) - 1$, is determined by the following chart:

Skein Intersection Information (for a base braid with 2 non-intersecting strands)
If the skein crosses over strand 1 of the base braid, we record a "1", so $e_j^l = 1$.
If the skein crosses under strand 1 of the base braid, we record a "-1", so $e_j^l = -1$
If the skein crosses over strand 2 of the base braid, we record a "2", so $e_j^l = 2$.
If the skein crosses under strand 2 of the base braid, we record a "-2", so $e_j^l = -2$.
If the skein crosses over an intersection $0.r'$, we record a "$0.r'$", so $e_j^l = 0.r'$.
If the skein crosses under an intersection $0.r'$, we record a "$-0.r'$", so $e_j^l = -0.r'$.

Figure 4.22 shows this labeling for the case of a skein with one component. When the curve we consider goes over the strand, the number of that strand is recorded

with a positive sign; when the curve goes under, the number of that strand is recorded with a negative sign.

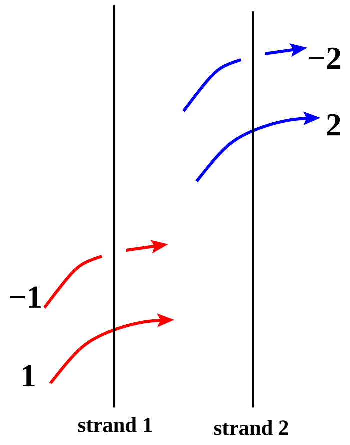


Figure 4.22. Labeling intersections between the skein and the base braid.

Now consider the case in which there are two components in the curve. The components may intersect each other, but each component may also intersect itself in several locations. It is worth noting that there is no difference between the way we label self-intersections (in the same component of the skein) versus how we label intersections between two different components of the skein. The difference is only realized by investigating the recorded arrays. If the crossing is a self-crossing, both labels "0.01" (the overcrossing) and "-0.01" (the undercrossing) will appear in the same component of the array E. Otherwise, the overcrossing will appear in one component of the array, and the undercrossing will appear in another component of the array. For example, in Figure 4.23, the diagram on the left has two components. Each one component contains a crossing labeled -0.01 and a crossing labeled 0.02. The other component contains a crossing labeled 0.01 and a crossing labeled -0.02. Together, they contain all information about both crossings. In the diagram on the right, the same component contains the label 0.01 and the label -0.01.

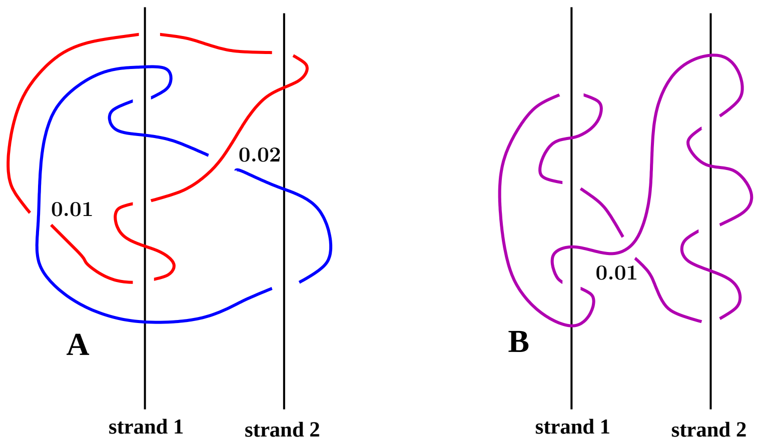


Figure 4.23. Intersections vs Self-intersections.

Figure 4.24 shows an example on a simple curve in the knot complement:

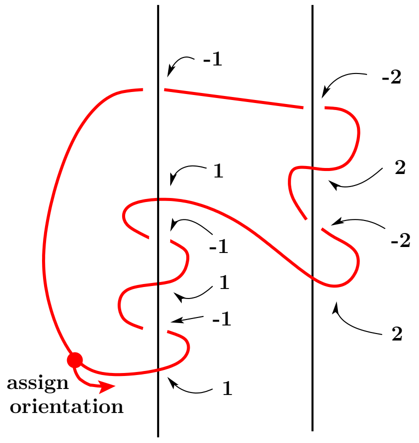


Figure 4.24. Labeling overcrossings and undercrossings, recording the array E

We record E^l for each component $0 \leq l \leq k-1$, and from this we form the skein intersection array:

$$\boxed{E = [E^0, ..., E^{k-1}]}$$

where k is the number of components in the skein.

4.11.3 Skein Index Input

Thus far, the concept of "highest" and "lowest" has been determined based on proximity to the top and bottom of the braid. Crossings closer to the top of the braid are considered higher than crossings closer to the bottom of the braid. While sufficient for recording information in the base knot, this ambiguity leaves room for multiple definitions of the (multi)curve representing the skein as it intersects the strands of the base braid. Figure 4.25 shows an ambiguous case in which the same strand information describes two different curves. To distinguish between these cases, we separately assign a height index (relative to the datum we established at the bottom of the diagram). This additional information serves as the connecting information between components of the (multi)curve, producing a sense of relative position. Although we aim for a unique result, the way we assign such an index is not unique. As noted in the example, ambiguity only arises in determining position along a single strand of the base braid, hence, we only require indexing to be sequential along each strand. Those places where the (multi)curve crosses a strand of the base braid must be ordered by index based on their proximity to the datum. Crossings, both self-intersections and crossings between different components, are also indexed, but their indices are not required to be sequential or even relate to the indices of the other points of interest. In fact, we can use an entirely different system of indices for the crossings with no bearing on the result. Figure 4.26 shows two valid ways of indexing intersections between the skein and the braid. The first is by height in which the index is assigned strictly based on distance from a datum at the bottom of the diagram. Points of interest close to the datum (towards the bottom of the diagram) are indexed by lower numbers and points far away from the datum (towards the top of the diagram) are indexed by higher indices, regardless of whether they are self-intersections of the skein or not and regardless of which strand of the base braid they intersect.

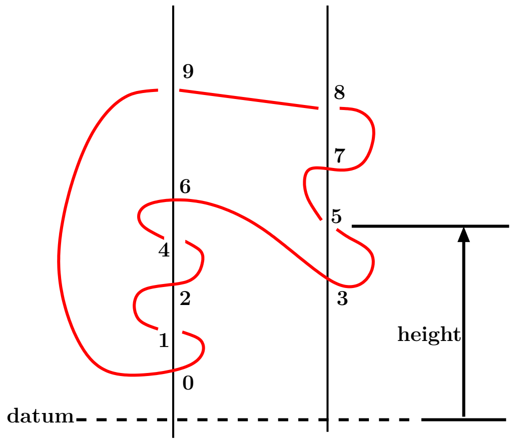


Figure 4.25. Assigning the index based on position from the datum.

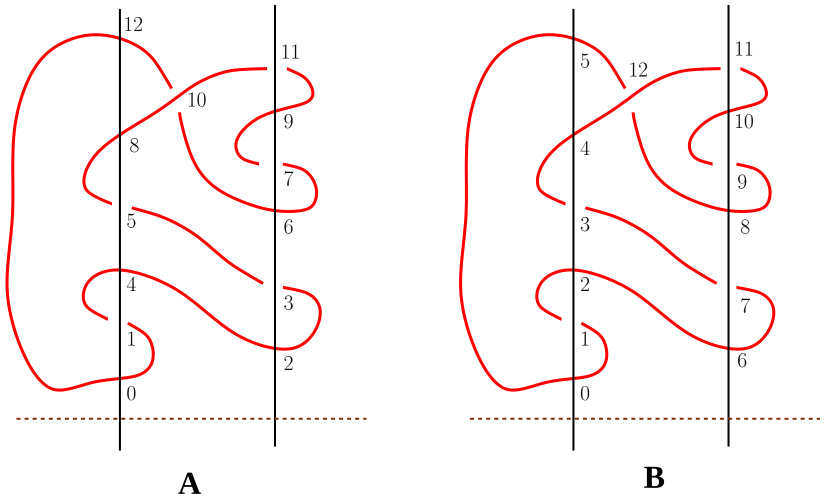


Figure 4.26. Two basic ways to index a curve: (A) by height, and (B) sequentially along each strand.

For each component l, we form the array:

$$\boxed{I^l = [i^l_0, i^l_1, ..., i^l_{(p_l)-1}]}$$

by starting at the point of interest corresponding to E^l_0 (the first entry in E^l), and recording the index i^l_j point by point. We proceed in the same order as the elements of E^l, so that the entries of I^l correspond to the entries of E^l.

We record I^l for each component $0 \leq l \leq k-1$, and from this we form the index array:

$$\boxed{I = [I^0, ..., I^{k-1}]}$$

where k is the number of components in the skein.

4.11.4 Skein Orientation Input

The last case of ambiguity is the order in which we record information, that is, the orientation we assign to the curve. After projecting the skein to a genus 2 handlebody, we are left with a curve whose orientation could be in either direction. Hence, we record the orientation of the curve by recording from which side of the strand we travel. The orientation is determined by recording from which side of the strand we travel. Given a strand n, we label the region to the left of the strand $n+2$ and we label the region to the right of the strand $n+3$. These labels are chosen so as not to coincide with strand labels. So, when the curve passes from the left of the strand to the right of the strand, we assign the value $n+2$ as the orientation. When the curve passes from the right of the strand to the left of the strand, we assign the value $n+3$ as the orientation. Since $n = 2$ here, the numbers we use here, 3 4,5, are chosen arbitrarily, and could be any consecutive integers (3 was chosen as the first of these integers to avoid conflicts with the labeling of 2-stranded braids). The important thing to notice is that an increase in these orientation numbers indicates a move from left to right, and a decrease indicates a move from right to left.

We already have a sense of orientation from the arrays E and I. In each component, with intersection array E^l and index array I^l, we traverse the component of the skein in the same direction. We make this direction explicit by assigning a sequence of numbers to determine when we are traveling left to right versus right to left as the skein passes back and forth between strands 1 and 2 of the base braid. In most cases, this information is extraneous. However, it is required to uniquely determine the skein.

For each component l of the skein, we form an array:

$$\boxed{Q^l = [q^l_0, q^l_1, ...q^l_{(p_l)-1}]}$$

where q^l_j, $0 \leq j \leq (p_l) - 1$, is determined by the following chart:

Skein Orientation Information
If the skein crosses over strand 1 from left to right, we record a "3", so $q_j^l = 3$.
If the skein crosses over strand 1 from right to left, we record a "4", so $q_j^l = 4$.
If the skein crosses over strand 2 from left to right, we record a "4", so $q_j^l = 4$.
If the skein crosses over strand 2 from right to left, we record a "5", so $q_j^l = 5$.
If the skein passes a self-intersection, we record a "6", so $q_j^l = 6$.

In general, this notation extends to a braid with n strands, which can be seen as a diagram with $n+1$ sections, labeled $3, ..., n+3$. Crossing from section k to $k+1$ is assigned the value k, while passing from section $k+1$ to k assigned the value $k+1$. In our case, we work in a genus 2 handlebody whose base braid has only two strands, and so the labeling is as shown in Figure 4.27.

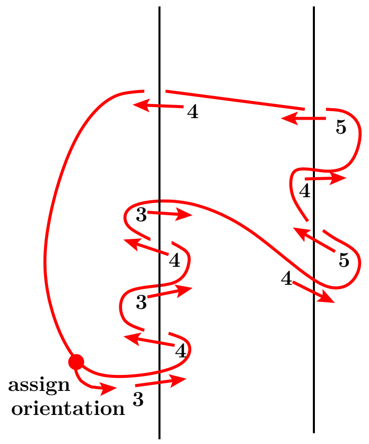


Figure 4.27. Labeling orientation based on orientation assigned to skein.

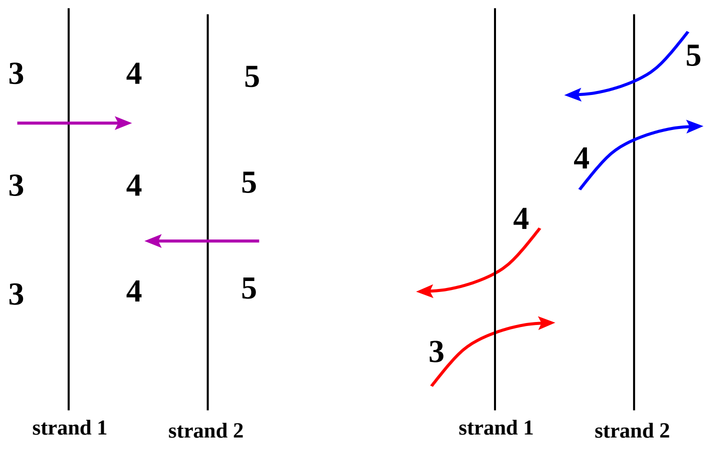


Figure 4.28. Recording orientation information.

We record Q^l for each component $0 \leq l \leq k-1$, and from this we form the skein intersection array:

$$\boxed{Q = [Q^0, ..., Q^{k-1}]}$$

where k is the number of components in the skein.

4.11.5 Skein Input Array

In describing the skein, we produced three arrays: the intersection array E, index array I, and orientation array Q. These arrays record all the necessary information about all components of the skein, the relative position of the different components with respect to each other, and the orientation information along each component. We now include all this information in an array of the form

$$\boxed{Z_0 = [0, E, 1000, U, I, Q]}$$

which is of the form

$$Z_0 = [s_0, E_0, c_0, u_0, I_0, Q_0]$$

where, $s_0 = 0$ is the initial power of the variable t, and $c_0 = 1000$ is the sign and value of the initial constant, which is 1. We recover our original skein as $c_0 t^{s_0} K_0 = 1t^0 K$, where $K = [E, I, Q]$, the skein uniquely represented by its intersection array, index array, and orientation array as in the following theorem. At each stage of the computation, we resolve one of the r crossings, producing 2^r terms. The final result is an array of the form

$$Z = [Z_0, Z_1, ..., Z_{2^r-1}]$$

where each component Z_i, representing one term of the final answer, is of the form

$$Z_z = [s_z, E_z, c_z, u_z, I_z, Q_z].$$

Here,

- s_z: power of the variable t;
- E_z: the intersection array of term z;
- c_z: sign and value of the constant, (...2000 = 2, 1000 = 1, 0 = 0, -1000 = -1, -2000 = -2,...);
- u_z: information on the crossings resolved in the last stage;
- I_z: index array of the term z;
- Q_z: orientation array of the term z.

We recover a linear combination of skeins in the variable t from Z as:

$$c_0 t^{s_0} K_0 + ... + c_z t^{s_z} K_z + ... + c_{2^r-1} t^{s_{2^r-1}} K_{2^r-1}$$

where $K_z = [E_z, I_z, Q_z]$ is the unique skein recovered from the intersection array, index array, and orientation array as in the following theorem. The array u_z stores information about the signs of the crossings left to be resolved, and is only used to perform the next stage of the computation. In the final result, $dim(u_z) = 0$, as their are no terms left to resolve. This is stated in the following proposition.

Proposition 4.4. *A relation between skeins at any stage produced by the resolution of crossings using a set of given skein relations can be recovered from the array* $Z = [Z_0, Z_1, ..., Z_{2^r-1}]$*, where* $Z_z = [s_z, E_z, c_z, u_z, I_z, Q_z]$ *for* $0 \leq z \leq 2^r - 1$*, where* r *is the number of crossings to be resolved.*

Finally, we note that every skein $K = [E, I, Q]$ can be uniquely recovered from this required input information.

Theorem 4.5. *Any skein K embedded in a genus 2 handlebody can be uniquely recovered from the arrays E, I, and Q by $K = [E, I, Q]$. This representation produces a unique skein (or relation of skeins), but such a skein K may have multiple equivalent representations.*

Proof. We recover the skein K from the given arrays E (the intersection array), I (the index array), and Q (the orientation array). In each component, l, we recover $K^l = [E^l, I^l, Q^l]$ as follows: the dimension of the arrays $dim(E^l) = dim(I^l) = dim(Q^l) = n$ is the number of points of interest we consider. Beginning with the array I^l, we will label points along each strand of the base braid with the sequential index values, from bottom to top by partitioning the array I^l into 3 sets: I_1^l, I_2^l, and I_r^l, according to the corresponding values in E^l. I_1^l contains all elements i_j^l for which $|e_j^l| = 1$. I_2^l contains all elements i_j^l for which $|e_j^l| = 2$. Finally, $I_r^l = I \setminus (I_1 \cup I_2)$ contains all the crossing labels, both overcrossing $0.r$ and undercrossings $-0.r$. This induces a partition on E^l (into E_1^l, E_2^l, and E_r^l) and Q^l (into Q_1^l, Q_2^l, and Q_r^l). Let $n_1 = dim(I_1^l)$, $n_2 = dim(I_2^l)$, and $n_r = dim(I_r^l)$. The value of the elements of I_1^l determines an ordering along strand 1. Similarly for I_2^l. Begin with the minimal element of I_1^l (the element with the lowest index), and consider the corresponding value of E_1^l: if it is 1 we draw an overcrossing, and if it is -1 we draw an undercrossing at this point. Use the corresponding value of Q_1^l to determine if the skein passes left to right or right to left at this crossing. Returning to the original array, I^l, beginning with i_0^l, we connect the skein by the corresponding elements of E^l. Whenever we come to an element $e_j^l = \pm 0.r$, we use the orientation of the previous element q_{j-1}^l to determine between which two strands of the base braid to place the skein intersection. In this manner, we proceed component by component to recover the original knot diagram.

Finally, each skein K may have multiple, equivalent representations. Indexing the skein based on height, we only require the index to be sequential on points of intersection along each strand. The index labeling on one strand of the base braid does not affect the labeling on any other strands. Additionally, the index labeling on self-intersections is independent of the labeling along each strand. Hence, the index array I^l used in each component is not unique. □

For clarity, we apply the method outlined in the proof to the following skein.

4.12 Implementing Proof of Theorem 4.5

To illustrate the method of the proof of Theorem 4.5, consider the following skein $K = [E, I, Q]$ given by:

$$E = [[1, -1, 1, -0.01, 2, -2, 0.01, -2, 2, -1, 1, 2, -2, 1]]$$
$$I = [[0, 1, 2, 3, 5, 4, 3, 6, 7, 8, 9, 10, 11, 12]]$$
$$Q = [[3, 4, 3, 4, 4, 5, 4, 4, 5, 4, 3, 4, 5, 4]]$$

We partition I into the following sets:

$$I_1 = [0, 1, 2, 8, 9, 12]$$
$$I_2 = [4, 5, 6, 7, 10, 11]$$
$$I_r = [3, 3]$$

To reconstruct the skein, we label points on strand 1 of the base braid by the elements of I_1, and points on strand 2 by the elements of I_2. For the elements of I_r, we consider the corresponding elements in Q: the elements at index 3 in I correspond to the crossings ± 0.01 in E and have orientation value 4 in Q. Recall that the orientation is determined by a labeling on the sections to the left of strand 1 (orientation value 3), between strand 1 and strand 2 (orientation value 4), and to the right of strand 2 (orientation value 5). The index 4 indiciates that this crossing lies between strand 1 and strand 2, so we label a point in the space between the strands where the crossing will occur with index value 3. This is shown in Figure 4.29.A. Next, we use the values of Q corresponding to the elements of I_1 to draw a segment of the skein at each point on strand 1, shown in Figure 4.29.B. We do the same for strand 2. Finally, we connect the segments with respect to this orientation, following the ordering given by E. This determines a unique skein $K = [E, I, Q]$.

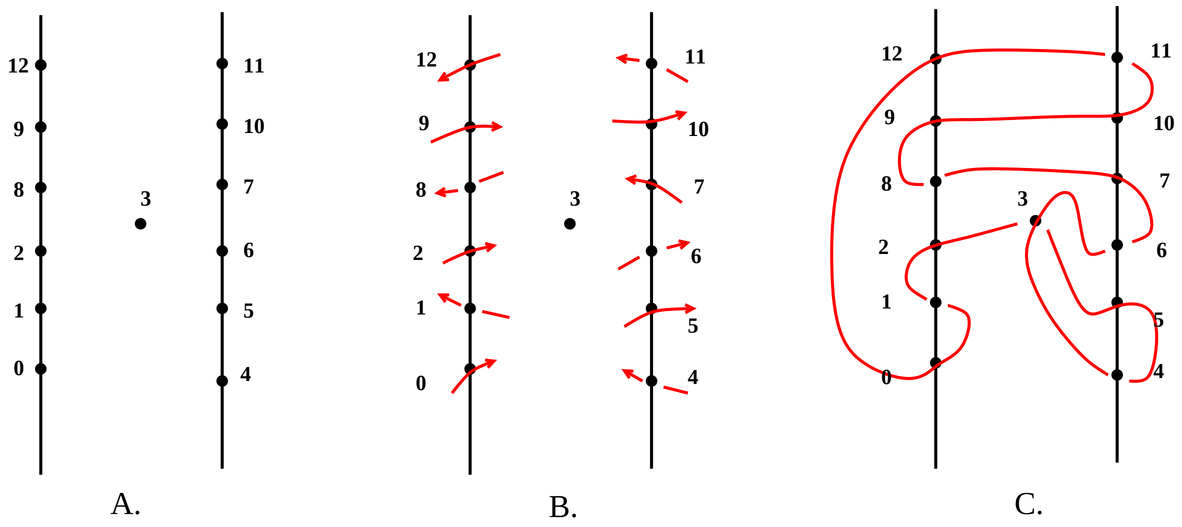


Figure 4.29. Recontructing the skein $K = [E, I, Q]$

4.13 Program Resolve Algorithm

In each component, $Z[z]$ of Z, we will resolve the crossings $1 \times 10^{k_r}, 2 \times 10^{k_r}, ..., r \times 10^{k_r}$, where k_r is the magnitude of the number of crossings, beginning with the first crossing. Without loss of generality, let us assume the crossings are labeled $0.01, 0.02, ...$ Beginning with $Z = [Z_0]$, we resolve crossing 0.01, producing two terms. The array Z has one element, the term Z_0, given by:

$$Z = [Z_0] = [[s_0, E, c_0, u_0, I, Q]]$$

where E, I, and Q are the initial input arrays, and $s_0 = 0$, $c_0 = 1000$, and $u_0 = [1]$.

In the element Z_0, crossing 0.01 is resolved first, and the element Z_0 is then replaced by two terms, Z_0' and Z_0'', produced by applying the skein relation:

$$Z = [Z_0', Z_0''] = [[r', E', c', u', I', Q'], [r'', E'', c'', u'', I'', Q'']]$$

after such a replacement, we can re-index the entries of Z as:

$$Z = [Z_0, Z_1]$$

In each of these entries, the crossing 0.01 has been resolved. We again consider the element Z_0, resolving the crossing 0.02. This produces 2 new terms, Z'_0 and $Z"_0$, and we replace Z_0 in Z:

$$Z = [Z'_0, Z''_0, Z_1]$$

The crossing 0.02 has been resolved in each of the components Z'_0 and Z''_0. Next we consider the element Z_1 and resolve the crossing 0.02. This produces 2 new terms, Z'_1 and Z''_1, and we replace Z_1:

$$Z = [Z'_0, Z''_0, Z'_1, Z''_1]$$

Again, we can re-index the entries of Z after replacement in each component:

$$Z = [Z_0, Z_1, Z_2, Z_3]$$

Note that after the resolution of 2 crossings, we have produced $2^2 = 4$ terms. Proceeding in this manner, we resolve all labeled crossings, resulting in an array Z of dimension 2^r, where r is the number of crossings. The result is realized as a linear combination of these entries, each containing the coefficient and power of the variable t specified by the given skein relations.

$$Z = [Z_0, Z_1, ..., Z_{2^r-1}]$$

where each Z_i is a skein containing no self-intersections.

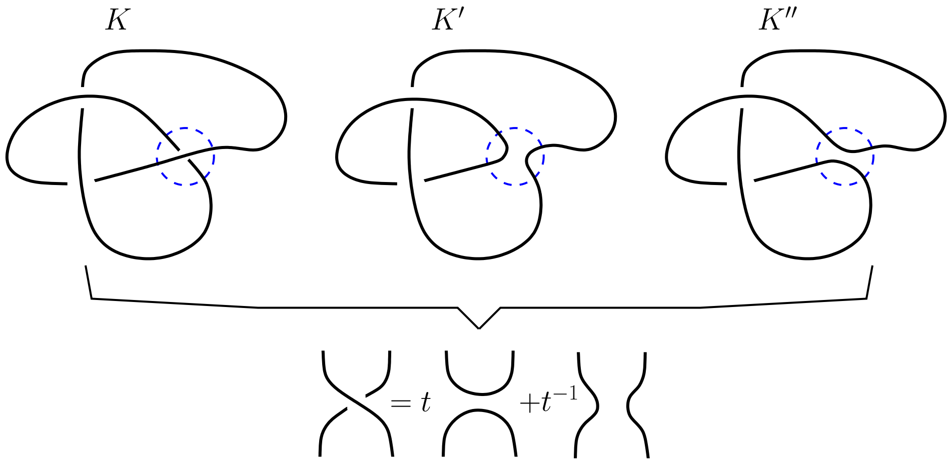
The only remaining question is to determine the values of the terms Z'_i and Z''_i which replace Z_i after resolution of a crossing. We know that Z'_i and Z''_i are of the form:

$$Z_i = [s_i, E_i, c_i, u_i, I_i, Q_i]$$
$$Z'_i = [s'_i, E'_i, c'_i, u'_i, I'_i, Q'_i]$$
$$Z''_i = [s''_i,''_i, c''_i, u''_i, I''_i, Q''_i]$$

When written as diagrams with coefficients in t, we have the relation

$$c_i t^{s_i} K_i = t(c'_i t^{s'_i} K'_i) + t^{-1}(c''_i t^{s''_i} K''_i),$$

where $K_i = [E_i, I_i, Q_i]$, $K'_i = [E'_i, I'_i, Q'_i]$, and $K''_i = [E''_i, I''_i, Q''_i]$. Given the skein relation, $K = tK' + t^{-1}K''$, we can see that $s_i' = s_i + 1$, $s_i'' = s_i - 1$, $c' = c = c''$. As for the skeins $K' = [E', I', Q']$ and $K'' = [E'', I'', Q'']$, they are related to $K = [E, I, Q]$ as shown in Figure 4.30. Thus, we must edit the array K to match the skeins K' and K'' shown in the Figure. However, the way we edit these arrays depends on several factors. If the crossing resolved is an intersection within the same component of the skein, only that component will be edited. If the crossing resolved is between two components of the skein, both components involved must be edited. We now consider both cases, and the arrays K' and K'' formed from K in each case.



$K = tK' + t^{-1}K''$

Figure 4.30. The relation between skeins K, K', and K''

4.13.1 Locating a Crossing

We start by locating the crossing in the array Z. Without loss of generality, consider resolving the crossing labeled 0.01 (the same discussion certainly extends to all crossings). We will resolve this crossing in each component of Z, so let us fix an element Z_i. When resolving the first crossing, 0.01, there is initially only one component of the array Z, which is split into two components after the resolution of the crossing. When resolving the second crossing, there are initially two components, each of which are split into two components, resulting in four components after the resolution of the crossing. When resolving the third crossing, there are initially four components which results in eight components after the resolution of the crossing, etc.

Each Z_i is of the form

$$Z_i = [s_i, E_i, c_i, u_i, I_i, Q_i].$$

Here, E_i is of the form

$$E_i = [E_i^0, E_i^1, ..., E_i^{k-1}].$$

Let $p = dimE_i$. We search through the array E_i, component by component, until we locate the entry 0.01 and -0.01, that is, $e^{l'}_{i,j'} = 0.01$ for some $0 \leq j' \leq p-1$ and $0 \leq l' \leq k-1$ and $e^{l''}_{i,j''} = -0.01$ for some $0 \leq j'' \leq p-1$ and $0 \leq l'' \leq k-1$. If $l' = l''$, the overcrossing (0.01) and the undercrossing (-0.01) are in the same component (Case 1). Otherwise, they are in different components (Case 2).

4.13.2 Case 1: Crossing Resolved in Same Component

If the overcrossing (0.01) and the undercrossing (-0.01) are in the same component, then $l' = l'' = l$ and only that component, E_i^l, needs to be edited. We actually form two components from the one component by considering the portion of the skein as follows: the overcrossing (0.01) and the undercrossing (-0.01) form a pair, with one portion of the skein between this pair and the other portion of the skein outside of this pair. When we break the crossing, we reattach the top and bottom of the former crossing in K to make the skein K', and we reattached the left and right of the former crossing to make the skein K''. This amounts, in one case, splitting off the portion between the crossings as a new component and, in the other case, reordering the portion of the skein between the crossings.

When we edit E_i^l, we will also edit I_i^l and Q_i^l, however, when editing the orientation array, some additional work is required as some portion of the skein is reversed while the rest remains in the same orientation. We form EA_i^l and EB_i^l from the skein E_i^l as explained in what follows.

Without loss of generality, let $j' < j''$. To form skein EA_i^l, we split the one component E_i^l into two components EA_1 and EA_2. First form a component EA_1 containing only the elements of index $j'+1$ through $j''-1$, $EA_1 = [e^{l'}_{i,j'+1}, e^{l'}_{i,j'+2}, ..., e^{l'}_{i,j''-2}, e^{l'}_{i,j''-1}]$. Form the second component EA_2 by removing the elements of index j' through j'' from E_i^l. Replace E_i^l in E_i by these components EA_1 and EA_2 ($EA_i^l = EA_1 \cup EA_2$). Note that the array EA_i^l has two components.

To form skein EB_i^l, reorder elements in the skein E_i^l by reversing the order of the elements of index $j'+1$ through $j''-1$, and removing the elements of index j' and j'': $EB_i^l = [e_{i,0}^l, e_{i,1}^l, ..., e_{i,j'-1}^l, e_{i,j''-1}^l, ..., e_{i,j'+1}^l, e_{i,j''+1}^l, ..., e_{i,p-1}^l]$, where $p = dimE_i^l$.

The element E_i^l, component l in the array E_i, is replaced by the two components of EA_i^l, EA_1 at index l and EA_2 at index $l+1$. This forms the array EA_i. To form the array EB_i, we replace by E_i^l by EB_i^l at index l. The replacement of E_i^l by EA_i^l increasing the dimension by 1. The replacement of E_i^l by EB_i^l does not increase the dimension.

We edit the arrays I_i^l and Q_i^l in the same way to form IA_i^l, QA_i^l, IB_i^l, and QB_i^l. However, when editing the orientation array, there is one additional step: we must modify the orientation of any portion of the skein whose direction was reversed. In the array, EA_i^l, orientation is preserved, so no changes are necessary. In the array EB_i^l, the portion of the skein from index $j'+1$ to $j''-1$ is reversed, so we edit the elements of the index array QB_i^l between index $j'+1$ and $j''-1$. For each element $q_{i,s}^l$, where $j'+1 \leq s \leq j''-1$, we reverse the orientation number. If $q_{i,s}^l = 3$ (right across strand 1), we change it to $q_{i,s}^l = 4$ (left across strand 1). If $q_{i,s}^l = 5$ (left across strand 2), we change it to $q_{i,s}^l = 4$ (right across strand 2). If $q_{i,s}^l = 4$, it could be either the skein passes left across strand 1 or right across strand 2. In this case, we look at the corresponding element $e_{i,s}^l$. If $|e_{i,s}^l| = 1$, the skein passes left across strand 1, so we change the orientation number to $q_{i,s}^l = 3$ (right across strand 1). If $|e_{i,s}^l| = 2$, the skein passes right across strand 2, so we change the orientation number to $q_{i,s}^l = 5$ (left across strand 2).

The only element we have left to modify is the element u_i, which contains the signs of the crossings left to be resolved. We modify u_i to produce the elements ua_i and ub_i by considering which of the remaining crossings are affected in the resolution of the crossing in this step. Any crossings included in portion of the skein which is reversed may experience a sign change. Each crossing $0.r$ has two components: the overcrossing $(0.r)$ and the undercrossing $(-0.r)$. If both components are contained in the portion of the skein which is reversed, the direction of both will change, and the sign on the crossing will remain the same. If only one component of any crossing, either the overcrossing or the undercrossing, is contained in the portion of the skein which is reversed, and the other is unaffected, the sign on the crossing will change. We determine this by inspecting the elements of E_i^l between index $j'+1$ and $j''-1$.

For each crossing $0.r$, if no element $|e^l_{i,s}| = 0.r$ in the interval $j'+1 \leq s \leq j''-1$, the sign of crossing $0.r$ is unaffected. If there is one element for which $|e^l_{i,s}| = 0.r$ in the interval $j'+1 \leq s \leq j''-1$, the sign of the crossing $0.r$ is reversed, as the orientation on only one component of the crossing was reversed. If there are two elements for which $|e^l_{i,s}| = 0.r$ in the interval $j'+1 \leq s \leq j''-1$, the crossing is unaffected, as the orientation of both components of the crossing was reversed.

Thus the skein $K_i = [E_i, I_i, Q_i]$ is replaced by skeins $KA_i = [EA_i, IA_i, QA_i]$ and $KB_i = [EB_i, IB_i, QB_i]$. Hence, we replace the component $Z_i = [s_i, E_i, c_i, u_i, I_i, Q_i]$ by $ZA_i = [s_i + 1, EA_i, c_i, ua_i, IA_i, QA_i]$ and $Z_i = [s_i - 1, EB_i, c_i, uB_i, IB_i, QB_i]$.

4.13.3 Case 2: Crossing Resolved in Different Components

If the overcrossing (0.01) and the undercrossing (-0.01) are in the different components, both $E^{l'}_i$ and $E^{l''}_i$ need to be edited. More precisely, we take the two components, one which contains the overcrossing and one which contains the undercrossing, and join them together as a single component in two different ways: When we break the crossing, we reattach the top and bottom of the former crossing in K to make the skein K', and we reattached the left and right of the former crossing to make the skein K''. This amounts, in one case, joining the arrays at the location of the overcrossing and, in the other case, joining the arrays in the location of the undercrossings.

When we edit $E^{l'}_i$, we will also edit $I^{l'}_i$ and $Q^{l'}_i$. Similarly, when we edit $E^{l''}_i$, we will also edit $I^{l''}_i$ and $Q^{l''}_i$. Again, the orientation array is edited in certain instances to account for reversing the direction of a portion of the skein.

We have located the overcrossing in component l' at index j', and the undercrossing in component l'' at index j''. We form $EA^{l'}_i$ and $EB^{l''}_i$ from the skeins $E^{l'}_i$ and $E^{l''}_i$ as follows.

Let $p' = dim(E^{l'}_i)$ and $p'' = dim(E^{l''}_i)$. To form skein $EA^{l'}_i$ (which replaces skein $E^{l'}_i$ after the computation), we form a component containing the elements of $E^{l'}_i$ from index 0 to $j'-1$, then the elements of $E^{l''}_i$ from index j'' to $p''-1$, then the elements of $E^{l''}_i$ from 0 to $j''-1$, and finally the elements of $E^{l'}_i$ from $j'+1$ to $p'-1$. Thus,

$$EA^{l'}_i = [e^{l'}_{i,0}, ..., e^{l'}_{i,j'-1}, e^{l''}_{i,j''+1}, ..., e^{l''}_{i,p''-1}, e^{l''}_{i,0}, ..., e^{l''}_{i,j''-1}, e^{l'}_{i,j'+1}, ..., e^{l'}_{i,p'-1}].$$

To form the skein $EB^{l''}_i$ (which replaces skein $E^{l''}_i$ after the computation), we form a component containing the elements of $E_i^{l'}$ from index 0 to $j'-1$, then the elements of $E_i^{l''}$ from index $j''-1$ to 0 (traversing the portion of the skein in the reverse direction), then the elements of $E_i^{l''}$ from $p''-1$ to $j''+1$ (traversing the portion of the skein in the reverse direction), and finally the elements of $E_i^{l'}$ from $j'+1$ to $p'-1$. Thus,

$$EB_i^{l'} = [e_{i,0}^{l'}, ..., e_{i,j'-1}^{l'}, e_{i,j''-1}^{l''}, ..., e_{i,0}^{l''}, e_{i,p''-1}^{l''}, ..., e_{i,j''+1}^{l''}, e_{i,j'+1}^{l'}, ..., e_{i,p'-1}^{l'}].$$

The elements $E_i^{l'}$ and $E_i^{l''}$, components l and l' in the array E_i, are replaced by the component $EA_i^{l'}$ at index l' to form the array EA_i. To form the array EB_i, we replace by $E_i^{l'}$ and $E_i^{l''}$ by $EB_i^{l''}$ at index l''. The replacement of $E_i^{l'}$ and $E_i^{l''}$ by $EA_i^{l'}$ decreases the dimension by 1. The replacement of $E_i^{l'}$ and $E_i^{l''}$ by $EB_i^{l''}$ also decreases the dimension by 1.

We edit the arrays $I_i^{l'}$, $Q_i^{l'}$, $I_i^{l'}$, and $Q_i^{l'}$ in the same way to form $IA_i^{l'}$, $QA_i^{l'}$, $IB_i^{l''}$, and $QB_i^{l''}$. As in case 1, when editing the orientation array, we modify the orientation of any portion of the skein whose direction was reversed. In the array, $EA_i^{l'}$, orientation is preserved, so no changes are necessary. In the array $EB_i^{l''}$ the entire component $E_i^{l''}$ of the skein is reversed, so we edit all the elements of $Q_i^{l''}$. For each element $q_{i,s}^l$, where $0 \leq s \leq j''-1$, we reverse the orientation number. If $q_{i,s}^l = 3$ (right across strand 1), we change it to $q_{i,s}^l = 4$ (left across strand 1). If $q_{i,s}^l = 5$ (left across strand 2), we change it to $q_{i,s}^l = 4$ (right across strand 2). If $q_{i,s}^l = 4$, it could be either the skein passes left across strand 1 or right across strand 2. In this case, we look at the corresponding element $e_{i,s}^l$. If $|e_{i,s}^l| = 1$, the skein passes left across strand 1, so we change the orientation number to $q_{i,s}^l = 3$ (right across strand 1). If $|e_{i,s}^l| = 2$, the skein passes right across strand 2, so we change the orientation number to $q_{i,s}^l = 5$ (left across strand 2).

In the same manner as in case 1, we modify the element u_i to produce the elements ua_i and ub_i by considering which of the remaining crossings are affected in the resolution of the crossing in this step. The entire skein $E_i^{l''}$ is reversed in $EB_i^{l''}$, and so any crossings included in this skein may experience a sign change. If both components of the crossing, the overcrossing $(0.r)$ and the undercrossing $(-0.r)$, are contained in $E_i^{l''}$, the sign on the crossing $0.r$ will remain the same. If only one component of any crossing is contained in E_i^l and the other is unaffected, the sign on the crossing will change. For each crossing $0.r$, if no element $|e_{i,s}^l| = 0.r$ in the interval $0 \leq s \leq j''-1$, the sign of crossing $0.r$ is unaffected. If there is one element for which $|e^l_{i,s}| = 0.r$ in the interval $0 \leq s \leq j'' - 1$, the sign of the crossing $0.r$ is reversed, as the orientation on only one component of the crossing was reversed. If there are two elements for which $|e^l_{i,s}| = 0.r$ in the interval $0 \leq s \leq j'' - 1$, the crossing is unaffected, as the orientation of both components of the crossing was reversed.

Thus the skein $K_i = [E_i, I_i, Q_i]$ is replaced by skeins $KA_i = [EA_i, IA_i, QA_i]$ and $KB_i = [EB_i, IB_i, QB_i]$. Hence, we replace the component $Z_i = [s_i, E_i, c_i, u_i, I_i, Q_i]$ by $ZA_i = [s_i + 1, EA_i, c_i, ua_i, IA_i, QA_i]$ and $Z_i = [s_i - 1, EB_i, c_i, uB_i, IB_i, QB_i]$.

4.14 Program Induce Algorithm

After resolving all existing crossings, we are left with skeins in the genus 2 handlebody which have no self-intersections. Such skeins may not be simple, and are certainly not guaranteed to be basis elements. To produce basis elements, we introduce a procedure from projective geometry by first making the following observation. All curves in the 2-punctured disk are basis elements in the skein module of the knot complement of a given knot. Viewing any braid with two non-intersecting strands (the complement of which is the genus 2 handlebody) from the top of the braid looking down yields a planar diagram which is the 2-punctured disk. The question that arises from this observation is this: what do skeins in the genus 2 handlebody look like when viewed from the top? More formally, how to we translate skeins in the genus 2-handlebody into skeins in the 2-punctured disk naturally embedded in the genus 2-handlebody? This sort of projection, from the genus 2 handlebody to the 2-punctured disk, produces a crossing each time the skein moves from crossing over strand 1 of the base braid to crossing under strand 2 of the base braid (or equivalently, crossing under strand 1 of the base braid to crossing over strand 2 of the base braid). That is, each time the skein moves from the "front" of the picture to the "back" of the picture in the side view, a crossing in the skein is introduced in the top view. Equipped with this knowledge, we propose a method of inducing crossings in the side view at the same locations as they would bewhen viewed from the top. Once these induced crossings have been resolved in every instance, the resulting skein will be a basis element of the skein module of the complement of the original knot.

A crossing in the top view occurs whenever the skein moves from the "front" of the diagram to the "back" of the diagram. Portions of the skein which are in "front" of the diagram are seen as overcrossings and portions of the skein which are in "back" of the diagram are seen as undercrossings in the side view. We will sort the crossings along strand 1, bringing overcrossings to lower indices and undercrossings to higher indices, exchanging the position of an overcrossing and an undercrossing on strand 1 by inducing a crossing between them. An example of such an exchange is shown in Figure 4.31. Crossing these two components of the skein induces two crossings, one on the left and one on the right, and the indices of the overcrossing and undercrossing are exchanged (the index of the overcrossing is decreased and the index of the undercrossing is increased). Whenever we cross components of the skein, we either cross two strands which are in the same component of the skein, or else cross two strands which are in different components. In the first case, if the overcrossing and undercrossing are consecutive in the same component, it is only necessary to induce a single crossing. In the second case, a crossing is introduced on each side of the overcrossing in the first component and on each side of the undercrossing in the second component. The orientation on the component(s) of the skein determines the signs on the induced crossings.

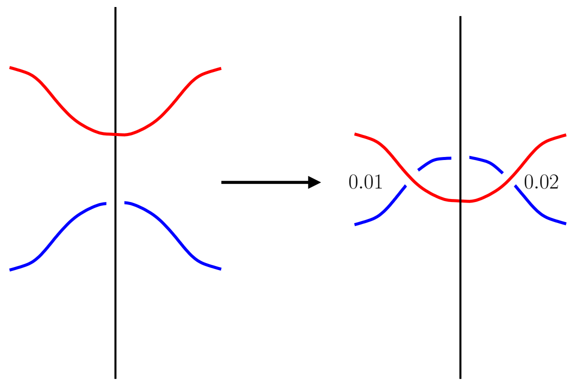


Figure 4.31. Crossing strands from different components produces two crossings, labeled from left to right, 0.01 and 0.02.

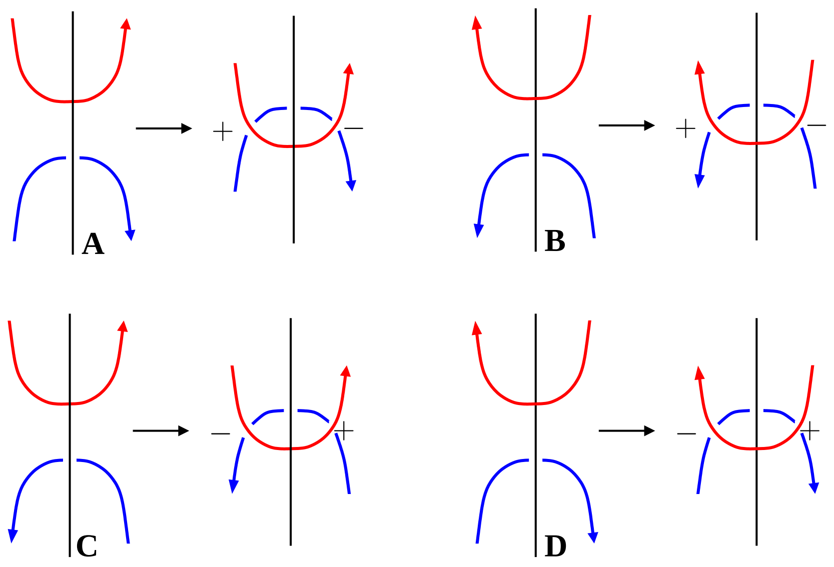


Figure 4.32. The orientations of the curves determine the signs of the crossings.

We implement a sorting method, moving overcrossings to lower indices and undercrossings to higher indices, on each strand of the base braid. Overcrossings and undercrossings are exchanged pairwise on each strand. Consider a skein $K = [E, I, Q]$. Starting with the first component of the skein and proceeding componentwise, we form an array called "$ONES$" (resp, "$TWOS$"), which contains the information about every entry on strand 1 (resp, 2). The procedure for inducing crossings on strand 2 is the same as for strand 1, so without loss of generality, consider strand 1. The array we produce on this strand is of the form:

$ONES = [[o1_1, o1_2, o1_3, o1_4], [o2_1, o2_2, o2_3, o2_4], ..., [ok_1, ok_2, ok_3, ok_4]]$

where the entries oi_j, j=1,2,3,4 contain all the information about the entry. The entries of $ONES$ are recorded as follows:

Entries in the array $ONES$ (strand 1)
$ONES = [[o1_1, o1_2, o1_3, o1_4], [o2_1, o2_2, o2_3, o2_4], ..., [ok_1, ok_2, ok_3, ok_4]]$
$ok_1 = +/-1$, the value of the e^l_j (overcrossing or undercrossing)
ok_2=index i^l_j of value
ok_3=component l of the array where this value can be found
ok_4=the index j within that component

For strand 2, we form a similar array, $TWOS$, with entires as follows:

Entries in the array $TWOS$ (strand 2)
$TWOS = [[t1_1, t1_2, t1_3, t1_4], [t2_1, t2_2, t2_3, t2_4], ..., [tk_1, tk_2, tk_3, tk_4]]$
$tk_1 = +/-2$, the value of the e^l_j (overcrossing or undercrossing)
tk_2=index i^l_j of value
tk_3=component l of the array where this value can be found
tk_4=the index j within that component

ok_3 and ok_4 (resp. tk_3 and tk_4) locate the entry within the arrays E, I, and Q containing the skein, index, and orientation information.

After forming the array $ONES$, we will split the index values, located at ok_1, into two arrays: P containing the index values ok_1 of $ONES$ such that $ok_0 = 1$ and N containing the index values ok_1 of $ONES$ such that $ok_0 = -1$. Next, we form an array B containing all the index values ok_1. By comparing the entries of P and N against the values of B, we determine where to induce the next crossing.

If the minimum element of B is the minimum element of N, we proceed to determine the elements of N and P that should be switched. Otherwise, while the minimum element of B is the minimum element of P, delete this element from both B and P, until either P is empty or the minimum element of B is the minimum element of N. If P is empty, no switch is required. If P is not empty, identify the maximum element of N. We compare this element to the minimum element remaining in P. If the maximum element of N is less than the minimum element of P, these two elements are the elements of P and N to be switched. Otherwise, delete the maximum element of N, and compare the new maximum element of N to the minimum element of P. Continue until the maximum element of N is less than the minimum element of P. These two elements, the maximum element remaining in N and the minimum element remaining in P, indicate the location at which we will induce a crossing (by switching the location of these elements).

We have now identified an element of P, corresponding to an element ok'_1 in $ONES$, and an element in N, corresponding to an element ok''_1 in $ONES$. We determine the location of the crossing to be induced as follows: let $l' = ok'_2$, $j' = ok'_3$, $l'' = ok''_2$, $j'' = ok''_3$. We are now faced with three cases: (1) the elements to be switched are in the same component and consecutive, (2) the elements to be switched are in the same component and nonconsecutive, and (3) the elements to be switched are in different components. The following table outlines where to insert a crossing in each case.

Inducing crossings in a skein
Case 1: elements in same component ($l' = l'' = l$), consecutive ($j' = j'' + 1$ or $j'' = j' + 1$)
If $j' < j''$, insert a crossing ± 0.99 at $e^l_{j''+1}$ and $e^l_{j'}$ (in this order)
If $j' > j''$, insert a crossing $\mp$).99 at $e^l_{j'+1}$ and $e^l_{j''}$ (in this order)
Case 2: elements in same component ($l' = l'' = l$), nonconsecutive
insert a crossing ± 0.01 at $e^l_{j'+1}$
insert a crossing ∓ 0.02 at $e^l_{j'+1}$
insert a crossing ∓ 0.01 at $e^l_{j''+1}$
insert a crossing ± 0.02 at $e^l_{j''}$
Case 3: elements in different components
insert a crossing ± 0.01 at $e^{l'}_{j'+1}$
insert a crossing ∓ 0.02 at $e^{l'}_{j'+1}$
insert a crossing ∓ 0.01 at $e^{l''}_{j''+1}$
insert a crossing ± 0.02 at $e^{l''}_{j''}$

In Case 1, we insert a single crossing. This crossing is resolved, producing 2 skeins. In Case 2 and Case 3, two crossings are inserted. These crossings are resolved, producing 4 skeins. We use the same algorithm as in Step 2 (Program Resolve) to resolve these crossings. Once resolved, we repeat this procedure on each skein. This procedure terminates when the array P formed from $ONES$ is empty (after cancellation with elements of B, as described above). At this point, the index of every overcrossing is lower than the index of any undercrossing, and the skein in the genus 2 handlebody is equivalent in planar representation to a skein in the 2-punctured disk. After sorting values in this manner, all resulting skeins, once reduced, will be one of the basis elements of the skein module of the genus 2 handlebody. Since this handlebody is embedded in the complement of a 2-bridge knot, the skeins produced will be basis elements of the Kauffman Bracket Skein Module of the knot complement.

Theorem 4.6. *The strand sorting method given in Algorithm 5 produces only basis elements x, y, z, and the unknot.*

Proof. The same method is applied to both strands of the base braid independently, so consider the elements along one strand of the base braid. The result of this strand sorting method is that overcrossings are moved to lower indices and undercrossings are moved to higher indices. This is accomplished by considering all elements pairwise along the strand. If the pair is a pair of overcrossings or a pair of undercrossings, no changes are made. If the pair is a pair of an overcrossing and an undercrossing, if the overcrossing has a higher index than the undercrossing, a crossing is induced between the two, switching their indices. If the overcrossing and undercrossing are consecutive in the same component, only one crossing is necessary. Otherwise, two crossings are induced. The crossing(s) are resolved, producing either 2 or 4 skeins in which the same pair either no longer exists (if the skein has been reduced at that pair) or else the indices of the pair have been switched such that the overcrossing has a lower index than the undercrossing. Continuing in this manner, all such pairs are switched until the following condition is satisfied: each skein in the resulting linear combination has the property that for all entries $e^l_j = 1$, and all entries $e^{l'}_{j'} = -1$, the corresponding indices satisfy the relation $i^l_j < i^{l'}_{j'}$. Such a skein in the genus 2 handlebody is actually a skein in the 2-punctured disk, and all non-intersecting skeins in the 2-punctured disk are either x, y, or z as shown in Figure 4.33. □

We note that in practice we treat the skeins y'_{pos} and y'_{neg}, introduced in the previous section, as basis elements. This is only for convenience in the computation, reducing the total number of steps required, hence reducing total computation time. Since these skeins are resolved independent of We use these variables as basis elements to simplify computations; rather than producing two additional terms at each encounter, which then requires the algorithm to be run separately on each term, we perform the resolution of all y'_{pos} and y'_{neg} elements after the algorithm terminates. Thus, the computation time is improved as the increase is only linear, rather than exponential.

This strand sorting method realizes all skeins in the genus 2 handlebody as skeins in the 2-punctured disk, which are basis elements shown in Figure 4.33.

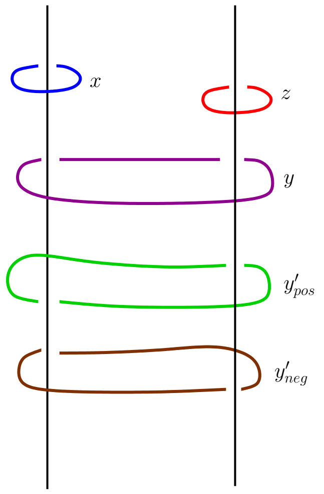


Figure 4.33. Basis elements.

4.15 Program Combine

The output of Program Induce, resulting from the strand sorting method of Theorem 4.6, is a linear combination of skeins which are basis elements of the Kauffman Bracket Skein Module of the Knot Complement of the given 2-bridge knot. Given this output, we identify the resulting skeins as one of the variables x, y, z, the unknot, or one of the simple skeins, y'_{pos} or y'_{neg}. These two simple skeins have been identified only for convenience and the reduction of computational complexity, and so we must replace them by their values in terms of x, y, and z.

4.15.1 Identifying Variables

We first reduce all remaining skeins using Algorithm 2. What remains are skeins of length 2 or 4. If the skein is of length 2, it is either x or z. If the skein is of length 4, it is either y, y'_{pos} or y'_{neg}. If the remaining skein is of length 0, it is the unknot.

Identifying Basis Elements and Simple Skeins
For $len(E^l) = 0$: $E^l =$unknot
For $len(E^l) = 2$: $E^l = [1, -1] = x$ $E^l = [-1, 1] = x$ $E^l = [2, -2] = z$ $E^l = [-2, 2] = z$
For $len(E^l) = 4$: $E^l = [1, 2, -2, -1] = y$ or any permutation of this array
$E^l = [1, -2, 2, -1] = y'_{pos}$ or $E^l = [1, -2, 2, -1] = y'_{neg}$ or any permutation of this array, whose value depends on the index of the entries

Recall the array $X = [x_0, x_1, x_2, x_3, x_4, x_5]$, where x_0 is the number of copies of the skein x, x_1 is the number of copies of the skein y, x_2 is the number of copies of the skein z, $x_3$is the number of copies of the skein y'_{pos}, x_4 is the number of copies of the unknot, and x_5 is the number of copies of the skein y'_{neg}.

For $len(E^l) = 0$, we increase x_4 by 1 and delete E^l from E.

For $len(E^l) = 2$, if $|e^l_0| = 1$, we increase x_0 by 1 and delete E^l from E. Otherwise, $|e^l_0| = 2$, and we increase x_2 by 1 and delete E^l from E. Note that all crossings have been resolved, and the computation was performed in a genus 2 handlebody, so $e^l_j = \pm 1$ or ± 2.

For $len(E^l) = 4$, if $e^l_0 = -e^l_1$ and $e^l_2 = -e^l_3$ and $\frac{e^l_1}{|e^l_1|} = \frac{e^l_2}{|e^l_2|}$, then we increase x_1 by 1. Similarly, if $e^l_0 = -e^l_3$ and $e^l_1 = -e^l_2$ and $\frac{e^l_0}{|e^l_0|} = \frac{e^l_1}{|e^l_1|}$, then we increase x_1 by 1.

Otherwise, E^l is one of the variables y'_{pos} or y'_{neg}. We determine which of the two by considering the elements of E^l such that $|e^l_j| = 1$. Take the element satisfying this condition with the highest index. If that element $e^l_j = 1$, we increase x_3 by 1. Otherwise, that element $e^l_j = -1$, and we increase x_5 by 1.

We replace the skeins y'_{pos} and y'_{neg} with their relations (in terms of x and y) given in the previous section. Finally, we use the relation for the unknot, replacing those skeins containing one or more copies of the unknot by the corresponding skeins. These relations are as follows:

y'_{pos}: y' with a positive twist $= -t^{-2}xz - t^{-4}y$

y'_{neg}: y' with a negative twist $= -t^4y - t^2xz$

unknot$=-t^2 - t^{-2}$

Note that in each case, a skein with one copy of these skeins is replaced by two skeins; the coefficients, powers of t, and variables are updated accordingly.

In the special case of the trefoil knot given in [5], higher powers of y can be written in terms of lower powers of y. Instances of futher relations, such as those shown below for the trefoil, are handled in a similar manner.

$y^2 = 1 + t^4 + t^2y - t^2x^2y - t^4x^2$

$y^3 = t^10 + t^6 - 3t^6x^2 + t^6x^4 + 2y + t^4y - 2t^4x^2y - t^4x^2y + t^4x^4y$

We replace the given skein by this linear combination, updating the coefficients, powers of t, and variables in each component of the array.

4.15.2 Cancellations

After implementing the algorithm in Step 3 (Program Induce), all skeins are written in terms of powers of x, y, and z. In program combine, we combine like terms by combining their coefficients, c_z.

CHAPTER 5
PYTHON PROGRAMS

In this chapter, we provide algorithms for each stage of the program. We state the input and output of each stage, and address the implementation of the methods described in Chapter 2. The final code, written in Python, is included in Appendix A.

5.1 Step 1: A Program to Project Curves

Program Project isotopes a skein in the complement of a knot past crossings in the base braid into a specified genus g handlebody embedded in the knot complement. As the curve moves past crossings in the base braid, new overcrossings and undercrossings are introduced, and the resulting skein may look more complicated.

5.1.1 The Projection Algorithm

The input of Algorithm 1 is a base braid B and a skein K. The required input information is as follows:

(1) n is the number of strands in the base braid;

(2) c is the number of crossings in the base braid;

(3) $B = [[b_0, ..., b_{c-1}], [v_0, ..., v_{c-1}]]$ is the base braid input array;

(4) skein: $K = [K^0, K^1, ..., K^{k-1}]$, where k in the number of components in the skein, with component $K^l = [S^l_0, S^l_1, ..., S^l_c]$ for $0 \leq l \leq k-1$, and section $S^l_d = [s^l_{d,0}, s^l_{d,1}, ..., s^l_{d,p-1}]$ for $0 \leq d \leq c$.

The output of Program Project is the skein, isotopic to the original skein, which lies entirely within a genus g handlebody, N.

```
Algorithm 1: Projecting a Skein in the Complement of the Base Knot
Input: $n$, $c$, $B$, $k$, $K$
Output: $E$
Data: Projecting a skein in the complement of the base knot to a genus g
      handlebody. The skein must pass through crossings of the base braid.
Initialization
Let $E = []$ start as an empty array and set $l = 0$
while $l < k$ do
    For each component, $K^l = [S^l_0, S^l_1, ..., S^l_c]$, set $d = 0$. while $i < c + 1$ do
        We consider section $S^l_i = [s^l_{i,0}, s^l_{i,1}, ..., s^l_{i,p-1}]$. We pass section $S^l_i$
          through crossing $b_i$, and append the answer to section $S^l_{i+1}$ Let $x = b_i$,
          $y = b_i + 1$, $n = len(S^l_i)$, and $t = 0$
        Exchange strand labels on all instances of x and y.
        while $t < n$ do
            if $s^l_{i,t} = x$ then
                $s^l_{i,t} = y$
            else if $s^l_{i,t} = -x$ then
                $s^l_{i,t} = -y$
            else if $s^l_{i,t} = y$ then
                $s^l_{i,t} = x$
            else if $s^l_{i,t} = -y$ then
                $s^l_{i,t} = -x$
            $t+ = 1$
    Let $t = 0$
    if the sign of $b_i$ is positive, $v_i > 0$ : then
        while $t < n$ and $|s^l_{i,t}| \neq x$ do
            $t+ = 1$
            $z = t$
        while $t < n$ and $|s^l_{i,t}| \neq y$ do
            $t- = 1$
        if $t < n$ then
            insert $-x$ at index $t + 1$ of $S^l_i$
            insert $y$ at index $t + 1$ of $S_i$
            $n+ = 2$
            $t = z + 3$
        while $t < n$ and $|s^l_{i,t}| \neq y$ do
            $t+ = 1$
        if $t < n$ then
```

5.2 Intermediate Step: Reducing Curves

Throughout the process of resolving crossings (Step 2: Program Resolve) and inducing crossings (Step 3: Program Induce) we can reduce the skein by canceling consecutive pairs of overcrossings or undercrossings. We determine if a pair can be cancelled by looking at the corresponding indices. If the indices are consecutive along the strand (not necessarily consecutive integers), then the pair can be canceled. We determine if a pair of indices is consecutive by comparing to all other indices along the strand.

5.2.1 The Reducing Algorithm

The input of Algorithm 2 is a skein $K = [E, I, Q]$:

(1) the skein intersection input $E = [E^0, ..., E^{k-1}]$, where $E^l = [e^l_0, e^l_1, ..., e^l_{(p_l)-1}]$ for each component $0 \leq l \leq k-1$;

(2) the skein index input $I = [I^0, ..., I^{k-1}]$, where $I^l = [i^l_0, i^l_1, ..., i^l_{(p_l)-1}]$ for each component $0 \leq l \leq k-1$;

(3) the skein orientation input $Q = [Q^0, ..., Q^{k-1}]$, where $Q^l = [q^l_0, q^l_1, ..., q^l_{(p_l)-1}]$ for each component $0 \leq l \leq k-1$.

The output is a reduced skein $K' = [E', I', Q']$, where $dim(E') = dim(I') = dim(Q') \leq dim(E) = dim(I) = dim(Q)$, such that E' has no consecutive pairs of overcrossings or undercrossings whose indices are consecutive along a strand (that is, there are no simplifications).

Algorithm 2: Reducing a skein

```
Input: A skein $K = [E, I, Q]$
Output: A reduced skein $K' = [E', I', Q']$, where
        $dim(E') = dim(I') = dim(Q') \leq dim(E) = dim(I) = dim(Q)$
Data: Reducing a skein by canceling pairs of overcrossings or undercrossings.
Initialization
Let $l = 0$
Let $D = []$ start out as an empty array while $l < k - 1$ do
    Set $j = 0$
    while $j < p_l$ do
        if $|e^l_j| = 1$ then
            append $i^l_j$ to $D$
        $j+ = 1$
Set $l = 0$ while $l < k - 1$ do
    Let $j = -1$ while $j < p_l - 1$ do
        if $|e^l_j|$ and $e^l_j = e^l_{j+1}$ then
            Set $r = 0$ (if indices are consecutive, this variable will remain at 0)
            Let $i^l_{min} = min(i^l_j, i^l_{j+1})$ Let $i^l_{max} = max(i^l_j, i^l_{j+1})$ Let
            $i^l_{mid} = i^l_{min} + 1$ while $i^l_{mid} < i^l_{max}$ do
                Let $d = 0$ while $d < len(D)$ do
                    if $D_d = i^l_j$ then
                        Set $r = 1$
                    $d+ = 1$
                $i^l_{mid}+ = 1$
            if $r = 0$ then
                delete $e^l_{j+1}$, $i^l_{j+1}$, and $q^l_{j+1}$
                delete $e^l_j$, $i^l_j$, and $q^l_j$
                $j- = 1$ Set $d = 0$ while $d < len(D)$ do
                    if $D_d = i^l_{min}$ or $D_d = i^l_{max}$ then
                        delete $D_d$
                    else
                        $d+ = 1$
            else
                j+=1
        else
            j+=1
    $l+ = 1$
```

5.3 Step 2: Program to Resolve Crossings

Program Resolve resolves crossings in a skein as a planar diagram in a genus 2 handlebody. The algorithm below resolves existing crossings by considering 2 cases: either the crossings are in the same component or the crossings are in different components. The algorithm is initially implemented on a single diagram, Algorithm 3, producing two diagrams. The resulting skeins are recorded in an array, $Z = [[\pm 1, E_1, 1000, U_1, I_1, Q_1], [\mp 1, E_2, 1000, U_2, I_2, Q_2]]$, each with a coefficient in t and information retained about the crossings left to be resolved (U_1 and U_2 may have different signs for the next crossing to be resolved). Indeed, as each crossing is resolved, this information is updated according to which part of the skein reversed directions as this affects the signs of some of the remaining crossings. After the first crossing is resolved, producing the array Z, we perform the same computation on each component of Z, Algorithm 4. We consider the next crossing, resolve that crossing in each component of Z, and replace that component with two new components, increasing the dimension of Z. The final array Z has dimension 2^r, where r is the number of crossings to be resolved.

5.3.1 The Resolving Algorithm

The input of Algorithm 3 is a skein K, with r self-intersections. The input variables are as follows:

(1) the skein intersection input $E = [E^0, ..., E^{k-1}]$, where $E^l = [e^l_0, e^l_1, ..., e^l_{(p_l)-1}]$ for each component $0 \leq l \leq k-1$;

(2) the skein index input $I = [I^0, ..., I^{k-1}]$, where $I^l = [i^l_0, i^l_1, ..., i^l_{(p_l)-1}]$ for each component $0 \leq l \leq k-1$;

(3) the skein orientation input $Q = [Q^0, ..., Q^{k-1}]$, where $Q^l = [q^l_0, q^l_1, ..., q^l_{(p_l)-1}]$ for each component $0 \leq l \leq k-1$;

(4) $S = [1 \times 10^{-k_r}, 2 \times 10^{-k_r}, ..., r \times 10^{-k_r}]$, the array containing the labels on the crossings to be resolved;

(5) $U = [\pm 1, \pm 1, ..., \pm 1]$, the array containing the signs of the crossings, with $dim(U) = r$.

The output of Program Resolve is 2^r skeins with coefficients in $\mathbb{Z}[t, t^{-1}]$. The skeins produced by resolution of all existing crossings are skeins in a genus 2 handlebody with no self-intersections when viewed as a planar diagram of the same form as the input diagram. The resulting linear combination of skeins in the variable t is not necessarily a linear combination of basis elements of the skein module, as not every non-intersecting skein in the genus 2 handlebody is a basis element of the skein module. We recover the linear combination of skeins from the output of Algorithm 4 as in Proposition 4.4.

```
Algorithm 3: Resolving One Crossing Using Skein Relations
Input: $E, I, Q, S, U$
Output: $Z$, an array containing two components,
        $Z = [[\pm 1, E_1, 1000, U_1, I_1, Q_1], [\mp 1, E_2, 1000, U_2, I_2, Q_2]]$
Data: Resolving crossing $S[s]$ in a skein in a genus 2 handlebody into two
      skeins using the Kauffman bracket skein relations.
Initialization
Let $a = []$ start as an empty array
Set $l = 0$
First, we locate the crossing to be resolved, $S[s]$ with sign $U[s]$
while $l < k$ do
    Let $j = 0$
    while $j < p_l$ do
        if $|e^l_j| = S[s]$ then
            append $l$ to $a$
            append $j$ to $a$
            $j+ = 1$
        else
            $j+ = 1$
    $l+ = 1$
Case 1: The overcrossing and undercrossing are in the same component of the
 curve $(l' = l'')$ if $a[0] = a[2]$ then
    Let $C = E$, $IC = I$, and $QC = Q$
    Let $A = []$, $IA = []$, and $QA = []$ start as empty arrays
    Let $j' = a[1]$ and $j'' = a[3]$
    while $j' < j'' - 1$ do
        append $C^l_{j'+1}$ to $A$
        append $IC^l_{j'+1}$ to $IA$
        append $QC^l_{j'+1}$ to $QA$
        remove $C^l_{j'+1}$ from $C$
        remove $IC^l_{j'+1}$ from $IC$
        remove $QC^l_{j'+1}$ from $QC$
        $j''- = 1$

    delete $C^l_{j'+1}$
    delete $C^l_{j'}$
    delete $IC^l_{j'+1}$
    delete $IC^l_{j'}$
    delete $QC^l$
```

The output of Algorithm 3, the array $Z = [[\pm 1, E_1, 1000, U_1, I_1, Q_1], [\mp 1, E_2, 1000, U_2, I_2, Q_2]]$, is used as the input of Algorithm 3.

Algorithm 4: Resolving All Crossings Using Skein Relations

```
Input: A curve, $Z$, with n crossings
Output: A curve, $Z$, with all crossings resolved
Data: Resolving exisiting crossings in a multicurve in a genus 2 handlebody
      in the complement of the base knot using the Kauffman Bracket Skein
      Relations.
Initialization
For all $s$ in $S$, we resolve the crossing $S[s]$ in each component $Z[z]$ as follows:
Fix $S[s]$ and let $z = 0$
while $z < len(Z)$ do
    Let $E = Z_z^1$
    Let $I = Z_z^4$
    Let $Q = Z_z^5$
    Let $U = Z_z^3$
    Implement ?? for skein $K = [E, I, Q]$ in $Z[z]$ $Z[z]$ is resolved into two
      components inserted at $Z[z+1]$ and $Z[z+2]$ Set $Z_{z+1}^0 = Z_{z+1}^0 + Z_z^0$
    Set $Z_{z+2}^0 = Z_{z+2}^0 + Z_z^0$
    Set $Z_{z+1}^2 = (Z_{z+1}^2 * Z_z^2)/1000$
    Set $Z_{z+2}^2 = (Z_{z+2}^2 * Z_z^2)/1000$
    delete $Z_z$
    $z+ = 1$
```

5.4 Step 3: Program to Induce Crossings

The output of Algorithm 4 is a linear combination of skeins in the genus 2 handlebody with coefficients in t. Such skeins may be complicated, and are not guaranteed to be basis elements of the skein module of the knot complement of the original diagram. To produce basis elements, we introduce crossings as described in Chapter 4. Locating such crossings, we form an array S_1 on strand 1 and S_2 on strand 2, Algorithm 5. Ordering these arrays as described in Chapter 4 introduces new crossings. In one case, a single crossing is induced, and this crossing is resolved by Algorithm 3. In all other cases, two crossings are induced, and Algorithm 4 is implemented. The result of Algorithm 5 is an array Z where each component is skein that is a basis element of the skein module described in Chapter 3, with additional information pertaining to its coefficient.

5.4.1 The Induce Crossings Algorithm

The input of Algorithm 5 is an array

$$Z = [[s_0, E_0, c_0, u_0, I_0, Q_0], [s_1, E_1, c_1, u_1, I_1, Q_1], \\ ..., [s_{2^r-1}, E_{2^r-1}, c_{2^r-1}, u_{2^r-1}, I_{2^r-1}, Q_{2^r-1}]]$$

where r is the number of crossings resolved in the previous stage (Step 2: Program Resolve). As in the input to Algorithm 3, each E_z, I_z, and Q_z are of the form:

(1) $s_0 \in \mathbb{Z}$ is the power of the variable t;

(2) the skein intersection input $E_z = [E_z^0, ..., E_z^{k-1}]$, where $E^l = [e_0^l, e_1^l, ..., e_{(p_l)-1}^l]$ for each component $0 \leq l \leq k-1$;

(3) $c_z = \pm c \times 10^3$, where $c \in \mathbb{N}$, which is the coefficient in front of the power of t;

(4) $U_z = \emptyset$, as all crossings were resolved at the previous stage;

(5) The skein index input $I_z = [I_z^0, ..., I_z^{k-1}]$, where $I^l = [i_0^l, i_1^l, ..., i_{(p_l)-1}^l]$ for each component $0 \leq l \leq k-1$;

(6) The skein orientation input $Q_z = [Q_z^0, ..., Q_z^{k-1}]$, where $Q^l = [q_0^l, q_1^l, ..., q_{(p_l)-1}^l]$ for each component $0 \leq l \leq k-1$.

To each component, we append the array $X_z = [0, 0, 0, 0, 0, 0]$, to store skeins which have already been reduced to basis elements. The form of this array is as follows:

(1) X_0: the number of parallel, independent copies of the skein x;

(2) X_1: the number of parallel, independent copies of the skein y;

(3) X_2: the number of parallel, independent copies of the skein z;

(4) X_3: the number of parallel, independent copies of the skein y'_{pos};

(5) X_4: the number of copies of the skein unknot;

(6) X_5: the number of parallel, independent copies of the skein y'_{neg}.

The output of Algorithm 5 is an array Z such that in each component z, the skein E_z is made up of only basis elements.

Algorithm 5: Inducing Crossings to Resolve a Curve into Basis Elements

```
Input: Linear combination of skeins, $Z = [[], [], ..., []]$, with coefficients in
       powers of $t$
Output: Linear combination of skeins $x$, $y$, $z$, $y'_{pos}$, $y'_{neg}$, and the unknot, with
        coefficients in powers of $t$
Data: Inducing crossings in a skein in a genus 2 handlebody to realize the
      skein as a linaer combination of basis elements of the skein module.
Initialization
Let $z = 0$
while $z < len(Z)$ do
    Consider component $Z_z = [s_z, E_z, c_z, u_z, I_z, Q_z]$
    Implement algorithm 2 with input skein $K = [E_z, I_z, Q_z]$
    Let $S = []$ start out as an empty array
    Let $l = 0$
    while $l < k$ do
        Consider $E_z^l = E^l$ $j = 0$
        while $l < k$ $and$ $j < p_l$ do
            if $|e_j^l| = 1$ then
                Append the array $[e_j^l, i_j^l, l, j]$ to $S$
            j+=1
        l+=1
    Let $P = []$, $N = []$, and $B = []$ start out as empty arrays
    Let $s = 1$
    while $s < len(S)$ do
        Append $S_1^s$ to $B$
        if $S_1^s = 1$ then
            Append $S_1^s$ to $P$
        else
            Append $S_1^s$ to $N$
        $s+ = 1$
    if $len(P) > 0$ then
        Let $p_{min} = min(P)$ and $p_{max} = max(P)$
    if $len(N) > 0$ then
        Let $n_{min} = min(N)$ and $n_{max} = max(N)$
    if $len(B) > 0$ then
        Let $b_{min} = min(B)$ and $b_{max} = max(B)$
    while $len(P) > 0$ do
        Let $r = 0$
        while $len(P) > 0$ $and$ $b_{min} = p_{min}$ $and$ do
```

5.5 Post-Processing: Program Combine

The output of Algorithm**??** is an array Z such that in each component z, the skein E_z is identifiable as one or several copies of the elements of X_z. It may be that the components of the skein must be reduced first (using Algorithm 2) before they can be identified. It is worth noting that this identification can be performed at each stage to reduce computational complexity. We state is here, as well as outline how to combine like terms in the final result. In most computations, shown in Chapter 6, the result has a drastic number of cancellations, leading to a final polynomial with few terms.

CHAPTER 6
APPLICATIONS

As proof of concept, we use the methods presented in this dissertation and the resulting program to recover some previous results of R. Gelca, J. Sain, and H. Wang.

6.1 Example 1: Computing powers of y

In [5], Gelca produced a relation for y^2 in terms of y in the complement of the trefoil knot by setting the top diagram equal to the botton in Figure 6.1.

$y^2 = t^4 - t^4x^2 + t^2y - t^2x^2y + 1$

The last term in the bottom relation, however, was resolved with some difficulty, as it becomes necessary to isotope the curve first, introduce a crossing in a specific location, and resolve that crossing.

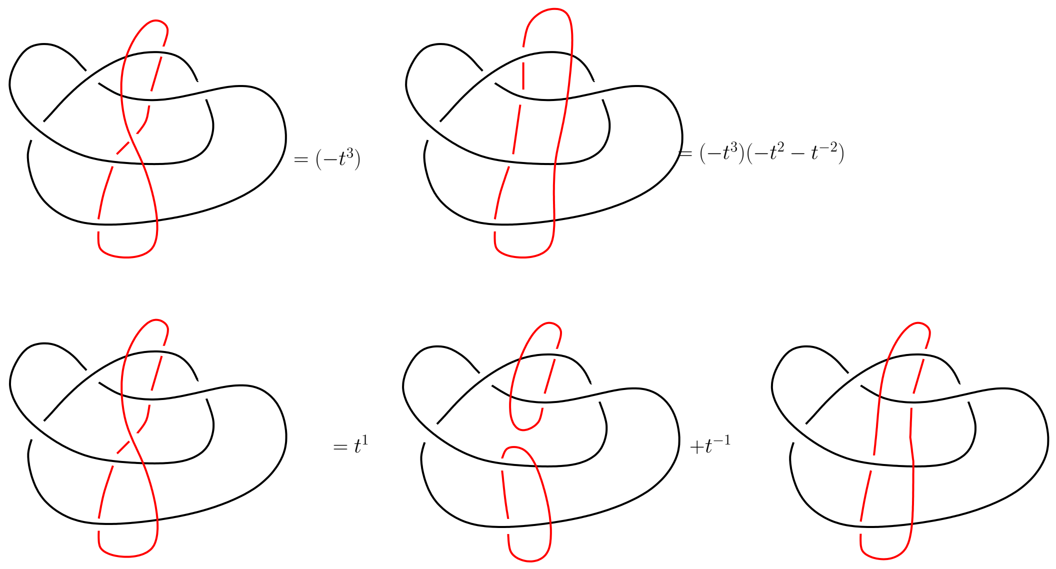


Figure 6.1. Resolving these curves produces the value of y^2

Similarly, Gelca found a relation for y^3 in terms of y by resolving the curve shown in Figure 6.2 with $n = 1$, and substituting the relation for y^2.

$y^3 = t^{10} + t^6 - 3t^6x^2 + t^6x^4 + 2y + t^4y - 2t^4x^2y - t^4x^2y + t^4x^4y$

These were the only relations necessary for the trefoil, as the highest power we encounter in any computation y^3. However, extending to a broader family of knots, it becomes necessary to compute higher powers of y. We can reproduce the results of these computations, both for y^2 and y^3 as well as arbitrary higher powers of y by the skein shown in Figure 6.2. We use Program Project to project the skein to the left of the diagram and we isotope the curve along the top of the diagram to the right side. Setting the two results equal to each other produces the desired relation. In the case of y^2, we use the diagram in Figure 6.2 with $n = 0$. Since the skein has no self-intersections, Step 2 (Program Resolve) is not necessary, and so we apply Algorithm 5 in Step 3 (Program Induce) to the skein on each side.

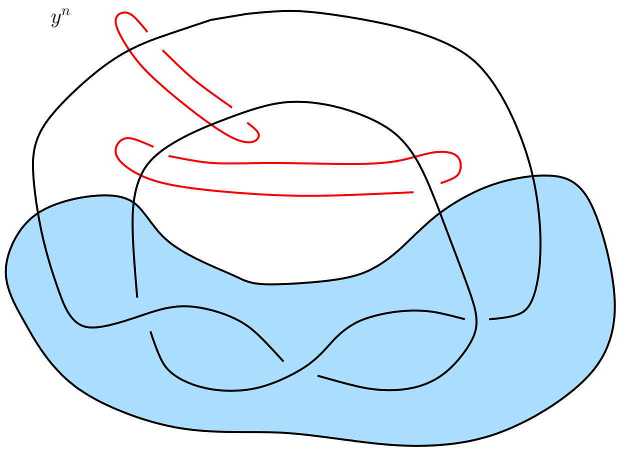


Figure 6.2. Resolving these curves produces the value of y^n

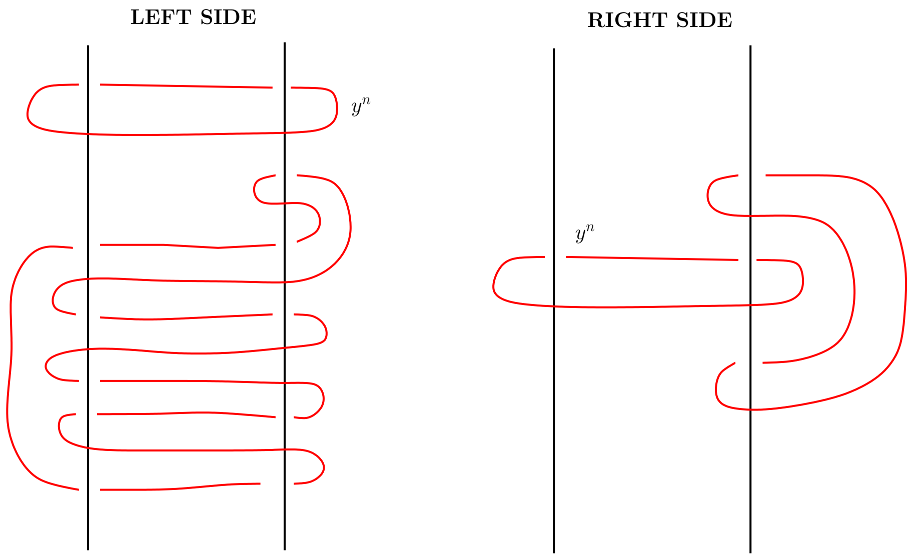


Figure 6.3. Curves pushed to the left and right side of the diagram

The input arrays are as follows.

The input for the left diagram is:

$E = [[-1.0, -2.0, 2.0, 1.0, -1.0, -2.0, 2.0, -1.0, 1.0, 2.0, -2.0, -1.0, 1.0, 2.0, -2.0, 2.0, -2.0, -1.0]]$

$I = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 15, 14, 17]]$

$Q = [[3, 4, 5, 4, 3, 4, 5, 4, 3, 4, 5, 4, 3, 4, 5, 4, 5, 4]]$

and so the array Z is of the form:

$Z = [[0, E, 1000, [1], I, Q]]$

The input for the right diagram is:

$E = [[2, -2, 2, -2]]$

$I = [[0, 3, 2, 1]]$

$Q = [[4, 5, 4, 5]]$

and so the array Z is of the form:

$Z = [[0, E, 1000, [1], I, Q]]$

Additionally, we quickly compute relations for y^4 and y^5 in terms of y using input given by Figure 6.2. The input for the relation for y^4 (Figure 6.2 with $n = 2$) is:

The input for the left diagram is:

$E = [[-1, -2, 2, 1, -1, -2, 2, -1, 1, 2, -2, -1, 1, 2, -2, 2, -2, -1], [1, 2, -2, -1]]$

$I = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 15, 14, 17], [18, 19, 20, 21]]$

$Q = [[3, 4, 5, 4, 3, 4, 5, 4, 3, 4, 5, 4, 3, 4, 5, 4, 5, 4], [3, 4, 5, 4]]$

and so the array Z is of the form:

$Z = [[0, E, 1000, [1], I, Q]]$

The input for the right diagram is:

$E = [[2, -2, 2, -2], [1, 2, -2, -1]]$

$I = [[1, 8, 7, 2], [5, 3, 4, 6]]$

$Q = [[4, 5, 4, 5], [3, 4, 5, 4]]$

and so the array Z is of the form:

$Z = [[0, E, 1000, [1], I, Q]]$

which produces:

$y^4 = t^{14} - t^8x^6 + 5t^8x^4 - 6t^8x^2 + t^8 - t^6x^6y + 5t^6x^4y - 6t^6x^2y + t^6y - 3t^4x^2 + 3t^4 - 3t^2x^2y + 3t^2y + 2$

The input for the left diagram is:

$E = [[-1, -2, 2, 1, -1, -2, 2, -1, 1, 2, -2, -1, 1, 2, -2, 2, -2, -1], [1, 2, -2, -1], [1, 2, -2, -1]]$

$I = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 16, 15, 14, 17], [18, 19, 20, 21], [22, 23, 24, 25]]$

$Q = [[3, 4, 5, 4, 3, 4, 5, 4, 3, 4, 5, 4, 3, 4, 5, 4, 5, 4], [3, 4, 5, 4], [3, 4, 5, 4]]$

and so the array Z is of the form:

$Z = [[0, E, 1000, [1], I, Q]]$

The input for the right diagram is:

$E = [[2, -2, 2, -2], [1, 2, -2, -1], [1, 2, -2, -1]]$

$I = [[1, 12, 11, 2], [5, 3, 4, 6], [9, 7, 8, 10]]$

$Q = [[4, 5, 4, 5], [3, 4, 5, 4], [3, 4, 5, 4]]$

and so the array Z is of the form:

$Z = [[0, E, 1000, [1], I, Q]]$

which produces:

$y^5 = -t^{22}x^2 + t^{22} - t^{20}x^2y + t^{20}y + t^{10}x^8 - 7t^{10}x^6 + 15t^{10}x^4 - 10t^{10}x^2 + 5t^{10} + t^8x^8y - 2t^8x^6y + 15t^8x^4y + 10t^8x^2y + t^8y + 4t^6x^4 + 12t^6x^2 + 4t^6 + 4t^4x^4y - 12t^4x^2y + 4t^4y + 5y$

We can use this information to investigate how higher powers of y can be written in terms of lower powers of y in the skein module of the complement of the trefoil knot. Studying the next several terms in this sequence may lead to a very educated guess at the formula for y^n.

6.2 Example 2: The (1,0)-curve

In Theorem 1 of [5], Gelca computed the action of the skein algebra of the cylinder over the boundary of the trefoil knot on the skein module of the knot complement by determining the action on the family of (p, q)-curves. In order to perform induction on the values p and q, it was necessary to first compute this action on the $(1, 0)$-curve (Lemma 3 of [5]). We reproduce this result using the methods presented in this dissertation.

The action on the $(1, 0)$-curve is determined by considering a skein which goes one time around the longitude of the knot and zero times around the meridian. Note that we must introduce three twists to cancel the effect of the curve passing the three crossings in the knot so that the skein has a twisting number 0 with respect to the base knot. We push this skein from the boundary of the knot in to the knot complement; resolving this skein in the knot complement gives the action of the skein algebra of the boundary on the skein module of the knot complement. We identify a handlebody in which the knot appears as a braid, shown in Figure 6.4.

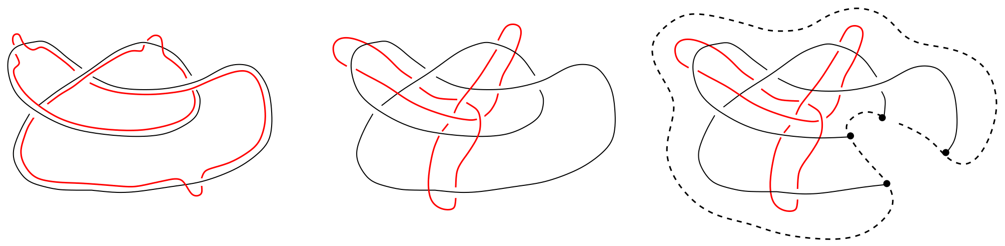


Figure 6.4. The skein is pushed from the boundary into the knot complement

Figure 6.5 shows the input for Progrom Project. We format the curve so that each component flows from the bottom of the diagram to the top of the diagram, passing once through each section of the diagram, and consider the skein in each section of the base braid.

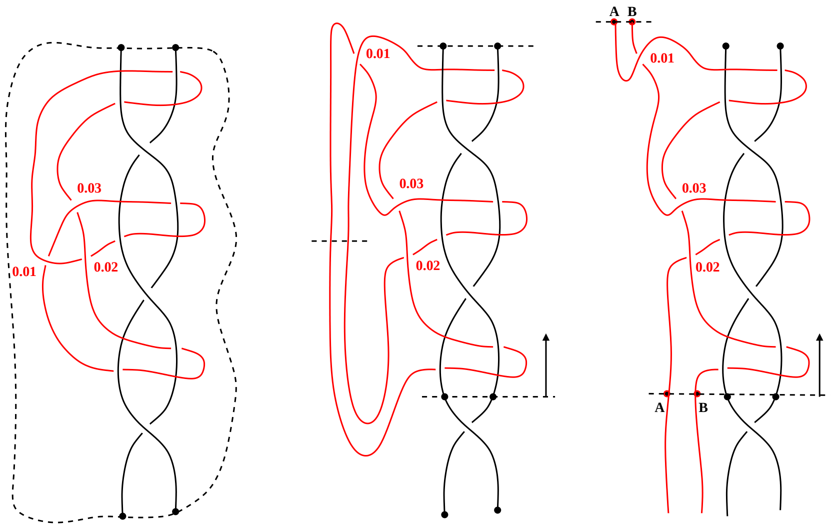


Figure 6.5. Implement Program Project

The input information for the base braid and the skein in each section of the base braid is as follows:

The base braid has 2 strands, so $n = 2$. The crossings are between strands 1 and 2, so we record a "1" for each of the two crossings. Strand 1 crosses over strand 2 in both cases, so we record a "1" for the sign of each crossing. Thus, $B = [[1, 1], [1, 1]]$.

Next, we record the skein section array. The two crossings section the base braid into three sections. For the three crossings in the diagram, we let the crossing with lowest index (0.02) be a part of the bottom section, the crossing 0.03 be part of the middle section, and the crossing 0.01 be part of the top section. For each of the two components shown in Figure 6.5, we form an array: $K^A = [[-0.02], [-1, 2, -2, 1, 0.03], [-0.01]]$ and $K^B = [[-1, 2, -2, 1, 0.02], [-0.03], [-1, 2, -2, 1, 0.01]]$. The array $K = [K^A, K^B]$ in the input for Algorithm 1. The result of this algorithm, after applying the connecting information, is the skein E shown in Figure 6.6 and given by the array:

$E = [[-1, -2, 2, 1, -1, 2, -2, 2, -2, -1, 0.02, -0.03, -1, 2, -2, 1, 0.01, -0.02, -1, -2, 2, -1, 1, -1,$

We assign index values and orientation values (according to the conventions described in Chapter 4):

$I = [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 92, 93, 16, 17, 18, 19, 91, 92, 10, 11, 12, 13, 14, 15, 93, 91]]$

$Q = [[3, 4, 5, 4, 3, 4, 5, 4, 5, 4, 92, 93, 3, 4, 5, 4, 91, 92, 3, 4, 5, 4, 3, 4, 93, 91]]$

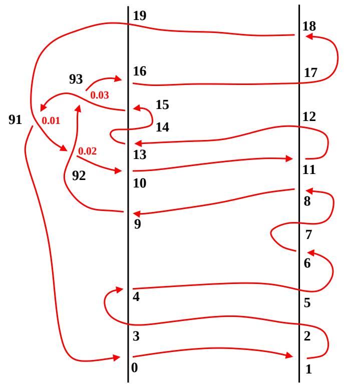


Figure 6.6. Implement Programs Resolve and Induce

We apply Algorithm 3 and Algorithm 4 to the skein $K = [E, I, Q]$ with input array $U = [-1, -1, -1]$ to store the signs of crossings 0.01, 0.02, 0.03. The resolution of these three crossings produces $2^3 = 8$ terms. The output of Program Resolve is an array:

$Z = [[3, [[-1, -2, 2, 1, -1, 2, -2, 2, -2, -1], [-1, 1, -1, 2, -2, -1], [-1, 2, -2, 1]], 1000, [-1, 1, 1], [[0$

corresponding to the following 8 terms:

term $1:\ 1\dot{t}^{(}3)[[-1, -2, 2, 1, -1, 2, -2, 2, -2, -1], [-1, 1, -1, 2, -2, -1], [-1, 2, -2, 1]]$

with index array $[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [15, 14, 13, 12, 11, 10], [16, 17, 18, 19]]$

and orientation array $[[3, 4, 5, 4, 3, 4, 5, 4, 5, 4], [3, 4, 3, 4, 5, 4], [3, 4, 5, 4]]$

term $2:\ 1{*}t^{(}1){*}[[-1, -2, 2, 1, -1, 2, -2, 2, -2, -1], [1, -2, 2, -1, -1, 1, -1, 2, -2, -1]]$

with index array $[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], [19, 18, 17, 16, 15, 14, 13, 12, 11, 10]]$

and orientation array $[[3, 4, 5, 4, 3, 4, 5, 4, 5, 4], [3, 4, 5, 4, 3, 4, 3, 4, 5, 4]]$

term $3:\ 1{*}t^{(}1){*}[[-1, -2, 2, 1, -1, 2, -2, 2, -2, -1, -1, -2, 2, -1, 1, -1], [1, -2, 2, -1]]$

with index array $[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15], [19, 18, 17, 16]]$

and orientation array $[[3, 4, 5, 4, 3, 4, 5, 4, 5, 4, 3, 4, 5, 4, 3, 4], [3, 4, 5, 4]]$

term $4:\ 1{*}t^{(}-1){*}[[-1, -2, 2, 1, -1, 2, -2, 2, -2, -1, -1, -2, 2, -1, 1, -1, -1, 2, -2, 1]]$

with index array $[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]]$

and orientation array $[[3, 4, 5, 4, 3, 4, 5, 4, 5, 4, 3, 4, 5, 4, 3, 4, 3, 4, 5, 4]]$

term $5:\ 1{*}t^{(}1){*}[[-1, -2, 2, 1, -1, 2, -2, 2, -2, -1, -1, 2, -2, 1], [-1, 1, -1, 2, -2, -1]]$

with index array $[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 16, 17, 18, 19], [15, 14, 13, 12, 11, 10]]$

and orientation array $[[3, 4, 5, 4, 3, 4, 5, 4, 5, 4, 3, 4, 5, 4], [3, 4, 3, 4, 5, 4]]$

term $6:\ 1{*}t^{(}-1){*}[[-1, -2, 2, 1, -1, 2, -2, 2, -2, -1, -1, -2, 2, -1, 1, -1, -1, 2, -2, 1]]$

with index array $[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]]$

and orientation array $[[3,4,5,4,3,4,5,4,5,4,3,4,5,4,3,4,3,4,5,4]]$

term $7: 1*t^{(}-1)*[[-1,-2,2,1,-1,2,-2,2,-2,-1,-1,-2,2,-1,1,-1,-1,2,-2,1]]$

with index array $[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19]]$

and orientation array $[[3,4,5,4,3,4,5,4,5,4,3,4,5,4,3,4,3,4,5,4]]$

term $8: 1*t^{(}-3)*[[-1,-2,2,1,-1,2,-2,2,-2,-1,-1,-2,2,-1,1,-1,-1,2,-2,1],[]]$

with index array $[[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19],[]]$

and orientation array $[[3,4,5,4,3,4,5,4,5,4,3,4,5,4,3,4,3,4,5,4],[]]$

We use this array as the input for Algorithm 5. The application of this algorithm, realizing each of the 8 skeins as a linear of basis elements of the skein module of the complement, produces 1257 terms. After cancellations, we are left with an array corresponding to the following 9 terms:

$Z = [[-3,[],-1000,[1],[],[],[6,0,0,0,0,0]],[-3,[],5000,[1],[],[],[4,0,0,0,0,0]],[-5,[],-1000,[1],$

term 1 : $-1*t^{(}-3)*x^6*y^0$

term 2 : $5*t^{(}-3)*x^4*y^0$

term 3 : $-1*t^{(}-5)*x^4*y^1$

term 4 : $3*t^{(}-5)*x^2*y^1$

term 5 : $-6*t^{(}-3)*x^2*y^0$

term 6 : $1*t^{(}-9)*x^0*y^1$

term 7 : $-1*t^{(}-5)*x^0*y^1$

term 8 : $1*t^{(}-3)*x^0*y^0$

term 9 : $1*t^{(}-7)*x^0*y^0$

This is the polynomial:

$-t^{-3}x^6+5t^{-3}x^4-t^{-5}x^4y+3t^{-5}x^2y-6t^{-3}x^2+t^{-9}y-t^{-5}y+t^{-3}+t^{-7}$

Multiplying by the framing factor $-t^9$ (as in [5]), and grouping terms, we obtain:

$t^6[x^6-5x^4+6x^2-1]+t^4[x^4-3x^2+1]-t^2-y$

which matches the result of Lemma 3 in [5].

CHAPTER 7
FUTURE DIRECTION

Step 1 of the Program (Project) is written in full generality, projecting curves in a handlebody of genus g for any value g. Step 2 (Resolve) and Step 3 (Induce) are written explicitly for computations in the genus 2 handlebody, and can be applied to a large family of knots (as many other families of knots include computations performed in a genus 2 handlebody). In fact, for computations in the complement of any knot, many times the curves of interest can be projected into a genus 2 handlebody. Step 2 and Step 3 of this program can be modified to perform computations all handlebodies of genus g, but the algorithms need to be updated to include multiple strands. The same algorithm is applied separately to each strand. Hence we could expand these computations to other families of knots.

The skein theory for (2,2p+1)-torus knots, as well as 2-bridge knots in general, is well understood, and so examples produced in these settings are useful and enlightening. A better understanding of the skein theory for skein computations in the complement of arbitrary knots is required before such methods could be applied in a more general setting. Step 2 and Step 3 of the program given in this dissertation apply to a genus 2 handlebody. While the given methods could certainly be applied to a genus g handlebody (inducing crossings along each strand is the manner specified), the resulting skeins would not necessarily be basis elements for a genus g handlebody when $g > 2$.

7.1 Alternate Skein Relations

On a brighter note, the algorithms given here can be easily modified to accommodate a different set of skein relations. Performing computations for the Jones Polynomial (rather than the Kauffman Bracket) is only a matter of changing a few signs in Algorithms 3, 4, and 5. Note that the signs of the crossings are retained and modified in the array u_z of each term. Given the Jones polynomial skein relations, which depend on the sign of the crossing, we can apply the same algorithm with few changes. Further, the algorithm actually compute skeins in a genus 2 handlebody, which is a submanifold of the knot complement. In this context, the variables x and

z are identified. These variables are kept separate throughout the process in the event that computations are performed in a genus 2 handlebody which globally does not identify x and z. Maintaining the generality of this procedure in every possible instance will promote usage of these methods in a more general context.

7.2 Conclusion

In this dissertation, we have outlined a method for the automation of skein computations that is accurate, effective, and efficient. We recovered results previously obtained over months of work in a fraction of the time, reducing computation time to the few minutes required to prepare the skein for input. The methods outlined in this paper can be modified to accommodate a broader family of curves, but only once more information about the skein theory of larger families of knots is well known. In the meantime, the family of 2-bridge knots, for which the algorithms in this dissertation apply, will provide an extensive library of examples. Such examples will promote the testing of conjectures and the discovery of new relations sufficient to occupy the next several years of work.

```
#STEP 1: PROGRAM PROJECT

#USER INPUTS
#m = the number of crossings in the base knot
#X= the base braid array
#e = the number of components in the skein
#V = the skein input array

INPUT: m, X, e, V

v = 0
while v < e:
    K = []
    k = 0
    print('Projected Knot Component', v + 1)
    while k < m + 1:
        L = []
        l = 0
        while l != 100:
            l = float(input('Enter crossing:'))
            L.append(l)
        del L[-1]
        K.append(L)
        k += 1
    k = 0
    l = 0
    V.append(K)
    v += 1
print(V)
E=[]
v = 0
while v < e:
    K = V[v]
    i = 0
    while i < m:
        x = X[i]
        y = X[i] + 1
        r = R[i]

        n = len(K[i])
        t = 0
        while t < n:
            if K[i][t] == x:
                K[i][t] = y
            elif K[i][t] == -x:
                K[i][t] = -y
            elif K[i][t] == y:
                K[i][t] = x
            elif K[i][t] == -y:
                K[i][t] = -x
            t += 1
        t = 0

        if r > 0:
            t = 0
            while t < n:
                while t < n and abs(K[i][t]) != x:
                    t += 1
                    z = t
                while t < n and abs(K[i][t]) != y:
                    t -= 1
                if t < n:
```

```
                K[i].insert(t + 1, -x)
                K[i].insert(t + 1, y)
                n += 2
                t = z + 3
            while t < n and abs(K[i][t]) != y:
                t += 1
            if t < n:
                K[i].insert(t, y)
                K[i].insert(t, -x)
                n += 2
                t += 3
    if r < 0:
        t = 0
        while t < n:
            while t < n and abs(K[i][t]) != x:
                t += 1
                z = t
            while t < n and abs(K[i][t]) != y:
                t -= 1
            if t < n:
                K[i].insert(t + 1, x)
                K[i].insert(t + 1, -y)
                n += 2
                t = z + 3
            while t < n and abs(K[i][t]) != y:
                t += 1
            if t < n:
                K[i].insert(t, -y)
                K[i].insert(t, x)
                n += 2
                t += 3
    t = 0
    if r > 0:
        while t < n:
            while t < n and abs(K[i][t]) != y + 1:
                t += 1
            if t < n:
                K[i].insert(t, y)
                n += 1
                t += 2
            while t < n and abs(K[i][t]) == y + 1:
                t += 1
            if t < n:
                K[i].insert(t, y)
                n += 1
                t += 2
    if r < 0:
        while t < n:
            while t < n and abs(K[i][t]) != y + 1:
                t += 1
            if t < n:
                K[i].insert(t, -y)
                n += 1
                t += 2
            while t < n and abs(K[i][t]) == y + 1:
                t += 1
            if t < n:
                K[i].insert(t, -y)
                n += 1
                t += 2
    s = 0
    while s < n and abs(K[i][s]) != y:
            s += 1
        q = n - 1
        while q > 0 and abs(K[i][q]) != y:
            q -= 1
        t = 0
        if r > 0:
            while t < n:
                while t < n and abs(K[i][t]) != x - 1:
                    t += 1
                if t > s and t < n:
                    K[i].insert(t, -x)
                    n += 1
                    t += 2
                while t < n and abs(K[i][t]) == x - 1:
                    t += 1
                if t < q and t < n - 1:
                    K[i].insert(t, -x)
                    n += 1
                    t += 2
        if r < 0:
            while t < n:
                while t < n and abs(K[i][t]) != x - 1:
                    t += 1
                if t > s and t < n:
                    K[i].insert(t, x)
                    n += 1
                    t += 2
                while t < n and abs(K[i][t]) == x - 1:
                    t += 1
                if t < q and t < n - 1:
                    K[i].insert(t, x)
                    n += 1
                    t += 2
        if x == 1 and n != 0:
            if r > 0:
                f = 0
                while f < n and abs(K[i][f]) != y:
                    f += 1
                K[i].insert(f, -x)
                n += 1
                f = n - 1
                while f < n and abs(K[i][f]) != y:
                    f -= 1
                K[i].insert(f + 1, -x)
                n += 1
            if r < 0:
                f = 0
                while f < n and abs(K[i][f]) != y:
                    f += 1
                K[i].insert(f, x)
                n += 1
                f = n - 1
                while f < n and abs(K[i][f]) != y:
                    f -= 1
                K[i].insert(f + 1, x)
                n += 1
        K[i + 1] = K[i] + K[i + 1]
        i += 1
    E.append(K[i])
    v += 1

OUTPUT: E
```

```
#STEP 2: PROGRAM RESOLVE

#USER INPUTS
#E = the intersection array
#I= the index array
#Q = the orientation array
#S = the array containing the crossings
#U = the array containing the signs of the crossings

INPUTS: E, I, Q, S, U


import copy
def copy_2d(p):
    return copy.deepcopy(p)

u=0
s=0.01
e=0
a=[]


while e<len(E):
    f=0
    while f<len(E[e]):
        if abs(E[e][f])==s:
            a.append(e)
            a.append(f)
            f+=1
        else:
            f+=1
    e+=1

if a[0]==a[2]:

    C=copy_2d(E)
    IC=copy_2d(I)
    QC=copy_2d(Q)
    A=[]
    IA=[]
    QA=[]

    e=a[1]
    f=a[3]
    while e<f-1:
        A.append(C[a[0]][e+1])
        IA.append(IC[a[0]][e+1])
        QA.append(QC[a[0]][e+1])
        C[a[0]].pop(e+1)
        IC[a[0]].pop(e+1)
        QC[a[0]].pop(e+1)
        f-=1

    C[a[0]].pop(a[1])
    C[a[0]].pop(a[1])
    O.append(IC[a[0]][a[1]])
    IC[a[0]].pop(a[1])
    IC[a[0]].pop(a[1])
    QC[a[0]].pop(a[1])
    QC[a[0]].pop(a[1])
    D=copy_2d(C)
    ID=copy_2d(IC)
    QD=copy_2d(QC)

    C.append(A)
    IC.append(IA)
    QC.append(QA)

    i=0
    while i<len(A):
        D[a[0]].insert(a[1],A[i])
        ID[a[0]].insert(a[1],IA[i])
        if QA[i]==3:
            QD[a[0]].insert(a[1],QA[i]+1)
        elif QA[i]==5:
            QD[a[0]].insert(a[1],QA[i]-1)
        else:
            if abs(A[i])==1:
                QD[a[0]].insert(a[1],QA[i]-1)
            elif abs(A[i])==2:
                QD[a[0]].insert(a[1],QA[i]+1)
            else:
                QD[a[0]].insert(a[1],QA[i])
        i+=1
    V=copy_2d(U)

    va=0
    VA=[]
    while va<len(A):
        if abs(A[va])<1:
            VA.append(int(100*A[va]))
        va+=1

    VB=[]
    VC=[]
    while len(VA)>0:
        vb=VA[0]
        if vb<0:
            vb=(-1)*vb
        VA.pop(0)
        if vb in VA:
            VB.append(vb)
        elif -vb in VA:
            VB.append(vb)
        else:
            VC.append(vb)
    VD=list(set(VC)-set(VB))

    va=0
    while va<len(VD):
        i=VD[va]-1
        print("i=",i)
        print("V=",V)
        print("VD=",VD)
        V[i]=V[i]*(-1)
        va+=1
    if U[u]>0:
        Z=[[1,C,1000,U,IC,QC],[-1,D,1000,V,ID,QD]]
    else:
        Z=[[1,D,1000,V,ID,QD],[-1,C,1000,U,IC,QC]]
    u+=1
```

```
else:
    F=copy_2d(E)
    IF=copy_2d(I)
    QF=copy_2d(Q)
    G=copy_2d(E)
    IG=copy_2d(I)
    QG=copy_2d(Q)
    F[a[0]].pop(a[1])
    O.append(IF[a[0]][a[1]])
    IF[a[0]].pop(a[1])
    QF[a[0]].pop(a[1])
    e=a[1]
    f=a[3]+1

    while f<len(F[a[2]]):
        F[a[0]].insert(e,F[a[2]][f])
        IF[a[0]].insert(e,IF[a[2]][f])
        QF[a[0]].insert(e,QF[a[2]][f])
        e+=1
        f+=1
    f=0
    while f<a[3]:
        F[a[0]].insert(e,F[a[2]][f])
        IF[a[0]].insert(e,IF[a[2]][f])
        QF[a[0]].insert(e,QF[a[2]][f])
        e+=1
        f+=1

    F.pop(a[2])
    IF.pop(a[2])
    QF.pop(a[2])
    G[a[0]].pop(a[1])
    IG[a[0]].pop(a[1])
    QG[a[0]].pop(a[1])
    B=[]
    IB=[]
    QB=[]
    e=a[1]
    f=a[3]-1

    while f>=0:
        G[a[0]].insert(e,G[a[2]][f])
        IG[a[0]].insert(e,IG[a[2]][f])
        if QG[a[2]][f]==3:
            QG[a[0]].insert(e,QG[a[2]][f]+1)
        elif QG[a[2]][f]==5:
            QG[a[0]].insert(e,QG[a[2]][f]-1)
        else:
            if abs(G[a[2]][f])==1:
                QG[a[0]].insert(e,QG[a[2]][f]-1)
            elif abs(G[a[2]][f])==2:
                QG[a[0]].insert(e,QG[a[2]][f]+1)
            else:
                QG[a[0]].insert(e,QG[a[2]][f])
        B.append(G[a[2]][f])
        IB.append(IG[a[2]][f])
        if QG[a[2]][f]==3:
            QB.append(QG[a[2]][f]+1)
        elif QG[a[2]][f]==5:
            QB.append(QG[a[2]][f]-1)
        else:
            if abs(G[a[2]][f])==1:
                QB.append(QG[a[2]][f]-1)
            elif abs(G[a[2]][f])==2:
                QB.append(QG[a[2]][f]+1)
            else:
                QB.append(QG[a[2]][f])
        e+=1
        f-=1
    f=len(G[a[2]])-1
    while f>a[3]:
        G[a[0]].insert(e,G[a[2]][f])
        IG[a[0]].insert(e,IG[a[2]][f])
        if QG[a[2]][f]==3:
            QG[a[0]].insert(e,QG[a[2]][f]+1)
        elif QG[a[2]][f]==5:
            QG[a[0]].insert(e,QG[a[2]][f]-1)
        else:
            if abs(G[a[2]][f])==1:
                QG[a[0]].insert(e,QG[a[2]][f]-1)
            elif abs(G[a[2]][f])==2:
                QG[a[0]].insert(e,QG[a[2]][f]+1)
            else:
                QG[a[0]].insert(e,QG[a[2]][f])
        B.append(G[a[2]][f])
        IB.append(IG[a[2]][f])
        if QG[a[2]][f]==3:
            QB.append(QG[a[2]][f]+1)
        elif QG[a[2]][f]==5:
            QB.append(QG[a[2]][f]-1)
        else:
            if abs(G[a[2]][f])==1:
                QB.append(QG[a[2]][f]-1)
            elif abs(G[a[2]][f])==2:
                QB.append(QG[a[2]][f]+1)
            else:
                QB.append(QG[a[2]][f])
        e+=1
        f-=1

    G.pop(a[2])
    IG.pop(a[2])
    QG.pop(a[2])
    V=copy_2d(U)
    va=0
    VA=[]
    while va<len(B):
        if abs(B[va])<1:
            VA.append(int(100*B[va]))
        va+=1

    VB=[]
    VC=[]
    while len(VA)>0:
        vb=VA[0]
        if vb<0:
            vb=(-1)*vb
        VA.pop(0)
        if vb in VA:
            VB.append(vb)
        elif -vb in VA:
            VB.append(vb)
        else:
            VC.append(vb)
    VD=list(set(VC)-set(VB))

    va=0
    while va<len(VD):
        i=VD[va]-1
        V[i]=V[i]*(-1)
        va+=1
    if U[u]>0:
        Z=[[1,F,1000,U,IF,QF],[-1,G,1000,V,IG,QG]]
    else:
        Z=[[1,G,1000,V,IG,QG],[-1,F,1000,U,IF,QF]]
    u+=1

s=0
while s<len(S):
    z=0
    while z<len(Z):
        E=Z[z][1]
        I=Z[z][4]
        Q=Z[z][5]
        U=Z[z][3]

        e=0
        a=[]

        while e<len(E):
            f=0
            while f<len(E[e]):
                if abs(E[e][f])==S[s]:
                    a.append(e)
                    a.append(f)
                    f+=1
                else:
                    f+=1
            e+=1
        if a[0]==a[2]:

            C=copy_2d(E)
            IC=copy_2d(I)
            QC=copy_2d(Q)
            A=[]
            IA=[]
            QA=[]

            e=a[1]
            f=a[3]
            while e<f-1:
                A.append(C[a[0]][e+1])
                IA.append(IC[a[0]][e+1])
                QA.append(QC[a[0]][e+1])
                C[a[0]].pop(e+1)
                IC[a[0]].pop(e+1)
                QC[a[0]].pop(e+1)
                f-=1

            C[a[0]].pop(a[1])
            C[a[0]].pop(a[1])
            O.append(IC[a[0]][a[1]])
            IC[a[0]].pop(a[1])
            IC[a[0]].pop(a[1])
            QC[a[0]].pop(a[1])
            QC[a[0]].pop(a[1])
            D=copy_2d(C)
            ID=copy_2d(IC)
            QD=copy_2d(QC)

            C.append(A)
            IC.append(IA)
            QC.append(QA)

            i=0
            while i<len(A):
                D[a[0]].insert(a[1],A[i])
                ID[a[0]].insert(a[1],IA[i])
                if QA[i]==3:
                    QD[a[0]].insert(a[1],QA[i]+1)
                elif QA[i]==5:
                    QD[a[0]].insert(a[1],QA[i]-1)
                else:
                    if abs(A[i])==1:
                        QD[a[0]].insert(a[1],QA[i]-1)
                    elif abs(A[i])==2:
                        QD[a[0]].insert(a[1],QA[i]+1)
                    else:
                        QD[a[0]].insert(a[1],QA[i])
                i+=1
            V=copy_2d(U)
            va=0
            VA=[]
            while va<len(A):
                if abs(A[va])<1:
                    VA.append(int(100*A[va]))
                va+=1

            VB=[]
            VC=[]
            while len(VA)>0:
                vb=VA[0]
                if vb<0:
                    vb=(-1)*vb
                VA.pop(0)
                if vb in VA:
                    VB.append(vb)
                elif -vb in VA:
                    VB.append(vb)
                else:
                    VC.append(vb)
            VD=list(set(VC)-set(VB))

            va=0
            while va<len(VD):
                i=VD[va]-1
                V[i]=V[i]*(-1)
                va+=1

            if Z[z][3][u]>0:
                Z.insert(z+1,[1,C,1000,U,IC,QC])
                Z.insert(z+2,[-1,D,1000,V,ID,QD])
            else:
                Z.insert(z+1,[1,D,1000,V,ID,QD])
                Z.insert(z+2,[-1,C,1000,U,IC,QC])

            Z[z+1][0]=Z[z+1][0]+Z[z][0]
            Z[z+2][0]=Z[z+2][0]+Z[z][0]
            Z[z+1][2]=int(Z[z+1][2]*Z[z][2]/1000)
            Z[z+2][2]=int(Z[z+2][2]*Z[z][2]/1000)
            Z.pop(z)
            z+=1

        else:
            F=copy_2d(E)
            IF=copy_2d(I)
            QF=copy_2d(Q)
            G=copy_2d(E)
            IG=copy_2d(I)
            QG=copy_2d(Q)
            F[a[0]].pop(a[1])
            O.append(IF[a[0]][a[1]])
            IF[a[0]].pop(a[1])
            QF[a[0]].pop(a[1])
            e=a[1]
            f=a[3]+1

            while f<len(F[a[2]]):
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            f=0
            while f<a[3]:
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1

            F.pop(a[2])
            IF.pop(a[2])
            QF.pop(a[2])
            G[a[0]].pop(a[1])
            IG[a[0]].pop(a[1])
            QG[a[0]].pop(a[1])
            B=[]
            IB=[]
            QB=[]
            e=a[1]
            f=a[3]-1

            while f>=0:
                G[a[0]].insert(e,G[a[2]][f])
                IG[a[0]].insert(e,IG[a[2]][f])
                if QG[a[2]][f]==3:
                    QG[a[0]].insert(e,QG[a[2]][f]+1)
                elif QG[a[2]][f]==5:
                    QG[a[0]].insert(e,QG[a[2]][f]-1)
                else:
                    if abs(G[a[2]][f])==1:
                        QG[a[0]].insert(e,QG[a[2]][f]-1)
                    elif abs(G[a[2]][f])==2:
                        QG[a[0]].insert(e,QG[a[2]][f]+1)
                    else:
                        QG[a[0]].insert(e,QG[a[2]][f])
                B.append(G[a[2]][f])
                IB.append(IG[a[2]][f])
            if QG[a[2]][f]==3:
                QB.append(QG[a[2]][f]+1)
            elif QG[a[2]][f]==5:
                QB.append(QG[a[2]][f]-1)
            else:
                if abs(G[a[2]][f])==1:
                    QB.append(QG[a[2]][f]-1)
                elif abs(G[a[2]][f])==2:
                    QB.append(QG[a[2]][f]+1)
                else:
                    QB.append(QG[a[2]][f])
            e+=1
            f-=1
        f=len(G[a[2]])-1
        while f>a[3]:
            G[a[0]].insert(e,G[a[2]][f])
            IG[a[0]].insert(e,IG[a[2]][f])
            if QG[a[2]][f]==3:
                QG[a[0]].insert(e,QG[a[2]][f]+1)
            elif QG[a[2]][f]==5:
                QG[a[0]].insert(e,QG[a[2]][f]-1)
            else:
                if abs(G[a[2]][f])==1:
                    QG[a[0]].insert(e,QG[a[2]][f]-1)
                elif abs(G[a[2]][f])==2:
                    QG[a[0]].insert(e,QG[a[2]][f]+1)
                else:
                    QG[a[0]].insert(e,QG[a[2]][f])
            B.append(G[a[2]][f])
            IB.append(IG[a[2]][f])
            if QG[a[2]][f]==3:
                QB.append(QG[a[2]][f]+1)
            elif QG[a[2]][f]==5:
                QB.append(QG[a[2]][f]-1)
            else:
                if abs(G[a[2]][f])==1:
                    QB.append(QG[a[2]][f]-1)
                elif abs(G[a[2]][f])==2:
                    QB.append(QG[a[2]][f]+1)
                else:
                    QB.append(QG[a[2]][f])
            e+=1
            f-=1

        G.pop(a[2])
        IG.pop(a[2])
        QG.pop(a[2])
        V=copy_2d(U)

        va=0
        VA=[]
        while va<len(B):
            if abs(B[va])<1:
                VA.append(int(100*B[va]))
            va+=1

        VB=[]
        VC=[]
        while len(VA)>0:
            vb=VA[0]
            if vb<0:
                vb=(-1)*vb
                VA.pop(0)
                if vb in VA:
                    VB.append(vb)
                elif -vb in VA:
                    VB.append(vb)
                else:
                    VC.append(vb)
            VD=list(set(VC)-set(VB))

            va=0
            while va<len(VD):
                i=VD[va]-1
                V[i]=V[i]*(-1)
                va+=1
            if Z[z][3][u]>0:
                Z.insert(z+1,[1,F,1000,U,IF,QF])
                Z.insert(z+2,[-1,G,1000,V,IG,QG])
            else:
                Z.insert(z+1,[1,G,1000,V,IG,QG])
                Z.insert(z+2,[-1,F,1000,U,IF,QF])

            Z[z+1][0]=Z[z+1][0]+Z[z][0]
            Z[z+2][0]=Z[z+2][0]+Z[z][0]
            Z[z+1][2]=int(Z[z+1][2]*Z[z][2]/1000)
            Z[z+2][2]=int(Z[z+2][2]*Z[z][2]/1000)
            Z.pop(z)
            z+=1
        z+=1
    s+=1
    u+=1

OUTPUT: Z
```

```
#Step 3: PROGRAM INDUCE
#User Inputs
#The output of Step 2 (Program Resolve)
#Or array of the form Z=[[0,E,1000,[1],I,Q]] for skein $K=[E,I,Q]$

INPUT: Z

import copy
def copy_2d(p):
    return copy.deepcopy(p)

zorb=0
while zorb<len(Z):
    Z[zorb].append([0,0,0,0,0,0])
    zorb+=1

crescendo=0
while crescendo<len(Z):
    jay=0
    DEX=[]
    while jay<len(Z[crescendo][4]):
        jj=0
        while jj<len(Z[crescendo][1][jay]):
            if abs(Z[crescendo][1][jay][jj])==1:
                DEX.append(Z[crescendo][4][jay][jj])
            jj+=1
        jay+=1

    babygary=0
    while babygary<len(Z[crescendo][1]):
        bg=-1
        while bg<len(Z[crescendo][1][babygary])-1:
            if abs(Z[crescendo][1][babygary][bg])==1 and Z[crescendo][1][babygary]
[bg]==Z[crescendo][1][babygary][bg+1]:
                remise=0
                smol=min(Z[crescendo][4][babygary][bg],Z[crescendo][4][babygary]
[bg+1])
                lorg=max(Z[crescendo][4][babygary][bg],Z[crescendo][4][babygary]
[bg+1])
                john=smol+1
                while john<lorg:
                    dex=0
                    while dex<len(DEX):
                        if DEX[dex]==john:
                            remise=1
                        dex+=1
                    john+=1
                if remise==0:
                    del Z[crescendo][1][babygary][bg+1]
                    del Z[crescendo][4][babygary][bg+1]
                    del Z[crescendo][5][babygary][bg+1]
                    del Z[crescendo][1][babygary][bg]
                    del Z[crescendo][4][babygary][bg]
                    del Z[crescendo][5][babygary][bg]
                    bg=-1
                    d=0
                    while d<len(DEX):
                        if DEX[d]==smol or DEX[d]==lorg:
                            del DEX[d]
                        else:
                            d+=1
                else:
                    bg+=1
                else:
                    bg+=1

            babygary+=1
        crescendo+=1

job=0
while job<len(Z):
    sob=0
    while sob<len(Z[job][1]):
        decide=0
        if len(Z[job][1][sob])==0:
            Z[job][6][4]=Z[job][6][4]+1
            del Z[job][1][sob]
            del Z[job][4][sob]
            del Z[job][5][sob]
            sob=0
        elif len(Z[job][1][sob])==2 and Z[job][1][sob][0]==-Z[job][1][sob][1]:
            if abs(Z[job][1][sob][0])==1:
                Z[job][6][0]=Z[job][6][0]+1
            elif abs(Z[job][1][sob][0])==2:
                Z[job][6][2]=Z[job][6][2]+1
            del Z[job][1][sob]
            del Z[job][4][sob]
            del Z[job][5][sob]
            sob=0

        elif len(Z[job][1][sob])==4:
            plutop=1
            rosa=0
            redover=0
            redunder=0
            blueover=0
            blueunder=0
            while rosa<4:
                if Z[job][1][sob][rosa]==1:
                    redover=Z[job][1][sob][rosa]
                    iredover=Z[job][4][sob][rosa]
                    qredover=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==-1:
                    redunder=Z[job][1][sob][rosa]
                    iredunder=Z[job][4][sob][rosa]
                    qredunder=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==2:
                    blueover=Z[job][1][sob][rosa]
                    iblueover=Z[job][4][sob][rosa]
                    qblueover=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==-2:
                    blueunder=Z[job][1][sob][rosa]
                    iblueunder=Z[job][4][sob][rosa]
                    qblueunder=Z[job][5][sob][rosa]
                rosa+=1

            if abs(redover)>0 and abs(redunder)>0:
                imaxred=max(iredover,iredunder)
                iminred=min(iredover,iredunder)
                rosa=iminred+1
                while rosa<imaxred:
                    coza=0
                    while coza<len(Z[job][4]):
                        sosa=0
                        while sosa<len(Z[job][4][coza]):
                            if Z[job][4][coza][sosa]==rosa:
                                if abs(Z[job][1][coza][sosa])==1:
                                    plutop=0
                            sosa+=1
                        coza+=1
                    rosa+=1

            if abs(blueover)>0 and abs(blueunder)>0:
                imaxblue=max(iblueover,iblueunder)
                iminblue=min(iblueover,iblueunder)
                rosa=iminblue+1
                while rosa<imaxblue:
                    coza=0
                    while coza<len(Z[job][4]):
                        sosa=0
                        while sosa<len(Z[job][4][coza]):
                            if Z[job][4][coza][sosa]==rosa:
                                if abs(Z[job][1][coza][sosa])==2:
                                    plutop=0
                            sosa+=1
                        coza+=1
                    rosa+=1

            if abs(Z[job][1][sob][0])==abs(Z[job][1][sob][1]) and abs(Z[job][1][sob]
[1])==abs(Z[job][1][sob][2]) and abs(Z[job][1][sob][2])==abs(Z[job][1][sob][3]):
                decide=0
            elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob][2]==-Z[job]
[1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==Z[job][1][sob][2]/abs(Z[job]
[1][sob][2]):
                decide=1
            elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==Z[job][1][sob][1]/abs(Z[job]
[1][sob][1]):
                decide=1
            elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob][2]==-Z[job]
[1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==-Z[job][1][sob][3]/
abs(Z[job][1][sob][3]):
                decide=2
            elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==-Z[job][1][sob][2]/
abs(Z[job][1][sob][2]):
                decide=2
            elif Z[job][1][sob][0]==Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2]:
                decide=3

            if decide>0 and plutop==1:
                b=[Z[job][4][sob][0],Z[job][4][sob][1],Z[job][4][sob][2],Z[job][4]
[sob][3]]
                bmax=max(b)
                bmin=min(b)
                dmax=[]
                bay=0
                while bay<len(Z[job][4]):
                    if len(Z[job][4][bay])>0:
                        dmax.append(max(Z[job][4][bay]))
                    bay+=1
                decidemax=max(dmax)
                dmin=[]
                bay=0
                while bay<len(Z[job][4]):
                    if len(Z[job][4][bay])>0:
                        dmin.append(min(Z[job][4][bay]))
                    bay+=1
                decidemin=min(dmin)

                if decidemax==bmax:
                    if decide==1:
                        Z[job][6][1]=Z[job][6][1]+1
                    if decide==2:
                        call=0
                        while call<len(Z[job][1][sob]):
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                elif decidemin==bmin:
                    if decide==1:
                        Z[job][6][1]=Z[job][6][1]+1
                    elif decide==2:
                        call=0
                        while call<len(Z[job][1][sob]):
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                else:
                    sob+=1
            else:
                sob+=1
        else:
            sob+=1
    job+=1

z=0
```

```
while z<len(Z):
    jay=0
    DEX=[]
    while jay<len(Z[z][4]):
        jj=0
        while jj<len(Z[z][1][jay]):
            if abs(Z[z][1][jay][jj])==1:
                DEX.append(Z[z][4][jay][jj])
            jj+=1
        jay+=1

    babygary=0
    while babygary<len(Z[z][1]):
        bg=-1
        while bg<len(Z[z][1][babygary])-1:
            if abs(Z[z][1][babygary][bg])==1 and Z[z][1][babygary][bg]==Z[z][1]
[babygary][bg+1]:
                remise=0
                smol=min(Z[z][4][babygary][bg],Z[z][4][babygary][bg+1])
                lorg=max(Z[z][4][babygary][bg],Z[z][4][babygary][bg+1])
                john=smol+1
                while john<lorg:
                    dex=0
                    while dex<len(DEX):
                        if DEX[dex]==john:
                            remise=1
                        dex+=1
                    john+=1
                if remise==0:
                    del Z[z][1][babygary][bg+1]
                    del Z[z][4][babygary][bg+1]
                    del Z[z][5][babygary][bg+1]
                    del Z[z][1][babygary][bg]
                    del Z[z][4][babygary][bg]
                    del Z[z][5][babygary][bg]
                    bg=-1
                    d=0
                    while d<len(DEX):
                        if DEX[d]==smol or DEX[d]==lorg:
                            del DEX[d]
                        else:
                            d+=1
                else:
                    bg+=1
            else:
                bg+=1
        babygary+=1

    ONES=[]
    r=0
    while r<len(Z[z][1]):

        if len(Z[z][1][r])<=2:
            if len(Z[z][1][r])==0 or Z[z][1][r][0]==Z[z][1][r][1]:
                Z[z][6][4]=Z[z][6][4]+1
            elif abs(Z[z][1][r][0])==1:
                Z[z][6][0]=Z[z][6][0]+1
            elif abs(Z[z][1][r][0])==2:
                Z[z][6][2]=Z[z][6][2]+1
            del Z[z][1][r]
            del Z[z][4][r]
            del Z[z][5][r]
        s=0
        while r<len(Z[z][1]) and s<len(Z[z][1][r]):
            if abs(Z[z][1][r][s])==1:
                ONES.append([Z[z][1][r][s],Z[z][4][r][s],r,s])
            s+=1
        r+=1
    POS=[]
    NEG=[]
    BOT=[]
    one=0
    while one<len(ONES):

        BOT.append(ONES[one][1])

        if ONES[one][0]==1:
            POS.append(ONES[one][1])
        else:
            NEG.append(ONES[one][1])

        one+=1

    if len(POS)>0:
        minpos=min(POS)
        maxpos=max(POS)

    if len(NEG)>0:
        minneg=min(NEG)
        maxneg=max(NEG)

    if len(BOT)>0:
        maxbot=max(BOT)
        minbot=min(BOT)

    while len(POS)>0:

        rancid=0
        while len(POS)>0 and minbot==minpos and rancid==0:
            if len(BOT)>0:
                BOT.remove(minbot)
            if len(POS)>0:
                POS.remove(minpos)
            if len(BOT)>0:
                minbot=min(BOT)
            if len(POS)>0:
                minpos=min(POS)
            else:
                rancid=1
        jo=0

        if rancid==0:
            NEGG=list.copy(NEG)
            if len(NEGG)>0:
                nextneg=max(NEGG)
                jo=1
            while jo==1 and nextneg>minpos:
                NEGG.remove(nextneg)
                if len(NEGG)>0:
                    nextneg=max(NEGG)
                else:
                    jo=0
            run=0
            while run<len(ONES):
                if ONES[run][1]==minpos:
                    k1=ONES[run][2]
                    k2=ONES[run][3]
                elif ONES[run][1]==nextneg:
                    q1=ONES[run][2]
                    q2=ONES[run][3]
                run+=1

        if rancid==0:

            bob=Z[z][4][k1][k2]
            Z[z][4][k1][k2]=Z[z][4][q1][q2]
            Z[z][4][q1][q2]=bob
            moe=0

            if abs(k2-q2)==1:
                moe=1

            pos1=k1
            pos2=k2
            neg1=q1
            neg2=q2
            if pos1==neg1 and neg2==0 and pos2==len(Z[z][1][k1])-1:
                curly=Z[z][1][k1][neg2]
                del Z[z][1][k1][neg2]
                Z[z][1][k1].append(curly)
                larry=Z[z][4][k1][neg2]
                del Z[z][4][k1][neg2]
                Z[z][4][k1].append(larry)
                morton=Z[z][5][k1][neg2]
                del Z[z][5][k1][neg2]
                Z[z][5][k1].append(morton)

                neg2=len(Z[z][1][k1])-1
                pos2-=1
                moe=1

            elif pos1==neg1 and pos2==0 and neg2==len(Z[z][1][k1])-1:
                curly=Z[z][1][k1][pos2]
                del Z[z][1][k1][pos2]
                Z[z][1][k1].append(curly)
                larry=Z[z][4][k1][pos2]
                del Z[z][4][k1][pos2]
                Z[z][4][k1].append(larry)
                morton=Z[z][5][k1][pos2]
                del Z[z][5][k1][pos2]
                Z[z][5][k1].append(morton)

                pos2=len(Z[z][1][k1])-1
                neg2-=1
                moe=1
            KAPPA=[]

        if rancid==0 and pos1==neg1 and moe==1:

            if pos2<neg2:
                if Z[z][5][pos1][pos2]<Z[z][5][pos1][neg2]:
                    Z[z][3]=[-1]
                    Z[z][0]=Z[z][0]+3
                    Z[z][2]=-Z[z][2]
                else:
                    Z[z][3]=[1]
                Z[z][0]=Z[z][0]-3
                Z[z][2]=-Z[z][2]
            Z[z][1][pos1].insert(pos2,-0.99)
            Z[z][4][pos1].insert(pos2,99)
            Z[z][5][pos1].insert(pos2,99)
            Z[z][1][pos1].insert(neg2+2,0.99)
            Z[z][4][pos1].insert(neg2+2,99)
            Z[z][5][pos1].insert(neg2+2,99)

            a=[pos1,pos2,neg1,neg2+2]
        else:
            if Z[z][5][pos1][pos2]<Z[z][5][pos1][neg2]:
                Z[z][3]=[1]
                Z[z][0]=Z[z][0]-3
                Z[z][2]=-Z[z][2]
            else:
                Z[z][3]=[-1]
                Z[z][0]=Z[z][0]+3
                Z[z][2]=-Z[z][2]
            Z[z][1][pos1].insert(neg2,0.99)
            Z[z][4][pos1].insert(neg2,99)
            Z[z][5][pos1].insert(neg2,99)
            Z[z][1][pos1].insert(pos2+2,-0.99)
            Z[z][4][pos1].insert(pos2+2,99)
            Z[z][5][pos1].insert(pos2+2,99)
            a=[neg1,neg2,pos1,pos2+2]

        E=Z[z][1]
        I=Z[z][4]
        Q=Z[z][5]
        X=Z[z][6]

        O=[]
        U=Z[z][3]
        u=0
        s=0.99

        C=copy_2d(E)
        IC=copy_2d(I)
        XC=copy_2d(X)
        QC=copy_2d(Q)

        A=[]
        IA=[]
        QA=[]

        e=a[1]
        f=a[3]
        while e<f-1:
            A.append(C[a[0]][e+1])
            IA.append(IC[a[0]][e+1])
            QA.append(QC[a[0]][e+1])
            C[a[0]].pop(e+1)
            IC[a[0]].pop(e+1)
            QC[a[0]].pop(e+1)
            f-=1

        C[a[0]].pop(a[1])
        C[a[0]].pop(a[1])
        O.append(IC[a[0]][a[1]])
        IC[a[0]].pop(a[1])
        IC[a[0]].pop(a[1])
```

```
            QC[a[0]].pop(a[1])
            QC[a[0]].pop(a[1])

            D=copy_2d(C)
            ID=copy_2d(IC)
            XD=copy_2d(XC)
            QD=copy_2d(QC)

            C.append(A)
            IC.append(IA)
            QC.append(QA)

            i=0
            while i<len(A):
                D[a[0]].insert(a[1],A[i])
                ID[a[0]].insert(a[1],IA[i])
                if QA[i]==3:
                    QD[a[0]].insert(a[1],QA[i]+1)
                elif QA[i]==5:
                    QD[a[0]].insert(a[1],QA[i]-1)
                else:
                    if abs(A[i])==1:
                        QD[a[0]].insert(a[1],QA[i]-1)
                    elif abs(A[i])==2:
                        QD[a[0]].insert(a[1],QA[i]+1)
                    else:
                        QD[a[0]].insert(a[1],QA[i])
                i+=1

            V=copy_2d(U)
            va=0
            VA=[]
            while va<len(A):
                if abs(A[va])<1:
                    VA.append(int(100*A[va]))
                va+=1

            VB=[]
            VC=[]
            while len(VA)>0:
                vb=VA[0]
                if vb<0:
                    vb=(-1)*vb
                VA.pop(0)
                if vb in VA:
                    VB.append(vb)
                elif -vb in VA:
                    VB.append(vb)
                else:
                    VC.append(vb)
            VD=list(set(VC)-set(VB))

            va=0
            while va<len(VD):
                i=VD[va]-1
                V[i]=V[i]*(-1)
                va+=1

            if U[u]>0:
                Z.insert(z+1,[1,C,1000,U,IC,QC,XC])
                Z.insert(z+2,[-1,D,1000,V,ID,QD,XD])
            else:
```

```
            Z.insert(z+1,[1,D,1000,V,ID,QD,XD])
            Z.insert(z+2,[-1,C,1000,U,IC,QC,XC])

        hay=0
        REX=[]
        while hay<len(Z[z+1][1]):
            hh=0
            while hh<len(Z[z+1][1][hay]):
                if abs(Z[z+1][1][hay][hh])==1:
                    REX.append(Z[z+1][4][hay][hh])
                hh+=1
            hay+=1
        babyhary=0
        while babyhary<len(Z[z+1][1]):
            bh=-1
            while bh<len(Z[z+1][1][babyhary])-1:
                if abs(Z[z+1][1][babyhary][bh])==1 and Z[z+1][1][babyhary]
[bh]==Z[z+1][1][babyhary][bh+1]:
                    smolg=min(Z[z+1][4][babyhary][bh],Z[z+1][4][babyhary][bh+1])
                    lorgg=max(Z[z+1][4][babyhary][bh],Z[z+1][4][babyhary][bh+1])
                    johhn=smolg+1
                    while johhn<lorgg:
                        rex=0
                        while rex<len(REX):
                            if REX[rex]==johhn:
                                demise=1
                            rex+=1
                        johhn+=1
                    if demise==0:
                        del Z[z+1][1][babyhary][bh+1]
                        del Z[z+1][4][babyhary][bh+1]
                        del Z[z+1][5][babyhary][bh+1]
                        del Z[z+1][1][babyhary][bh]
                        del Z[z+1][4][babyhary][bh]
                        del Z[z+1][5][babyhary][bh]
                        bh=-1
                        dh=0
                        while dh<len(REX):
                            if REX[dh]==smolg or REX[dh]==lorgg:
                                del REX[dh]
                            else:
                                dh+=1
                    else:
                        bh+=1
                else:
                    bh+=1
            babyhary+=1

        hay=0
        REX=[]
        while hay<len(Z[z+2][1]):
            hh=0
            while hh<len(Z[z+2][1][hay]):
                if abs(Z[z+2][1][hay][hh])==1:
                    REX.append(Z[z+2][4][hay][hh])
                hh+=1
            hay+=1
        babyhary=0
        while babyhary<len(Z[z+2][1]):
            bh=-1
            while bh<len(Z[z+2][1][babyhary])-1:
                if abs(Z[z+2][1][babyhary][bh])==1 and Z[z+2][1][babyhary]
```

```
[bh]==Z[z+2][1][babyhary][bh+1]:
                        demise=0
                        smolg=min(Z[z+2][4][babyhary][bh],Z[z+2][4][babyhary][bh+1])
                        lorgg=max(Z[z+2][4][babyhary][bh],Z[z+2][4][babyhary][bh+1])
                        johhn=smolg+1
                        while johhn<lorgg:
                            rex=0
                            while rex<len(REX):
                                if REX[rex]==johhn:
                                    demise=1
                                rex+=1
                            johhn+=1
                        if demise==0:
                            del Z[z+2][1][babyhary][bh+1]
                            del Z[z+2][4][babyhary][bh+1]
                            del Z[z+2][5][babyhary][bh+1]
                            del Z[z+2][1][babyhary][bh]
                            del Z[z+2][4][babyhary][bh]
                            del Z[z+2][5][babyhary][bh]
                            bh=-1
                            dh=0
                            while dh<len(REX):
                                if REX[dh]==smolg or REX[dh]==lorgg:
                                    del REX[dh]
                                else:
                                    dh+=1
                        else:
                            bh+=1
                    else:
                        bh+=1
                babyhary+=1

            Z[z+1][0]=Z[z+1][0]+Z[z][0]
            Z[z+2][0]=Z[z+2][0]+Z[z][0]
            Z[z+1][2]=int(Z[z+1][2]*Z[z][2]/1000)
            Z[z+2][2]=int(Z[z+2][2]*Z[z][2]/1000)
            Z.pop(z)

            u+=1

        elif rancid==0:
            if k1==q1:
                if pos2<neg2:
                    if Z[z][5][k1][pos2]>Z[z][5][k1][neg2]:
                        Z[z][3]=[-1,1]
                        Z[z][1][k1].insert(pos2,0.02)
                        Z[z][4][k1].insert(pos2,92)
                        Z[z][5][k1].insert(pos2,92)
                        Z[z][1][k1].insert(pos2+2,0.01)
                        Z[z][4][k1].insert(pos2+2,91)
                        Z[z][5][k1].insert(pos2+2,91)
                        Z[z][1][q1].insert(neg2+2,-0.01)
                        Z[z][4][q1].insert(neg2+2,91)
                        Z[z][5][q1].insert(neg2+2,91)
                        Z[z][1][q1].insert(neg2+4,-0.02)
                        Z[z][4][q1].insert(neg2+4,92)
                        Z[z][5][q1].insert(neg2+4,92)

                    else:
                        Z[z][3]=[-1,1]
                        Z[z][1][k1].insert(pos2,0.01)
                        Z[z][4][k1].insert(pos2,91)
                        Z[z][5][k1].insert(pos2,91)
                        Z[z][1][k1].insert(pos2+2,0.02)
                        Z[z][4][k1].insert(pos2+2,92)
                        Z[z][5][k1].insert(pos2+2,92)
                        Z[z][1][q1].insert(neg2+2,-0.02)
                        Z[z][4][q1].insert(neg2+2,92)
                        Z[z][5][q1].insert(neg2+2,92)
                        Z[z][1][q1].insert(neg2+4,-0.01)
                        Z[z][4][q1].insert(neg2+4,91)
                        Z[z][5][q1].insert(neg2+4,91)

                else:
                    if Z[z][5][k1][neg2]>Z[z][5][k1][pos2]:
                        Z[z][3]=[-1,1]
                        Z[z][1][k1].insert(pos2,0.01)
                        Z[z][4][k1].insert(pos2,91)
                        Z[z][5][k1].insert(pos2,91)
                        Z[z][1][k1].insert(pos2+2,0.02)
                        Z[z][4][k1].insert(pos2+2,92)
                        Z[z][5][k1].insert(pos2+2,92)
                        Z[z][1][q1].insert(neg2,-0.02)
                        Z[z][4][q1].insert(neg2,92)
                        Z[z][5][q1].insert(neg2,92)
                        Z[z][1][q1].insert(neg2+2,-0.01)
                        Z[z][4][q1].insert(neg2+2,91)
                        Z[z][5][q1].insert(neg2+2,91)
                    else:
                        Z[z][3]=[-1,1]
                        Z[z][1][k1].insert(pos2,0.02)
                        Z[z][4][k1].insert(pos2,92)
                        Z[z][5][k1].insert(pos2,92)
                        Z[z][1][k1].insert(pos2+2,0.01)
                        Z[z][4][k1].insert(pos2+2,91)
                        Z[z][5][k1].insert(pos2+2,91)
                        Z[z][1][q1].insert(neg2,-0.01)
                        Z[z][4][q1].insert(neg2,91)
                        Z[z][5][q1].insert(neg2,91)
                        Z[z][1][q1].insert(neg2+2,-0.02)
                        Z[z][4][q1].insert(neg2+2,92)
                        Z[z][5][q1].insert(neg2+2,92)
            else:
                if Z[z][5][k1][pos2]>Z[z][5][q1][neg2]:
                    Z[z][3]=[-1,1]
                    Z[z][1][k1].insert(pos2,0.02)
                    Z[z][4][k1].insert(pos2,92)
                    Z[z][5][k1].insert(pos2,92)
                    Z[z][1][k1].insert(pos2+2,0.01)
                    Z[z][4][k1].insert(pos2+2,91)
                    Z[z][5][k1].insert(pos2+2,91)
                    Z[z][1][q1].insert(neg2,-0.01)
                    Z[z][4][q1].insert(neg2,91)
                    Z[z][5][q1].insert(neg2,91)
                    Z[z][1][q1].insert(neg2+2,-0.02)
                    Z[z][4][q1].insert(neg2+2,92)
                    Z[z][5][q1].insert(neg2+2,92)
                elif Z[z][5][q1][neg2]>Z[z][5][k1][pos2]:
                    Z[z][3]=[-1,1]
                    Z[z][1][k1].insert(pos2,0.01)
                    Z[z][4][k1].insert(pos2,91)
                    Z[z][5][k1].insert(pos2,91)
                    Z[z][1][k1].insert(pos2+2,0.02)
                    Z[z][4][k1].insert(pos2+2,92)
                Z[z][5][k1].insert(pos2+2,92)
                Z[z][1][q1].insert(neg2,-0.02)
                Z[z][4][q1].insert(neg2,92)
                Z[z][5][q1].insert(neg2,92)
                Z[z][1][q1].insert(neg2+2,-0.01)
                Z[z][4][q1].insert(neg2+2,91)
                Z[z][5][q1].insert(neg2+2,91)

            elif Z[z][5][k1][pos2]==3 and Z[z][5][q1][neg2]==3:
                Z[z][3]=[1,-1]
                Z[z][1][k1].insert(pos2,0.01)
                Z[z][4][k1].insert(pos2,91)
                Z[z][5][k1].insert(pos2,91)
                Z[z][1][k1].insert(pos2+2,0.02)
                Z[z][4][k1].insert(pos2+2,92)
                Z[z][5][k1].insert(pos2+2,92)
                Z[z][1][q1].insert(neg2,-0.01)
                Z[z][4][q1].insert(neg2,91)
                Z[z][5][q1].insert(neg2,91)
                Z[z][1][q1].insert(neg2+2,-0.02)
                Z[z][4][q1].insert(neg2+2,92)
                Z[z][5][q1].insert(neg2+2,92)

            elif Z[z][5][k1][pos2]==4 and Z[z][5][q1][neg2]==4:
                Z[z][3]=[1,-1]
                Z[z][1][k1].insert(pos2,0.02)
                Z[z][4][k1].insert(pos2,92)
                Z[z][5][k1].insert(pos2,92)
                Z[z][1][k1].insert(pos2+2,0.01)
                Z[z][4][k1].insert(pos2+2,91)
                Z[z][5][k1].insert(pos2+2,91)
                Z[z][1][q1].insert(neg2,-0.02)
                Z[z][4][q1].insert(neg2,92)
                Z[z][5][q1].insert(neg2,92)
                Z[z][1][q1].insert(neg2+2,-0.01)
                Z[z][4][q1].insert(neg2+2,91)
                Z[z][5][q1].insert(neg2+2,91)

        E= Z[z][1]
        I= Z[z][4]
        Q= Z[z][5]
        X= Z[z][6]

        O=[]
        S=[0.02]
        U=Z[z][3]
        u=0
        s=0.01
        e=0
        a=[]

        while e<len(E):
            f=0
            while f<len(E[e]):
                if abs(E[e][f])==s:
                    a.append(e)
                    a.append(f)
                    f+=1
                else:
                    f+=1
            e+=1
        if a[0]==a[2]:
            C=copy_2d(E)
            IC=copy_2d(I)
            XC=copy_2d(X)
            QC=copy_2d(Q)
            A=[]
            IA=[]
            QA=[]
            e=a[1]
            f=a[3]
            while e<f-1:
                A.append(C[a[0]][e+1])
                IA.append(IC[a[0]][e+1])
                QA.append(QC[a[0]][e+1])
                C[a[0]].pop(e+1)
                IC[a[0]].pop(e+1)
                QC[a[0]].pop(e+1)
                f-=1
            C[a[0]].pop(a[1])
            C[a[0]].pop(a[1])
            O.append(IC[a[0]][a[1]])
            IC[a[0]].pop(a[1])
            IC[a[0]].pop(a[1])
            QC[a[0]].pop(a[1])
            QC[a[0]].pop(a[1])
            D=copy_2d(C)
            ID=copy_2d(IC)
            XD=copy_2d(X)
            QD=copy_2d(QC)
            C.append(A)
            IC.append(IA)
            QC.append(QA)

            i=0
            while i<len(A):
                D[a[0]].insert(a[1],A[i])
                ID[a[0]].insert(a[1],IA[i])
                if QA[i]==3:
                    QD[a[0]].insert(a[1],QA[i]+1)
                elif QA[i]==5:
                    QD[a[0]].insert(a[1],QA[i]-1)
                else:
                    if abs(A[i])==1:
                        QD[a[0]].insert(a[1],QA[i]-1)
                    elif abs(A[i])==2:
                        QD[a[0]].insert(a[1],QA[i]+1)
                    else:
                        QD[a[0]].insert(a[1],QA[i])
                i+=1
            V=copy_2d(U)
            va=0
            VA=[]
            while va<len(A):
                if abs(A[va])<1:
                    VA.append(int(100*A[va]))
                va+=1
            VB=[]
            VC=[]
            while len(VA)>0:
                vb=VA[0]
                if vb<0:
                    vb=(-1)*vb
                VA.pop(0)
                if vb in VA:
                    VB.append(vb)
                elif -vb in VA:
                    VB.append(vb)
                else:
                    VC.append(vb)
            VD=list(set(VC)-set(VB))
            va=0
            while va<len(VD):
                i=VD[va]-1
                V[i]=V[i]*(-1)
                va+=1
            if U[u]>0:
                KAPPA=[[1,C,1000,U,IC,QC,XC],[-1,D,1000,V,ID,QD,XD]]
            else:
                KAPPA=[[1,D,1000,V,ID,QD,XD],[-1,C,1000,U,IC,QC,XC]]
            u+=1
        else:
            F=copy_2d(E)
            IF=copy_2d(I)
            XF=copy_2d(X)
            QF=copy_2d(Q)
            G=copy_2d(E)
            IG=copy_2d(I)
            XG=copy_2d(X)
            QG=copy_2d(Q)
            F[a[0]].pop(a[1])
            O.append(IF[a[0]][a[1]])
            IF[a[0]].pop(a[1])
            QF[a[0]].pop(a[1])
            e=a[1]
            f=a[3]+1
            while f<len(F[a[2]]):
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            f=0
            while f<a[3]:
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            F.pop(a[2])
            IF.pop(a[2])
            QF.pop(a[2])
            G[a[0]].pop(a[1])
            IG[a[0]].pop(a[1])
            QG[a[0]].pop(a[1])
            B=[]
            IB=[]
            QB=[]
            e=a[1]
            f=a[3]-1
            while f>=0:
                G[a[0]].insert(e,G[a[2]][f])
                IG[a[0]].insert(e,IG[a[2]][f])
                if QG[a[2]][f]==3:
                    QG[a[0]].insert(e,QG[a[2]][f]+1)
        elif QG[a[2]][f]==5:
            QG[a[0]].insert(e,QG[a[2]][f]-1)
        else:
            if abs(G[a[2]][f])==1:
                QG[a[0]].insert(e,QG[a[2]][f]-1)
            elif abs(G[a[2]][f])==2:
                QG[a[0]].insert(e,QG[a[2]][f]+1)
            else:
                QG[a[0]].insert(e,QG[a[2]][f])
        B.append(G[a[2]][f])
        IB.append(IG[a[2]][f])
        if QG[a[2]][f]==3:
            QB.append(QG[a[2]][f]+1)
        elif QG[a[2]][f]==5:
            QB.append(QG[a[2]][f]-1)
        else:
            if abs(G[a[2]][f])==1:
                QB.append(QG[a[2]][f]-1)
            elif abs(G[a[2]][f])==2:
                QB.append(QG[a[2]][f]+1)
            else:
                QB.append(QG[a[2]][f])
        e+=1
        f-=1
    f=len(G[a[2]])-1
    while f>a[3]:
        G[a[0]].insert(e,G[a[2]][f])
        IG[a[0]].insert(e,IG[a[2]][f])
        if QG[a[2]][f]==3:
            QG[a[0]].insert(e,QG[a[2]][f]+1)
        elif QG[a[2]][f]==5:
            QG[a[0]].insert(e,QG[a[2]][f]-1)
        else:
            if abs(G[a[2]][f])==1:
                QG[a[0]].insert(e,QG[a[2]][f]-1)
            elif abs(G[a[2]][f])==2:
                QG[a[0]].insert(e,QG[a[2]][f]+1)
            else:
                QG[a[0]].insert(e,QG[a[2]][f])
        B.append(G[a[2]][f])
        IB.append(IG[a[2]][f])
        if QG[a[2]][f]==3:
            QB.append(QG[a[2]][f]+1)
        elif QG[a[2]][f]==5:
            QB.append(QG[a[2]][f]-1)
        else:
            if abs(G[a[2]][f])==1:
                QB.append(QG[a[2]][f]-1)
            elif abs(G[a[2]][f])==2:
                QB.append(QG[a[2]][f]+1)
            else:
                QB.append(QG[a[2]][f])
        e+=1
        f-=1
    G.pop(a[2])
    IG.pop(a[2])
    QG.pop(a[2])
    V=copy_2d(U)
    va=0
    VA=[]
    while va<len(B):
        if abs(B[va])<1:
```

```
                    VA.append(int(100*B[va]))
                va+=1
            VB=[]
            VC=[]
            while len(VA)>0:
                vb=VA[0]
                if vb<0:
                    vb=(-1)*vb
                VA.pop(0)
                if vb in VA:
                    VB.append(vb)
                elif -vb in VA:
                    VB.append(vb)
                else:
                    VC.append(vb)
            VD=list(set(VC)-set(VB))
            va=0
            while va<len(VD):
                i=VD[va]-1
                V[i]=V[i]*(-1)
                va+=1
            if U[u]>0:
                KAPPA=[[1,F,1000,U,IF,QF,XF],[-1,G,1000,V,IG,QG,XG]]
            else:
                KAPPA=[[1,G,1000,V,IG,QG,XG],[-1,F,1000,U,IF,QF,XF]]
            u+=1
        s=0
        while s<len(S):
            z2=0
            while z2<len(KAPPA):
                E=KAPPA[z2][1]
                I=KAPPA[z2][4]
                Q=KAPPA[z2][5]
                X=KAPPA[z2][6]
                U=KAPPA[z2][3]
                e=0
                a=[]
                while e<len(E):
                    f=0
                    while f<len(E[e]):
                        if abs(E[e][f])==S[s]:
                            a.append(e)
                            a.append(f)
                            f+=1
                        else:
                            f+=1
                    e+=1
                if a[0]==a[2]:
                    C=copy_2d(E)
                    IC=copy_2d(I)
                    XC=copy_2d(X)
                    QC=copy_2d(Q)
                    A=[]
                    IA=[]
                    QA=[]
                    e=a[1]
                    f=a[3]
                    while e<f-1:
                        A.append(C[a[0]][e+1])
                        IA.append(IC[a[0]][e+1])
                        QA.append(QC[a[0]][e+1])
                        C[a[0]].pop(e+1)
        IC[a[0]].pop(e+1)
        QC[a[0]].pop(e+1)
        f-=1
    C[a[0]].pop(a[1])
    C[a[0]].pop(a[1])
    O.append(IC[a[0]][a[1]])
    IC[a[0]].pop(a[1])
    IC[a[0]].pop(a[1])
    QC[a[0]].pop(a[1])
    QC[a[0]].pop(a[1])
    D=copy_2d(C)
    ID=copy_2d(IC)
    XD=copy_2d(X)
    QD=copy_2d(QC)
    C.append(A)
    IC.append(IA)
    QC.append(QA)
    i=0
    while i<len(A):
        D[a[0]].insert(a[1],A[i])
        ID[a[0]].insert(a[1],IA[i])
        if QA[i]==3:
            QD[a[0]].insert(a[1],QA[i]+1)
        elif QA[i]==5:
            QD[a[0]].insert(a[1],QA[i]-1)
        else:
            if abs(A[i])==1:
                QD[a[0]].insert(a[1],QA[i]-1)
            elif abs(A[i])==2:
                QD[a[0]].insert(a[1],QA[i]+1)
            else:
                QD[a[0]].insert(a[1],QA[i])
        i+=1
    V=copy_2d(U)
    va=0
    VA=[]
    while va<len(A):
        if abs(A[va])<1:
            VA.append(int(100*A[va]))
        va+=1
    VB=[]
    VC=[]
    while len(VA)>0:
        vb=VA[0]
        if vb<0:
            vb=(-1)*vb
        VA.pop(0)
        if vb in VA:
            VB.append(vb)
        elif -vb in VA:
            VB.append(vb)
        else:
            VC.append(vb)
    VD=list(set(VC)-set(VB))
    va=0
    while va<len(VD):
        i=VD[va]-1
        V[i]=V[i]*(-1)
        va+=1

    if KAPPA[z2][3][u]>0:
        KAPPA.insert(z2+1,[1,C,1000,U,IC,QC,XC])
```

```
                KAPPA.insert(z2+2,[-1,D,1000,V,ID,QD,XD])
            else:
                KAPPA.insert(z2+1,[1,D,1000,V,ID,QD,XD])
                KAPPA.insert(z2+2,[-1,C,1000,U,IC,QC,XC])
            KAPPA[z2+1][0]=KAPPA[z2+1][0]+KAPPA[z2][0]
            KAPPA[z2+2][0]=KAPPA[z2+2][0]+KAPPA[z2][0]
            KAPPA[z2+1][2]=int(KAPPA[z2+1][2]*KAPPA[z2][2]/1000)
            KAPPA[z2+2][2]=int(KAPPA[z2+2][2]*KAPPA[z2][2]/1000)
            KAPPA.pop(z2)
            z2+=1
        else:
            F=copy_2d(E)
            IF=copy_2d(I)
            XF=copy_2d(X)
            QF=copy_2d(Q)
            G=copy_2d(E)
            IG=copy_2d(I)
            XG=copy_2d(X)
            QG=copy_2d(Q)
            F[a[0]].pop(a[1])
            O.append(IF[a[0]][a[1]])
            IF[a[0]].pop(a[1])
            QF[a[0]].pop(a[1])
            e=a[1]
            f=a[3]+1
            while f<len(F[a[2]]):
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            f=0
            while f<a[3]:
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            F.pop(a[2])
            IF.pop(a[2])
            QF.pop(a[2])
            G[a[0]].pop(a[1])
            IG[a[0]].pop(a[1])
            QG[a[0]].pop(a[1])
            B=[]
            IB=[]
            QB=[]
            e=a[1]
            f=a[3]-1
            while f>=0:
                G[a[0]].insert(e,G[a[2]][f])
                IG[a[0]].insert(e,IG[a[2]][f])
                if QG[a[2]][f]==3:
                    QG[a[0]].insert(e,QG[a[2]][f]+1)
                elif QG[a[2]][f]==5:
                    QG[a[0]].insert(e,QG[a[2]][f]-1)
                else:
                    if abs(G[a[2]][f])==1:
                        QG[a[0]].insert(e,QG[a[2]][f]-1)
                    elif abs(G[a[2]][f])==2:
                        QG[a[0]].insert(e,QG[a[2]][f]+1)
                    else:
```

```
            QG[a[0]].insert(e,QG[a[2]][f])
    B.append(G[a[2]][f])
    IB.append(IG[a[2]][f])
    if QG[a[2]][f]==3:
        QB.append(QG[a[2]][f]+1)
    elif QG[a[2]][f]==5:
        QB.append(QG[a[2]][f]-1)
    else:
        if abs(G[a[2]][f])==1:
            QB.append(QG[a[2]][f]-1)
        elif abs(G[a[2]][f])==2:
            QB.append(QG[a[2]][f]+1)
        else:
            QB.append(QG[a[2]][f])
    e+=1
    f-=1
f=len(G[a[2]])-1
while f>a[3]:
    G[a[0]].insert(e,G[a[2]][f])
    IG[a[0]].insert(e,IG[a[2]][f])
    if QG[a[2]][f]==3:
        QG[a[0]].insert(e,QG[a[2]][f]+1)
    elif QG[a[2]][f]==5:
        QG[a[0]].insert(e,QG[a[2]][f]-1)
    else:
        if abs(G[a[2]][f])==1:
            QG[a[0]].insert(e,QG[a[2]][f]-1)
        elif abs(G[a[2]][f])==2:
            QG[a[0]].insert(e,QG[a[2]][f]+1)
        else:
            QG[a[0]].insert(e,QG[a[2]][f])
    B.append(G[a[2]][f])
    IB.append(IG[a[2]][f])
    if QG[a[2]][f]==3:
        QB.append(QG[a[2]][f]+1)
    elif QG[a[2]][f]==5:
        QB.append(QG[a[2]][f]-1)
    else:
        if abs(G[a[2]][f])==1:
            QB.append(QG[a[2]][f]-1)
        elif abs(G[a[2]][f])==2:
            QB.append(QG[a[2]][f]+1)
        else:
            QB.append(QG[a[2]][f])
    e+=1
    f-=1
G.pop(a[2])
IG.pop(a[2])
QG.pop(a[2])
U=KAPPA[z2][3]
V=copy_2d(U)
va=0
VA=[]
while va<len(B):
    if abs(B[va])<1:
        VA.append(int(100*B[va]))
    va+=1
VB=[]
VC=[]
while len(VA)>0:
    vb=VA[0]
    if vb<0:
```

```
                                vb=(-1)*vb
                            VA.pop(0)
                            if vb in VA:
                                VB.append(vb)
                            elif -vb in VA:
                                VB.append(vb)
                            else:
                                VC.append(vb)
                        VD=list(set(VC)-set(VB))
                        va=0
                        while va<len(VD):
                            i=VD[va]-1
                            V[i]=V[i]*(-1)
                            va+=1
                        if KAPPA[z2][3][u]>0:
                            KAPPA.insert(z2+1,[1,F,1000,U,IF,QF,XF])
                            KAPPA.insert(z2+2,[-1,G,1000,V,IG,QG,XG])
                        else:
                            KAPPA.insert(z2+1,[1,G,1000,V,IG,QG,XG])
                            KAPPA.insert(z2+2,[-1,F,1000,U,IF,QF,XF])
                        KAPPA[z2+1][0]=KAPPA[z2+1][0]+KAPPA[z2][0]
                        KAPPA[z2+2][0]=KAPPA[z2+2][0]+KAPPA[z2][0]
                        KAPPA[z2+1][2]=int(KAPPA[z2+1][2]*KAPPA[z2][2]/1000)
                        KAPPA[z2+2][2]=int(KAPPA[z2+2][2]*KAPPA[z2][2]/1000)
                        KAPPA.pop(z2)
                        z2+=1
                    z2+=1
                s+=1
                u+=1

            crescendo=0
            while crescendo<len(KAPPA):
                jay=0
                DEX=[]
                while jay<len(KAPPA[crescendo][4]):
                    jj=0
                    while jj<len(KAPPA[crescendo][1][jay]):
                        if abs(KAPPA[crescendo][1][jay][jj])==1:
                            DEX.append(KAPPA[crescendo][4][jay][jj])
                        jj+=1
                    jay+=1
                babygary=0
                while babygary<len(KAPPA[crescendo][1]):
                    bg=-1
                    while bg<len(KAPPA[crescendo][1][babygary])-1:
                        if abs(KAPPA[crescendo][1][babygary][bg])==1 and
KAPPA[crescendo][1][babygary][bg]==KAPPA[crescendo][1][babygary][bg+1]:
                            remise=0
                            smol=min(KAPPA[crescendo][4][babygary]
[bg],KAPPA[crescendo][4][babygary][bg+1])
                            lorg=max(KAPPA[crescendo][4][babygary]
[bg],KAPPA[crescendo][4][babygary][bg+1])
                            john=smol+1
                            while john<lorg:
                                dex=0
                                while dex<len(DEX):
                                    if DEX[dex]==john:
                                        remise=1
                                    dex+=1
                                john+=1
                            if remise==0:
                                del KAPPA[crescendo][1][babygary][bg+1]
                            del KAPPA[crescendo][4][babygary][bg+1]
                            del KAPPA[crescendo][5][babygary][bg+1]
                            del KAPPA[crescendo][1][babygary][bg]
                            del KAPPA[crescendo][4][babygary][bg]
                            del KAPPA[crescendo][5][babygary][bg]
                            bg=-1
                            d=0
                            while d<len(DEX):
                                if DEX[d]==smol or DEX[d]==lorg:
                                    del DEX[d]
                                else:
                                    d+=1
                        else:
                            bg+=1
                    else:
                        bg+=1

                babygary+=1
            crescendo+=1

        powerz=Z[z][0]
        signz=Z[z][2]

        kappa=0
        while len(KAPPA)>0:
            gary=len(KAPPA)-1
            food=KAPPA[gary]
            KAPPA.pop()
            Z.insert(z,food)
            Z[z][0]=Z[z][0]+powerz
            Z[z][2]=int(Z[z][2]*signz/1000)

        del Z[z+4]

    job=z
    rob=min(len(Z),z+4)
    while job<rob:
        sob=0
        while sob<len(Z[job][1]):
            decide=0
            if len(Z[job][1][sob])==0:
                Z[job][6][4]=Z[job][6][4]+1
                del Z[job][1][sob]
                del Z[job][4][sob]
                del Z[job][5][sob]
                sob=0
            elif len(Z[job][1][sob])==2 and Z[job][1][sob][0]==-Z[job][1][sob][1]:
                if abs(Z[job][1][sob][0])==1:
                    Z[job][6][0]=Z[job][6][0]+1
                elif abs(Z[job][1][sob][0])==2:
                    Z[job][6][2]=Z[job][6][2]+1
                del Z[job][1][sob]
                del Z[job][4][sob]
                del Z[job][5][sob]
                sob=0

            elif len(Z[job][1][sob])==4:
                plutop=1
                rosa=0
                redover=0
                redunder=0
                blueover=0
```

```
                    blueunder=0
                    while rosa<4:
                        if Z[job][1][sob][rosa]==1:
                            redover=Z[job][1][sob][rosa]
                            iredover=Z[job][4][sob][rosa]
                            qredover=Z[job][5][sob][rosa]
                        elif Z[job][1][sob][rosa]==-1:
                            redunder=Z[job][1][sob][rosa]
                            iredunder=Z[job][4][sob][rosa]
                            qredunder=Z[job][5][sob][rosa]
                        elif Z[job][1][sob][rosa]==2:
                            blueover=Z[job][1][sob][rosa]
                            iblueover=Z[job][4][sob][rosa]
                            qblueover=Z[job][5][sob][rosa]
                        elif Z[job][1][sob][rosa]==-2:
                            blueunder=Z[job][1][sob][rosa]
                            iblueunder=Z[job][4][sob][rosa]
                            qblueunder=Z[job][5][sob][rosa]
                        rosa+=1
                    if abs(redover)>0 and abs(redunder)>0:
                        imaxred=max(iredover,iredunder)
                        iminred=min(iredover,iredunder)
                        rosa=iminred+1
                        while rosa<imaxred:
                            coza=0
                            while coza<len(Z[job][4]):
                                sosa=0
                                while sosa<len(Z[job][4][coza]):
                                    if Z[job][4][coza][sosa]==rosa:
                                        if abs(Z[job][1][coza][sosa])==1:
                                            plutop=0
                                    sosa+=1
                                coza+=1
                            rosa+=1
                    if abs(blueover)>0 and abs(blueunder)>0:
                        imaxblue=max(iblueover,iblueunder)
                        iminblue=min(iblueover,iblueunder)
                        rosa=iminblue+1
                        while rosa<imaxblue:
                            coza=0
                            while coza<len(Z[job][4]):
                                sosa=0
                                while sosa<len(Z[job][4][coza]):
                                    if Z[job][4][coza][sosa]==rosa:
                                        if abs(Z[job][1][coza][sosa])==2:
                                            plutop=0
                                    sosa+=1
                                coza+=1
                            rosa+=1

                    if abs(Z[job][1][sob][0])==abs(Z[job][1][sob][1]) and abs(Z[job]
[1][sob][1])==abs(Z[job][1][sob][2]) and abs(Z[job][1][sob][2])==abs(Z[job][1][sob]
[3]):
                        decide=0
                    elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob]
[2]==-Z[job][1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==Z[job][1][sob]
[2]/abs(Z[job][1][sob][2]):
                        decide=1
                    elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob]
[1]==-Z[job][1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==Z[job][1][sob]
[1]/abs(Z[job][1][sob][1]):
                        decide=1
                    elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob]
[2]==-Z[job][1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==-Z[job][1][sob]
[3]/abs(Z[job][1][sob][3]):
                        decide=2
                    elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob]
[1]==-Z[job][1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==-Z[job][1][sob]
[2]/abs(Z[job][1][sob][2]):
                        decide=2
                    elif Z[job][1][sob][0]==Z[job][1][sob][3] and Z[job][1][sob][1]==-
Z[job][1][sob][2]:
                        decide=3

                    if decide>0 and plutop==1:
                        b=[Z[job][4][sob][0],Z[job][4][sob][1],Z[job][4][sob][2],
Z[job][4][sob][3]]
                        bmax=max(b)
                        bmin=min(b)

                        dmax=[]
                        bay=0
                        while bay<len(Z[job][4]):
                            if len(Z[job][4][bay])>0:
                                dmax.append(max(Z[job][4][bay]))
                            bay+=1
                        decidemax=max(dmax)
                        dmin=[]
                        bay=0
                        while bay<len(Z[job][4]):
                            if len(Z[job][4][bay])>0:
                                dmin.append(min(Z[job][4][bay]))
                            bay+=1
                        decidemin=min(dmin)

                        if decidemax==bmax:
                            if decide ==1:
                                Z[job][6][1]=Z[job][6][1]+1
                            elif decide ==2:
                                call=0
                                while call<len(Z[job][1][sob]):
                                    if Z[job][1][sob][call]==1:
                                        posguy=Z[job][4][sob][call]
                                    elif Z[job][1][sob][call]==-1:
                                        negguy=Z[job][4][sob][call]
                                    call+=1
                                if posguy>negguy:
                                    Z[job][6][3]=Z[job][6][3]+1
                                    Z[job][6][5]=Z[job][6][5]+1
                            elif decide ==3:
                                if abs(Z[job][1][sob][1])==1:
                                    Z[job][6][0]=Z[job][6][0]+1
                                elif abs(Z[job][1][sob][1])==2:
                                    Z[job][6][2]=Z[job][6][2]+1
                            del Z[job][1][sob]
                            del Z[job][4][sob]
                            del Z[job][5][sob]
                            sob=0
                        elif decidemin==bmin:
                            if decide==1:
                                Z[job][6][1]=Z[job][6][1]+1
                            elif decide==2:
                                call=0
                                while call<len(Z[job][1][sob]):
```

```
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                else:
                    sob+=1
            else:
                sob+=1
        else:
            sob+=1
    job+=1

ONES=[]
app=0
while app<len(Z[z][1]):
    if len(Z[z][1][app])<=2:
        app+=1
    sapp=0
    while app<len(Z[z][1]) and sapp<len(Z[z][1][app]):
        if abs(Z[z][1][app][sapp])==1:
            ONES.append([Z[z][1][app][sapp],Z[z][4][app][sapp],app,sapp])
        sapp+=1
    app+=1
POS=[]
NEG=[]
BOT=[]
one=0
while one<len(ONES):

    BOT.append(ONES[one][1])

    if ONES[one][0]==1:
        POS.append(ONES[one][1])
    else:
        NEG.append(ONES[one][1])
    one+=1

if len(POS)>0:
    minpos=min(POS)
    maxpos=max(POS)

if len(NEG)>0:
    minneg=min(NEG)
    maxneg=max(NEG)

if len(BOT)>0:
    maxbot=max(BOT)
    minbot=min(BOT)
```

```
        while len(POS)>0 and minbot==minpos:
            if len(BOT)>0:
                BOT.remove(minbot)
            if len(POS)>0:
                POS.remove(minpos)
            if len(BOT)>0:
                minbot=min(BOT)
            if len(POS)>0:
                minpos=min(POS)
    z+=1

crescendo=0
while crescendo<len(Z):
    jay=0
    DEX=[]
    while jay<len(Z[crescendo][4]):
        jj=0
        while jj<len(Z[crescendo][1][jay]):
            if abs(Z[crescendo][1][jay][jj])==1:
                DEX.append(Z[crescendo][4][jay][jj])
            jj+=1
        jay+=1

    babygary=0
    while babygary<len(Z[crescendo][1]):
        bg=-1
        while bg<len(Z[crescendo][1][babygary])-1:
            if abs(Z[crescendo][1][babygary][bg])==1 and Z[crescendo][1][babygary]
[bg]==Z[crescendo][1][babygary][bg+1]:
                remise=0
                smol=min(Z[crescendo][4][babygary][bg],Z[crescendo][4][babygary]
[bg+1])
                lorg=max(Z[crescendo][4][babygary][bg],Z[crescendo][4][babygary]
[bg+1])
                john=smol+1
                while john<lorg:
                    dex=0
                    while dex<len(DEX):
                        if DEX[dex]==john:
                            remise=1
                        dex+=1
                    john+=1
                if remise==0:
                    del Z[crescendo][1][babygary][bg+1]
                    del Z[crescendo][4][babygary][bg+1]
                    del Z[crescendo][5][babygary][bg+1]
                    del Z[crescendo][1][babygary][bg]
                    del Z[crescendo][4][babygary][bg]
                    del Z[crescendo][5][babygary][bg]
                    bg=-1
                    d=0
                    while d<len(DEX):
                        if DEX[d]==smol or DEX[d]==lorg:
                            del DEX[d]
                        else:
                            d+=1
                else:
                    bg+=1
            else:
                bg+=1
        babygary+=1
    crescendo+=1

job=0
while job<len(Z):
    sob=0
    while sob<len(Z[job][1]):
        decide=0
        if len(Z[job][1][sob])==0:
            Z[job][6][4]=Z[job][6][4]+1
            del Z[job][1][sob]
            del Z[job][4][sob]
            del Z[job][5][sob]
            sob=0
        elif len(Z[job][1][sob])==2 and Z[job][1][sob][0]==-Z[job][1][sob][1]:
            if abs(Z[job][1][sob][0])==1:
                Z[job][6][0]=Z[job][6][0]+1
            elif abs(Z[job][1][sob][0])==2:
                Z[job][6][2]=Z[job][6][2]+1
            del Z[job][1][sob]
            del Z[job][4][sob]
            del Z[job][5][sob]
            sob=0

        elif len(Z[job][1][sob])==4:
            plutop=1
            rosa=0
            redover=0
            redunder=0
            blueover=0
            blueunder=0
            while rosa<4:
                if Z[job][1][sob][rosa]==1:
                    redover=Z[job][1][sob][rosa]
                    iredover=Z[job][4][sob][rosa]
                    qredover=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==-1:
                    redunder=Z[job][1][sob][rosa]
                    iredunder=Z[job][4][sob][rosa]
                    qredunder=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==2:
                    blueover=Z[job][1][sob][rosa]
                    iblueover=Z[job][4][sob][rosa]
                    qblueover=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==-2:
                    blueunder=Z[job][1][sob][rosa]
                    iblueunder=Z[job][4][sob][rosa]
                    qblueunder=Z[job][5][sob][rosa]
                rosa+=1
            if abs(redover)>0 and abs(redunder)>0:
                imaxred=max(iredover,iredunder)
                iminred=min(iredover,iredunder)
                rosa=iminred+1
                while rosa<imaxred:
                    coza=0
                    while coza<len(Z[job][4]):
                        sosa=0
                        while sosa<len(Z[job][4][coza]):
                            if Z[job][4][coza][sosa]==rosa:
                                if abs(Z[job][1][coza][sosa])==1:
                                    plutop=0
                            sosa+=1
                        coza+=1
                    rosa+=1
            if abs(blueover)>0 and abs(blueunder)>0:
                imaxblue=max(iblueover,iblueunder)
                iminblue=min(iblueover,iblueunder)
                rosa=iminblue+1
                while rosa<imaxblue:
                    coza=0
                    while coza<len(Z[job][4]):
                        sosa=0
                        while sosa<len(Z[job][4][coza]):
                            if Z[job][4][coza][sosa]==rosa:
                                if abs(Z[job][1][coza][sosa])==2:
                                    plutop=0
                            sosa+=1
                        coza+=1
                    rosa+=1

            if abs(Z[job][1][sob][0])==abs(Z[job][1][sob][1]) and abs(Z[job][1][sob]
[1])==abs(Z[job][1][sob][2]) and abs(Z[job][1][sob][2])==abs(Z[job][1][sob][3]):
                decide=0
            elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob][2]==-Z[job]
[1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==Z[job][1][sob][2]/abs(Z[job]
[1][sob][2]):
                decide=1
            elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==Z[job][1][sob][1]/abs(Z[job]
[1][sob][1]):
                decide=1
            elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob][2]==-Z[job]
[1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==-Z[job][1][sob][3]/
abs(Z[job][1][sob][3]):
                decide=2
            elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==-Z[job][1][sob][2]/
abs(Z[job][1][sob][2]):
                decide=2
            elif Z[job][1][sob][0]==Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2]:
                decide=3

            if decide>0 and plutop==1:
                b=[Z[job][4][sob][0],Z[job][4][sob][1],Z[job][4][sob][2], Z[job][4]
[sob][3]]
                bmax=max(b)
                bmin=min(b)

                dmax=[]
                bay=0
                while bay<len(Z[job][4]):
                    if len(Z[job][4][bay])>0:
                        dmax.append(max(Z[job][4][bay]))
                    bay+=1
                decidemax=max(dmax)
                dmin=[]
                bay=0
                while bay<len(Z[job][4]):
                    if len(Z[job][4][bay])>0:
                        dmin.append(min(Z[job][4][bay]))
                    bay+=1
                decidemin=min(dmin)
                if decidemax==bmax:
                    if decide ==1:
                        Z[job][6][1]=Z[job][6][1]+1
                    if decide ==2:
                        call=0
                        while call<len(Z[job][1][sob]):
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                elif decidemin==bmin:
                    if decide==1:
                        Z[job][6][1]=Z[job][6][1]+1
                    elif decide==2:
                        call=0
                        while call<len(Z[job][1][sob]):
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                else:
                    sob+=1
            else:
                sob+=1
        else:
            sob+=1
    job+=1

crescendo=0
while crescendo<len(Z):
    jay=0
    DEX=[]
    while jay<len(Z[crescendo][4]):
        jj=0
        while jj<len(Z[crescendo][1][jay]):
            if abs(Z[crescendo][1][jay][jj])==2:
                DEX.append(Z[crescendo][4][jay][jj])
            jj+=1
        jay+=1

    babygary=0
    while babygary<len(Z[crescendo][1]):
        bg=-1
        while bg<len(Z[crescendo][1][babygary])-1:
            if abs(Z[crescendo][1][babygary][bg])==2 and Z[crescendo][1][babygary]
[bg]==Z[crescendo][1][babygary][bg+1]:
                remise=0
                smol=min(Z[crescendo][4][babygary][bg],Z[crescendo][4][babygary]
[bg+1])
                lorg=max(Z[crescendo][4][babygary][bg],Z[crescendo][4][babygary]
[bg+1])
                john=smol+1
                while john<lorg:
                    dex=0
                    while dex<len(DEX):
                        if DEX[dex]==john:
                            remise=1
                        dex+=1
                    john+=1
                if remise==0:
                    del Z[crescendo][1][babygary][bg+1]
                    del Z[crescendo][4][babygary][bg+1]
                    del Z[crescendo][5][babygary][bg+1]
                    del Z[crescendo][1][babygary][bg]
                    del Z[crescendo][4][babygary][bg]
                    del Z[crescendo][5][babygary][bg]
                    bg=-1
                    d=0
                    while d<len(DEX):
                        if DEX[d]==smol or DEX[d]==lorg:
                            del DEX[d]
                        else:
                            d+=1
                else:
                    bg+=1
            else:
                bg+=1

        babygary+=1
    crescendo+=1


job=0
while job<len(Z):
    sob=0
    while sob<len(Z[job][1]):
        decide=0
        if len(Z[job][1][sob])==0:
            Z[job][6][4]=Z[job][6][4]+1
            del Z[job][1][sob]
            del Z[job][4][sob]
            del Z[job][5][sob]
            sob=0
        elif len(Z[job][1][sob])==2 and Z[job][1][sob][0]==-Z[job][1][sob][1]:
            if abs(Z[job][1][sob][0])==1:
                Z[job][6][0]=Z[job][6][0]+1
            elif abs(Z[job][1][sob][0])==2:
                Z[job][6][2]=Z[job][6][2]+1
            del Z[job][1][sob]
            del Z[job][4][sob]
            del Z[job][5][sob]
            sob=0

        elif len(Z[job][1][sob])==4:
            plutop=1
            rosa=0
            redover=0
            redunder=0
            blueover=0
            blueunder=0
            while rosa<4:
                if Z[job][1][sob][rosa]==1:
                    redover=Z[job][1][sob][rosa]
                    iredover=Z[job][4][sob][rosa]
                    qredover=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==-1:
                    redunder=Z[job][1][sob][rosa]
                    iredunder=Z[job][4][sob][rosa]
                    qredunder=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==2:
                    blueover=Z[job][1][sob][rosa]
                    iblueover=Z[job][4][sob][rosa]
                    qblueover=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==-2:
                    blueunder=Z[job][1][sob][rosa]
                    iblueunder=Z[job][4][sob][rosa]
                    qblueunder=Z[job][5][sob][rosa]
                rosa+=1
            if abs(redover)>0 and abs(redunder)>0:
                imaxred=max(iredover,iredunder)
                iminred=min(iredover,iredunder)
                rosa=iminred+1
                while rosa<imaxred:
                    coza=0
                    while coza<len(Z[job][4]):
                        sosa=0
                        while sosa<len(Z[job][4][coza]):
                            if Z[job][4][coza][sosa]==rosa:
                                if abs(Z[job][1][coza][sosa])==1:
                                    plutop=0
                            sosa+=1
                        coza+=1
                    rosa+=1

            if abs(blueover)>0 and abs(blueunder)>0:
                imaxblue=max(iblueover,iblueunder)
                iminblue=min(iblueover,iblueunder)
                rosa=iminblue+1
                while rosa<imaxblue:
                    coza=0
                    while coza<len(Z[job][4]):
                        sosa=0
                        while sosa<len(Z[job][4][coza]):
                            if Z[job][4][coza][sosa]==rosa:
                                if abs(Z[job][1][coza][sosa])==2:
                                    plutop=0
                            sosa+=1
                        coza+=1
                    rosa+=1

            if abs(Z[job][1][sob][0])==abs(Z[job][1][sob][1]) and abs(Z[job][1][sob]
[1])==abs(Z[job][1][sob][2]) and abs(Z[job][1][sob][2])==abs(Z[job][1][sob][3]):
                decide=0
            elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob][2]==-Z[job]
[1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==Z[job][1][sob][2]/abs(Z[job]
[1][sob][2]):
                decide=1
            elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==Z[job][1][sob][1]/abs(Z[job]
[1][sob][1]):
                decide=1
            elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob][2]==-Z[job]
[1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==-Z[job][1][sob][3]/
abs(Z[job][1][sob][3]):
                decide=2
            elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==-Z[job][1][sob][2]/
abs(Z[job][1][sob][2]):
                decide=2
            elif Z[job][1][sob][0]==Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2]:
                decide=3

            if decide>0 and plutop==1:
                b=[Z[job][4][sob][0],Z[job][4][sob][1],Z[job][4][sob][2], Z[job][4]
[sob][3]]
                bmax=max(b)
                bmin=min(b)
                dmax=[]
                bay=0
                while bay<len(Z[job][4]):
                    if len(Z[job][4][bay])>0:
                        dmax.append(max(Z[job][4][bay]))
                    bay+=1
                decidemax=max(dmax)
                dmin=[]
                bay=0
                while bay<len(Z[job][4]):
                    if len(Z[job][4][bay])>0:
                        dmin.append(min(Z[job][4][bay]))
                    bay+=1
                decidemin=min(dmin)

                if decidemax==bmax:
                    if decide ==1:
                        Z[job][6][1]=Z[job][6][1]+1
                    if decide ==2:
                        call=0
                        while call<len(Z[job][1][sob]):
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                elif decidemin==bmin:
                    if decide==1:
                        Z[job][6][1]=Z[job][6][1]+1
                    elif decide==2:
                        call=0
                        while call<len(Z[job][1][sob]):
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                else:
                    sob+=1
            else:
                sob+=1
        else:
            sob+=1
    job+=1

z=0
while z<len(Z):
    print("z=",z)
    TWOS=[]
    r=0
    while r<len(Z[z][1]):
        if len(Z[z][1][r])<=2:
            if len(Z[z][1][r])==0 or Z[z][1][r][0]==Z[z][1][r][1]:
                Z[z][6][4]=Z[z][6][4]+1
            elif abs(Z[z][1][r][0])==1:
                Z[z][6][0]=Z[z][6][0]+1
            elif abs(Z[z][1][r][0])==2:
                Z[z][6][2]=Z[z][6][2]+1
            del Z[z][1][r]
            del Z[z][4][r]
            del Z[z][5][r]
        s=0
        while r<len(Z[z][1]) and s<len(Z[z][1][r]):
            if abs(Z[z][1][r][s])==2:
                TWOS.append([Z[z][1][r][s],Z[z][4][r][s],r,s])
            s+=1
        r+=1
    POS=[]
    NEG=[]
    BOT=[]
    one=0
    while one<len(TWOS):

        BOT.append(TWOS[one][1])

        if TWOS[one][0]==2:
            POS.append(TWOS[one][1])
        else:
            NEG.append(TWOS[one][1])

        one+=1

    if len(POS)>0:
        minpos=min(POS)
        maxpos=max(POS)

    if len(NEG)>0:
        minneg=min(NEG)
        maxneg=max(NEG)

    if len(BOT)>0:
        maxbot=max(BOT)
        minbot=min(BOT)

    while len(POS)>0:

        rancid=0
        while len(POS)>0 and minbot==minpos and rancid==0:
            if len(BOT)>0:
                BOT.remove(minbot)
            if len(POS)>0:
                POS.remove(minpos)
            if len(BOT)>0:
                minbot=min(BOT)
            if len(POS)>0:
                minpos=min(POS)
            else:
                rancid=1

        jo=0

        if rancid==0:
            NEGG=list.copy(NEG)
            if len(NEGG)>0:
                nextneg=max(NEGG)
                jo=1
            while jo==1 and nextneg>minpos:
                NEGG.remove(nextneg)
                if len(NEGG)>0:
                    nextneg=max(NEGG)
                else:
                    jo=0
            run=0
            while run<len(TWOS):
                if TWOS[run][1]==minpos:
                    k1=TWOS[run][2]
                    k2=TWOS[run][3]
                elif TWOS[run][1]==nextneg:
                    q1=TWOS[run][2]
                q2=TWOS[run][3]
            run+=1
    quorren=0
    while quorren<len(Z):
        quorren+=1

    if rancid==0:

        bob=Z[z][4][k1][k2]
        Z[z][4][k1][k2]=Z[z][4][q1][q2]
        Z[z][4][q1][q2]=bob
        moe=0

        if abs(k2-q2)==1:
            moe=1

        pos1=k1
        pos2=k2
        neg1=q1
        neg2=q2

        if pos1==neg1 and neg2==0 and pos2==len(Z[z][1][k1])-1:
            curly=Z[z][1][k1][neg2]
            del Z[z][1][k1][neg2]
            Z[z][1][k1].append(curly)
            larry=Z[z][4][k1][neg2]
            del Z[z][4][k1][neg2]
            Z[z][4][k1].append(larry)
            morton=Z[z][5][k1][neg2]
            del Z[z][5][k1][neg2]
            Z[z][5][k1].append(morton)

            neg2=len(Z[z][1][k1])-1
            pos2-=1
            moe=1
        elif pos1==neg1 and pos2==0 and neg2==len(Z[z][1][k1])-1:
            curly=Z[z][1][k1][pos2]
            del Z[z][1][k1][pos2]
            Z[z][1][k1].append(curly)
            larry=Z[z][4][k1][pos2]
            del Z[z][4][k1][pos2]
            Z[z][4][k1].append(larry)
            morton=Z[z][5][k1][pos2]
            del Z[z][5][k1][pos2]
            Z[z][5][k1].append(morton)

            pos2=len(Z[z][1][k1])-1
            neg2-=1
            moe=1
        KAPPA=[]

    if rancid==0 and pos1==neg1 and moe==1:
        if pos2<neg2:
            if Z[z][5][pos1][pos2]<Z[z][5][pos1][neg2]:
                Z[z][3]=[-1]
                Z[z][0]=Z[z][0]+3
                Z[z][2]=-Z[z][2]
            else:
                Z[z][3]=[1]
                Z[z][0]=Z[z][0]-3
                Z[z][2]=-Z[z][2]
        Z[z][1][pos1].insert(pos2,-0.99)
        Z[z][4][pos1].insert(pos2,99)
        Z[z][5][pos1].insert(pos2,99)
        Z[z][1][pos1].insert(neg2+2,0.99)
        Z[z][4][pos1].insert(neg2+2,99)
        Z[z][5][pos1].insert(neg2+2,99)

        a=[pos1,pos2,neg1,neg2+2]
    else:
        if Z[z][5][pos1][pos2]<Z[z][5][pos1][neg2]:
            Z[z][3]=[1]
            Z[z][0]=Z[z][0]-3
            Z[z][2]=-Z[z][2]
        else:
            Z[z][3]=[-1]
            Z[z][0]=Z[z][0]+3
            Z[z][2]=-Z[z][2]

        Z[z][1][pos1].insert(neg2,0.99)
        Z[z][4][pos1].insert(neg2,99)
        Z[z][5][pos1].insert(neg2,99)
        Z[z][1][pos1].insert(pos2+2,-0.99)
        Z[z][4][pos1].insert(pos2+2,99)
        Z[z][5][pos1].insert(pos2+2,99)

        a=[neg1,neg2,pos1,pos2+2]

    E=Z[z][1]
    I=Z[z][4]
    Q=Z[z][5]
    X=Z[z][6]

    O=[]
    U=Z[z][3]
    u=0
    s=0.99

    C=copy_2d(E)
    IC=copy_2d(I)
    XC=copy_2d(X)
    QC=copy_2d(Q)

    A=[]
    IA=[]
    QA=[]

    e=a[1]
    f=a[3]
    while e<f-1:
        A.append(C[a[0]][e+1])
        IA.append(IC[a[0]][e+1])
        QA.append(QC[a[0]][e+1])
        C[a[0]].pop(e+1)
        IC[a[0]].pop(e+1)
        QC[a[0]].pop(e+1)
        f-=1

    C[a[0]].pop(a[1])
    C[a[0]].pop(a[1])
    O.append(IC[a[0]][a[1]])
    IC[a[0]].pop(a[1])
    IC[a[0]].pop(a[1])
```

```
            QC[a[0]].pop(a[1])
            QC[a[0]].pop(a[1])

            D=copy_2d(C)
            ID=copy_2d(IC)
            XD=copy_2d(XC)
            QD=copy_2d(QC)

            C.append(A)
            IC.append(IA)
            QC.append(QA)

            i=0
            while i<len(A):
                D[a[0]].insert(a[1],A[i])
                ID[a[0]].insert(a[1],IA[i])
                if QA[i]==3:
                    QD[a[0]].insert(a[1],QA[i]+1)
                elif QA[i]==5:
                    QD[a[0]].insert(a[1],QA[i]-1)
                else:
                    if abs(A[i])==1:
                        QD[a[0]].insert(a[1],QA[i]-1)
                    elif abs(A[i])==2:
                        QD[a[0]].insert(a[1],QA[i]+1)
                    else:
                        QD[a[0]].insert(a[1],QA[i])
                i+=1

            V=copy_2d(U)

            va=0
            VA=[]
            while va<len(A):
                if abs(A[va])<1:
                    VA.append(int(100*A[va]))
                va+=1

            VB=[]
            VC=[]
            while len(VA)>0:
                vb=VA[0]
                if vb<0:
                    vb=(-1)*vb
                VA.pop(0)
                if vb in VA:
                    VB.append(vb)
                elif -vb in VA:
                    VB.append(vb)
                else:
                    VC.append(vb)
            VD=list(set(VC)-set(VB))

            va=0
            while va<len(VD):
                i=VD[va]-1
                V[i]=V[i]*(-1)
                va+=1

            if U[u]>0:
                Z.insert(z+1,[1,C,1000,U,IC,QC,XC])
                Z.insert(z+2,[-1,D,1000,V,ID,QD,XD])
            else:
                Z.insert(z+1,[1,D,1000,V,ID,QD,XD])
                Z.insert(z+2,[-1,C,1000,U,IC,QC,XC])
            hay=0
            REX=[]
            while hay<len(Z[z+1][1]):
                hh=0
                while hh<len(Z[z+1][1][hay]):
                    if abs(Z[z+1][1][hay][hh])==2:
                        REX.append(Z[z+1][4][hay][hh])
                    hh+=1
                hay+=1
            babyhary=0
            while babyhary<len(Z[z+1][1]):
                bh=-1
                while bh<len(Z[z+1][1][babyhary])-1:
                    if abs(Z[z+1][1][babyhary][bh])==2 and Z[z+1][1][babyhary]
[bh]==Z[z+1][1][babyhary][bh+1]:
                        demise=0
                        smolg=min(Z[z+1][4][babyhary][bh],Z[z+1][4][babyhary][bh+1])
                        lorgg=max(Z[z+1][4][babyhary][bh],Z[z+1][4][babyhary][bh+1])
                        johhn=smolg+1
                        while johhn<lorgg:
                            rex=0
                            while rex<len(REX):
                                if REX[rex]==johhn:
                                    demise=1
                                rex+=1
                            johhn+=1
                        if demise==0:
                            del Z[z+1][1][babyhary][bh+1]
                            del Z[z+1][4][babyhary][bh+1]
                            del Z[z+1][5][babyhary][bh+1]
                            del Z[z+1][1][babyhary][bh]
                            del Z[z+1][4][babyhary][bh]
                            del Z[z+1][5][babyhary][bh]
                            bh=-1
                            dh=0
                            while dh<len(REX):
                                if REX[dh]==smolg or REX[dh]==lorgg:
                                    del REX[dh]
                                else:
                                    dh+=1
                        else:
                            bh+=1
                    else:
                        bh+=1
                babyhary+=1

            hay=0
            REX=[]
            while hay<len(Z[z+2][1]):
                hh=0
                while hh<len(Z[z+2][1][hay]):
                    if abs(Z[z+2][1][hay][hh])==2:
                        REX.append(Z[z+2][4][hay][hh])
                    hh+=1
                hay+=1
            babyhary=0
            while babyhary<len(Z[z+2][1]):
                bh=-1
                while bh<len(Z[z+2][1][babyhary])-1:
                    if abs(Z[z+2][1][babyhary][bh])==2 and Z[z+2][1][babyhary]
[bh]==Z[z+2][1][babyhary][bh+1]:
                        demise=0
                        smolg=min(Z[z+2][4][babyhary][bh],Z[z+2][4][babyhary][bh+1])
                        lorgg=max(Z[z+2][4][babyhary][bh],Z[z+2][4][babyhary][bh+1])
                        johhn=smolg+1
                        while johhn<lorgg:
                            rex=0
                            while rex<len(REX):
                                if REX[rex]==johhn:
                                    demise=1
                                rex+=1
                            johhn+=1
                        if demise==0:
                            del Z[z+2][1][babyhary][bh+1]
                            del Z[z+2][4][babyhary][bh+1]
                            del Z[z+2][5][babyhary][bh+1]
                            del Z[z+2][1][babyhary][bh]
                            del Z[z+2][4][babyhary][bh]
                            del Z[z+2][5][babyhary][bh]
                            bh=-1
                            dh=0
                            while dh<len(REX):
                                if REX[dh]==smolg or REX[dh]==lorgg:
                                    del REX[dh]
                                else:
                                    dh+=1
                        else:
                            bh+=1
                    else:
                        bh+=1
                babyhary+=1

            Z[z+1][0]=Z[z+1][0]+Z[z][0]
            Z[z+2][0]=Z[z+2][0]+Z[z][0]
            Z[z+1][2]=int(Z[z+1][2]*Z[z][2]/1000)
            Z[z+2][2]=int(Z[z+2][2]*Z[z][2]/1000)
            Z.pop(z)

            u+=1
        elif rancid==0:
            if k1==q1:
                if pos2<neg2:
                    if Z[z][5][k1][pos2]>Z[z][5][k1][neg2]:
                        Z[z][3]=[-1,1]
                        Z[z][1][k1].insert(pos2,0.02)
                        Z[z][4][k1].insert(pos2,92)
                        Z[z][5][k1].insert(pos2,92)
                        Z[z][1][k1].insert(pos2+2,0.01)
                        Z[z][4][k1].insert(pos2+2,91)
                        Z[z][5][k1].insert(pos2+2,91)
                        Z[z][1][q1].insert(neg2+2,-0.01)
                        Z[z][4][q1].insert(neg2+2,91)
                        Z[z][5][q1].insert(neg2+2,91)
                        Z[z][1][q1].insert(neg2+4,-0.02)
                        Z[z][4][q1].insert(neg2+4,92)
                        Z[z][5][q1].insert(neg2+4,92)

                    else:
                        Z[z][3]=[-1,1]
                        Z[z][1][k1].insert(pos2,0.01)
                        Z[z][4][k1].insert(pos2,91)
                        Z[z][5][k1].insert(pos2,91)
                        Z[z][1][k1].insert(pos2+2,0.02)
                        Z[z][4][k1].insert(pos2+2,92)
                        Z[z][5][k1].insert(pos2+2,92)
                        Z[z][1][q1].insert(neg2+2,-0.02)
                        Z[z][4][q1].insert(neg2+2,92)
                        Z[z][5][q1].insert(neg2+2,92)
                        Z[z][1][q1].insert(neg2+4,-0.01)
                        Z[z][4][q1].insert(neg2+4,91)
                        Z[z][5][q1].insert(neg2+4,91)
                else:
                    if Z[z][5][k1][neg2]>Z[z][5][k1][pos2]:
                        Z[z][3]=[-1,1]
                        Z[z][1][k1].insert(pos2,0.01)
                        Z[z][4][k1].insert(pos2,91)
                        Z[z][5][k1].insert(pos2,91)
                        Z[z][1][k1].insert(pos2+2,0.02)
                        Z[z][4][k1].insert(pos2+2,92)
                        Z[z][5][k1].insert(pos2+2,92)
                        Z[z][1][q1].insert(neg2,-0.02)
                        Z[z][4][q1].insert(neg2,92)
                        Z[z][5][q1].insert(neg2,92)
                        Z[z][1][q1].insert(neg2+2,-0.01)
                        Z[z][4][q1].insert(neg2+2,91)
                        Z[z][5][q1].insert(neg2+2,91)
                    else:
                        Z[z][3]=[-1,1]
                        Z[z][1][k1].insert(pos2,0.02)
                        Z[z][4][k1].insert(pos2,92)
                        Z[z][5][k1].insert(pos2,92)
                        Z[z][1][k1].insert(pos2+2,0.01)
                        Z[z][4][k1].insert(pos2+2,91)
                        Z[z][5][k1].insert(pos2+2,91)
                        Z[z][1][q1].insert(neg2,-0.01)
                        Z[z][4][q1].insert(neg2,91)
                        Z[z][5][q1].insert(neg2,91)
                        Z[z][1][q1].insert(neg2+2,-0.02)
                        Z[z][4][q1].insert(neg2+2,92)
                        Z[z][5][q1].insert(neg2+2,92)
            else:
                if Z[z][5][k1][pos2]>Z[z][5][q1][neg2]:
                    Z[z][3]=[-1,1]
                    Z[z][1][k1].insert(pos2,0.02)
                    Z[z][4][k1].insert(pos2,92)
                    Z[z][5][k1].insert(pos2,92)
                    Z[z][1][k1].insert(pos2+2,0.01)
                    Z[z][4][k1].insert(pos2+2,91)
                    Z[z][5][k1].insert(pos2+2,91)
                    Z[z][1][q1].insert(neg2,-0.01)
                    Z[z][4][q1].insert(neg2,91)
                    Z[z][5][q1].insert(neg2,91)
                    Z[z][1][q1].insert(neg2+2,-0.02)
                    Z[z][4][q1].insert(neg2+2,92)
                    Z[z][5][q1].insert(neg2+2,92)

                elif Z[z][5][q1][neg2]>Z[z][5][k1][pos2]:
                    Z[z][3]=[-1,1]
                    Z[z][1][k1].insert(pos2,0.01)
                    Z[z][4][k1].insert(pos2,91)
                    Z[z][5][k1].insert(pos2,91)
                    Z[z][1][k1].insert(pos2+2,0.02)
                    Z[z][4][k1].insert(pos2+2,92)
            Z[z][5][k1].insert(pos2+2,92)
            Z[z][1][q1].insert(neg2,-0.02)
            Z[z][4][q1].insert(neg2,92)
            Z[z][5][q1].insert(neg2,92)
            Z[z][1][q1].insert(neg2+2,-0.01)
            Z[z][4][q1].insert(neg2+2,91)
            Z[z][5][q1].insert(neg2+2,91)

        elif Z[z][5][k1][pos2]==4 and Z[z][5][q1][neg2]==4:
            Z[z][3]=[1,-1]
            Z[z][1][k1].insert(pos2,0.01)
            Z[z][4][k1].insert(pos2,91)
            Z[z][5][k1].insert(pos2,91)
            Z[z][1][k1].insert(pos2+2,0.02)
            Z[z][4][k1].insert(pos2+2,92)
            Z[z][5][k1].insert(pos2+2,92)
            Z[z][1][q1].insert(neg2,-0.01)
            Z[z][4][q1].insert(neg2,91)
            Z[z][5][q1].insert(neg2,91)
            Z[z][1][q1].insert(neg2+2,-0.02)
            Z[z][4][q1].insert(neg2+2,92)
            Z[z][5][q1].insert(neg2+2,92)

        elif Z[z][5][k1][pos2]==5 and Z[z][5][q1][neg2]==5:
            Z[z][3]=[1,-1]
            Z[z][1][k1].insert(pos2,0.02)
            Z[z][4][k1].insert(pos2,92)
            Z[z][5][k1].insert(pos2,92)
            Z[z][1][k1].insert(pos2+2,0.01)
            Z[z][4][k1].insert(pos2+2,91)
            Z[z][5][k1].insert(pos2+2,91)
            Z[z][1][q1].insert(neg2,-0.02)
            Z[z][4][q1].insert(neg2,92)
            Z[z][5][q1].insert(neg2,92)
            Z[z][1][q1].insert(neg2+2,-0.01)
            Z[z][4][q1].insert(neg2+2,91)
            Z[z][5][q1].insert(neg2+2,91)

    E= Z[z][1]
    I= Z[z][4]
    Q= Z[z][5]
    X= Z[z][6]

    O=[]
    S=[0.02]
    U=Z[z][3]
    u=0
    s=0.01
    e=0
    a=[]

    while e<len(E):
        f=0
        while f<len(E[e]):
            if abs(E[e][f])==s:
                a.append(e)
                a.append(f)
                f+=1
            else:
                f+=1
        e+=1
    if a[0]==a[2]:
        C=copy_2d(E)
        IC=copy_2d(I)
        XC=copy_2d(X)
        QC=copy_2d(Q)
        A=[]
        IA=[]
        QA=[]
        e=a[1]
        f=a[3]
        while e<f-1:
            A.append(C[a[0]][e+1])
            IA.append(IC[a[0]][e+1])
            QA.append(QC[a[0]][e+1])
            C[a[0]].pop(e+1)
            IC[a[0]].pop(e+1)
            QC[a[0]].pop(e+1)
            f-=1
        C[a[0]].pop(a[1])
        C[a[0]].pop(a[1])
        O.append(IC[a[0]][a[1]])
        IC[a[0]].pop(a[1])
        IC[a[0]].pop(a[1])
        QC[a[0]].pop(a[1])
        QC[a[0]].pop(a[1])
        D=copy_2d(C)
        ID=copy_2d(IC)
        XD=copy_2d(X)
        QD=copy_2d(QC)
        C.append(A)
        IC.append(IA)
        QC.append(QA)
        i=0
        while i<len(A):
            D[a[0]].insert(a[1],A[i])
            ID[a[0]].insert(a[1],IA[i])
            if QA[i]==3:
                QD[a[0]].insert(a[1],QA[i]+1)
            elif QA[i]==5:
                QD[a[0]].insert(a[1],QA[i]-1)
            else:
                if abs(A[i])==1:
                    QD[a[0]].insert(a[1],QA[i]-1)
                elif abs(A[i])==2:
                    QD[a[0]].insert(a[1],QA[i]+1)
                else:
                    QD[a[0]].insert(a[1],QA[i])
            i+=1
        V=copy_2d(U)
        va=0
        VA=[]
        while va<len(A):
            if abs(A[va])<1:
                VA.append(int(100*A[va]))
            va+=1
        VB=[]
        VC=[]
        while len(VA)>0:
            vb=VA[0]
            if vb<0:
                vb=(-1)*vb
            VA.pop(0)
            if vb in VA:
                    VB.append(vb)
                elif -vb in VA:
                    VB.append(vb)
                else:
                    VC.append(vb)
            VD=list(set(VC)-set(VB))
            va=0
            while va<len(VD):
                i=VD[va]-1
                V[i]=V[i]*(-1)
                va+=1
            if U[u]>0:
                KAPPA=[[1,C,1000,U,IC,QC,XC],[-1,D,1000,V,ID,QD,XD]]
            else:
                KAPPA=[[1,D,1000,V,ID,QD,XD],[-1,C,1000,U,IC,QC,XC]]
            u+=1
        else:
            F=copy_2d(E)
            IF=copy_2d(I)
            XF=copy_2d(X)
            QF=copy_2d(Q)
            G=copy_2d(E)
            IG=copy_2d(I)
            XG=copy_2d(X)
            QG=copy_2d(Q)
            F[a[0]].pop(a[1])
            O.append(IF[a[0]][a[1]])
            IF[a[0]].pop(a[1])
            QF[a[0]].pop(a[1])
            e=a[1]
            f=a[3]+1
            while f<len(F[a[2]]):
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            f=0
            while f<a[3]:
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            F.pop(a[2])
            IF.pop(a[2])
            QF.pop(a[2])
            G[a[0]].pop(a[1])
            IG[a[0]].pop(a[1])
            QG[a[0]].pop(a[1])
            B=[]
            IB=[]
            QB=[]
            e=a[1]
            f=a[3]-1
            while f>=0:
                G[a[0]].insert(e,G[a[2]][f])
                IG[a[0]].insert(e,IG[a[2]][f])
                if QG[a[2]][f]==3:
                    QG[a[0]].insert(e,QG[a[2]][f]+1)
                elif QG[a[2]][f]==5:
                    QG[a[0]].insert(e,QG[a[2]][f]-1)
                    else:
                        if abs(G[a[2]][f])==1:
                            QG[a[0]].insert(e,QG[a[2]][f]-1)
                        elif abs(G[a[2]][f])==2:
                            QG[a[0]].insert(e,QG[a[2]][f]+1)
                        else:
                            QG[a[0]].insert(e,QG[a[2]][f])
                    B.append(G[a[2]][f])
                    IB.append(IG[a[2]][f])
                    if QG[a[2]][f]==3:
                        QB.append(QG[a[2]][f]+1)
                    elif QG[a[2]][f]==5:
                        QB.append(QG[a[2]][f]-1)
                    else:
                        if abs(G[a[2]][f])==1:
                            QB.append(QG[a[2]][f]-1)
                        elif abs(G[a[2]][f])==2:
                            QB.append(QG[a[2]][f]+1)
                        else:
                            QB.append(QG[a[2]][f])
                    e+=1
                    f-=1
                f=len(G[a[2]])-1
                while f>a[3]:
                    G[a[0]].insert(e,G[a[2]][f])
                    IG[a[0]].insert(e,IG[a[2]][f])
                    if QG[a[2]][f]==3:
                        QG[a[0]].insert(e,QG[a[2]][f]+1)
                    elif QG[a[2]][f]==5:
                        QG[a[0]].insert(e,QG[a[2]][f]-1)
                    else:
                        if abs(G[a[2]][f])==1:
                            QG[a[0]].insert(e,QG[a[2]][f]-1)
                        elif abs(G[a[2]][f])==2:
                            QG[a[0]].insert(e,QG[a[2]][f]+1)
                        else:
                            QG[a[0]].insert(e,QG[a[2]][f])
                    B.append(G[a[2]][f])
                    IB.append(IG[a[2]][f])
                    if QG[a[2]][f]==3:
                        QB.append(QG[a[2]][f]+1)
                    elif QG[a[2]][f]==5:
                        QB.append(QG[a[2]][f]-1)
                    else:
                        if abs(G[a[2]][f])==1:
                            QB.append(QG[a[2]][f]-1)
                        elif abs(G[a[2]][f])==2:
                            QB.append(QG[a[2]][f]+1)
                        else:
                            QB.append(QG[a[2]][f])
                    e+=1
                    f-=1
                G.pop(a[2])
                IG.pop(a[2])
                QG.pop(a[2])
                V=copy_2d(U)
                va=0
                VA=[]
                while va<len(B):
                    if abs(B[va])<1:
                        VA.append(int(100*B[va]))
                    va+=1
                VB=[]
                VC=[]
                while len(VA)>0:
                    vb=VA[0]
                    if vb<0:
                        vb=(-1)*vb
                    VA.pop(0)
                    if vb in VA:
                        VB.append(vb)
                    elif -vb in VA:
                        VB.append(vb)
                    else:
                        VC.append(vb)
                VD=list(set(VC)-set(VB))
                va=0
                while va<len(VD):
                    i=VD[va]-1
                    V[i]=V[i]*(-1)
                    va+=1
                if U[u]>0:
                    KAPPA=[[1,F,1000,U,IF,QF,XF],[-1,G,1000,V,IG,QG,XG]]
                else:
                    KAPPA=[[1,G,1000,V,IG,QG,XG],[-1,F,1000,U,IF,QF,XF]]
                u+=1
            s=0
            while s<len(S):
                z2=0
                while z2<len(KAPPA):
                    E=KAPPA[z2][1]
                    I=KAPPA[z2][4]
                    Q=KAPPA[z2][5]
                    X=KAPPA[z2][6]
                    U=KAPPA[z2][3]
                    e=0
                    a=[]
                    while e<len(E):
                        f=0
                        while f<len(E[e]):
                            if abs(E[e][f])==S[s]:
                                a.append(e)
                                a.append(f)
                                f+=1
                            else:
                                f+=1
                        e+=1
                    if a[0]==a[2]:
                        C=copy_2d(E)
                        IC=copy_2d(I)
                        QC=copy_2d(Q)
                        XC=copy_2d(X)
                        A=[]
                        IA=[]
                        QA=[]
                        e=a[1]
                        f=a[3]
                        while e<f-1:
                            A.append(C[a[0]][e+1])
                            IA.append(IC[a[0]][e+1])
                            QA.append(QC[a[0]][e+1])
                            C[a[0]].pop(e+1)
                            IC[a[0]].pop(e+1)
                            QC[a[0]].pop(e+1)
```

```
        f-=1
    C[a[0]].pop(a[1])
    C[a[0]].pop(a[1])
    O.append(IC[a[0]][a[1]])
    IC[a[0]].pop(a[1])
    IC[a[0]].pop(a[1])
    QC[a[0]].pop(a[1])
    QC[a[0]].pop(a[1])
    D=copy_2d(C)
    ID=copy_2d(IC)
    XD=copy_2d(X)
    QD=copy_2d(QC)
    C.append(A)
    IC.append(IA)
    QC.append(QA)
    i=0
    while i<len(A):
        D[a[0]].insert(a[1],A[i])
        ID[a[0]].insert(a[1],IA[i])
        if QA[i]==3:
            QD[a[0]].insert(a[1],QA[i]+1)
        elif QA[i]==5:
            QD[a[0]].insert(a[1],QA[i]-1)
        else:
            if abs(A[i])==1:
                QD[a[0]].insert(a[1],QA[i]-1)
            elif abs(A[i])==2:
                QD[a[0]].insert(a[1],QA[i]+1)
            else:
                QD[a[0]].insert(a[1],QA[i])
        i+=1
    V=copy_2d(U)
    va=0
    VA=[]
    while va<len(A):
        if abs(A[va])<1:
            VA.append(int(100*A[va]))
        va+=1
    VB=[]
    VC=[]
    while len(VA)>0:
        vb=VA[0]
        if vb<0:
            vb=(-1)*vb
        VA.pop(0)
        if vb in VA:
            VB.append(vb)
        elif -vb in VA:
            VB.append(vb)
        else:
            VC.append(vb)
    VD=list(set(VC)-set(VB))
    va=0
    while va<len(VD):
        i=VD[va]-1
        V[i]=V[i]*(-1)
        va+=1
    if KAPPA[z2][3][u]>0:
        KAPPA.insert(z2+1,[1,C,1000,U,IC,QC,XC])
        KAPPA.insert(z2+2,[-1,D,1000,V,ID,QD,XD])
    else:
        KAPPA.insert(z2+1,[1,D,1000,V,ID,QD,XD])
```

```
                KAPPA.insert(z2+2,[-1,C,1000,U,IC,QC,XC])
            KAPPA[z2+1][0]=KAPPA[z2+1][0]+KAPPA[z2][0]
            KAPPA[z2+2][0]=KAPPA[z2+2][0]+KAPPA[z2][0]
            KAPPA[z2+1][2]=int(KAPPA[z2+1][2]*KAPPA[z2][2]/1000)
            KAPPA[z2+2][2]=int(KAPPA[z2+2][2]*KAPPA[z2][2]/1000)
            KAPPA.pop(z2)
            z2+=1
        else:
            F=copy_2d(E)
            IF=copy_2d(I)
            XF=copy_2d(X)
            QF=copy_2d(Q)
            G=copy_2d(E)
            IG=copy_2d(I)
            XG=copy_2d(X)
            QG=copy_2d(Q)
            F[a[0]].pop(a[1])
            O.append(IF[a[0]][a[1]])
            IF[a[0]].pop(a[1])
            QF[a[0]].pop(a[1])
            e=a[1]
            f=a[3]+1
            while f<len(F[a[2]]):
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            f=0
            while f<a[3]:
                F[a[0]].insert(e,F[a[2]][f])
                IF[a[0]].insert(e,IF[a[2]][f])
                QF[a[0]].insert(e,QF[a[2]][f])
                e+=1
                f+=1
            F.pop(a[2])
            IF.pop(a[2])
            QF.pop(a[2])
            G[a[0]].pop(a[1])
            IG[a[0]].pop(a[1])
            QG[a[0]].pop(a[1])
            B=[]
            IB=[]
            QB=[]
            e=a[1]
            f=a[3]-1
            while f>=0:
                G[a[0]].insert(e,G[a[2]][f])
                IG[a[0]].insert(e,IG[a[2]][f])
                if QG[a[2]][f]==3:
                    QG[a[0]].insert(e,QG[a[2]][f]+1)
                elif QG[a[2]][f]==5:
                    QG[a[0]].insert(e,QG[a[2]][f]-1)
                else:
                    if abs(G[a[2]][f])==1:
                        QG[a[0]].insert(e,QG[a[2]][f]-1)
                    elif abs(G[a[2]][f])==2:
                        QG[a[0]].insert(e,QG[a[2]][f]+1)
                    else:
                        QG[a[0]].insert(e,QG[a[2]][f])
                B.append(G[a[2]][f])
                IB.append(IG[a[2]][f])
                if QG[a[2]][f]==3:
                    QB.append(QG[a[2]][f]+1)
                elif QG[a[2]][f]==5:
                    QB.append(QG[a[2]][f]-1)
                else:
                    if abs(G[a[2]][f])==1:
                        QB.append(QG[a[2]][f]-1)
                    elif abs(G[a[2]][f])==2:
                        QB.append(QG[a[2]][f]+1)
                    else:
                        QB.append(QG[a[2]][f])
                e+=1
                f-=1
            f=len(G[a[2]])-1
            while f>a[3]:
                G[a[0]].insert(e,G[a[2]][f])
                IG[a[0]].insert(e,IG[a[2]][f])
                if QG[a[2]][f]==3:
                    QG[a[0]].insert(e,QG[a[2]][f]+1)
                elif QG[a[2]][f]==5:
                    QG[a[0]].insert(e,QG[a[2]][f]-1)
                else:
                    if abs(G[a[2]][f])==1:
                        QG[a[0]].insert(e,QG[a[2]][f]-1)
                    elif abs(G[a[2]][f])==2:
                        QG[a[0]].insert(e,QG[a[2]][f]+1)
                    else:
                        QG[a[0]].insert(e,QG[a[2]][f])
                B.append(G[a[2]][f])
                IB.append(IG[a[2]][f])
                if QG[a[2]][f]==3:
                    QB.append(QG[a[2]][f]+1)
                elif QG[a[2]][f]==5:
                    QB.append(QG[a[2]][f]-1)
                else:
                    if abs(G[a[2]][f])==1:
                        QB.append(QG[a[2]][f]-1)
                    elif abs(G[a[2]][f])==2:
                        QB.append(QG[a[2]][f]+1)
                    else:
                        QB.append(QG[a[2]][f])
                e+=1
                f-=1
            G.pop(a[2])
            IG.pop(a[2])
            QG.pop(a[2])
            U=KAPPA[z2][3]
            V=copy_2d(U)
            va=0
            VA=[]
            while va<len(B):
                if abs(B[va])<1:
                    VA.append(int(100*B[va]))
                va+=1
            VB=[]
            VC=[]
            while len(VA)>0:
                vb=VA[0]
                if vb<0:
                    vb=(-1)*vb
                VA.pop(0)
                if vb in VA:
```

```
                                VB.append(vb)
                            elif -vb in VA:
                                VB.append(vb)
                            else:
                                VC.append(vb)
                        VD=list(set(VC)-set(VB))
                        va=0
                        while va<len(VD):
                            i=VD[va]-1
                            V[i]=V[i]*(-1)
                            va+=1
                        if KAPPA[z2][3][u]>0:
                            KAPPA.insert(z2+1,[1,F,1000,U,IF,QF,XF])
                            KAPPA.insert(z2+2,[-1,G,1000,V,IG,QG,XG])
                        else:
                            KAPPA.insert(z2+1,[1,G,1000,V,IG,QG,XG])
                            KAPPA.insert(z2+2,[-1,F,1000,U,IF,QF,XF])
                        KAPPA[z2+1][0]=KAPPA[z2+1][0]+KAPPA[z2][0]
                        KAPPA[z2+2][0]=KAPPA[z2+2][0]+KAPPA[z2][0]
                        KAPPA[z2+1][2]=int(KAPPA[z2+1][2]*KAPPA[z2][2]/1000)
                        KAPPA[z2+2][2]=int(KAPPA[z2+2][2]*KAPPA[z2][2]/1000)
                        KAPPA.pop(z2)
                        z2+=1
                    z2+=1
                s+=1
                u+=1

            crescendo=0
            while crescendo<len(KAPPA):
                jay=0
                DEX=[]
                while jay<len(KAPPA[crescendo][4]):
                    jj=0
                    while jj<len(KAPPA[crescendo][1][jay]):
                        if abs(KAPPA[crescendo][1][jay][jj])==2:
                            DEX.append(KAPPA[crescendo][4][jay][jj])
                        jj+=1
                    jay+=1

                babygary=0
                while babygary<len(KAPPA[crescendo][1]):
                    bg=-1
                    while bg<len(KAPPA[crescendo][1][babygary])-1:
                        if abs(KAPPA[crescendo][1][babygary][bg])==2 and
KAPPA[crescendo][1][babygary][bg]==KAPPA[crescendo][1][babygary][bg+1]:
                            remise=0
                            smol=min(KAPPA[crescendo][4][babygary]
[bg],KAPPA[crescendo][4][babygary][bg+1])
                            lorg=max(KAPPA[crescendo][4][babygary]
[bg],KAPPA[crescendo][4][babygary][bg+1])
                            john=smol+1
                            while john<lorg:
                                dex=0
                                while dex<len(DEX):
                                    if DEX[dex]==john:
                                        remise=1
                                    dex+=1
                                john+=1
                            if remise==0:
                                del KAPPA[crescendo][1][babygary][bg+1]
                                del KAPPA[crescendo][4][babygary][bg+1]
                                del KAPPA[crescendo][5][babygary][bg+1]
                                del KAPPA[crescendo][1][babygary][bg]
                                del KAPPA[crescendo][4][babygary][bg]
                                del KAPPA[crescendo][5][babygary][bg]
                                bg=-1
                                d=0
                                while d<len(DEX):
                                    if DEX[d]==smol or DEX[d]==lorg:
                                        del DEX[d]
                                    else:
                                        d+=1
                            else:
                                bg+=1
                        else:
                            bg+=1

                    babygary+=1
                crescendo+=1

            powerz=Z[z][0]
            signz=Z[z][2]

            kappa=0
            while len(KAPPA)>0:
                gary=len(KAPPA)-1
                food=KAPPA[gary]
                KAPPA.pop()
                Z.insert(z,food)
                Z[z][0]=Z[z][0]+powerz
                Z[z][2]=int(Z[z][2]*signz/1000)

            del Z[z+4]

        job=z
        rob=min(len(Z),z+4)
        while job<rob:
            sob=0
            while sob<len(Z[job][1]):
                decide=0
                if len(Z[job][1][sob])==0:
                    Z[job][6][4]=Z[job][6][4]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0

                elif len(Z[job][1][sob])==2 and Z[job][1][sob][0]==-Z[job][1][sob][1]:
                    if abs(Z[job][1][sob][0])==1:
                        Z[job][6][0]=Z[job][6][0]+1
                    elif abs(Z[job][1][sob][0])==2:
                        Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0

                elif len(Z[job][1][sob])==4:
                    plutop=1
                    rosa=0
                    redover=0
                    redunder=0
                    blueover=0
                    blueunder=0
                    while rosa<4:
                        if Z[job][1][sob][rosa]==1:
                            redover=Z[job][1][sob][rosa]
                            iredover=Z[job][4][sob][rosa]
                            qredover=Z[job][5][sob][rosa]
                        elif Z[job][1][sob][rosa]==-1:
                            redunder=Z[job][1][sob][rosa]
                            iredunder=Z[job][4][sob][rosa]
                            qredunder=Z[job][5][sob][rosa]
                        elif Z[job][1][sob][rosa]==2:
                            blueover=Z[job][1][sob][rosa]
                            iblueover=Z[job][4][sob][rosa]
                            qblueover=Z[job][5][sob][rosa]
                        elif Z[job][1][sob][rosa]==-2:
                            blueunder=Z[job][1][sob][rosa]
                            iblueunder=Z[job][4][sob][rosa]
                            qblueunder=Z[job][5][sob][rosa]
                        rosa+=1
                    if abs(redover)>0 and abs(redunder)>0:
                        imaxred=max(iredover,iredunder)
                        iminred=min(iredover,iredunder)
                        rosa=iminred+1
                        while rosa<imaxred:
                            coza=0
                            while coza<len(Z[job][4]):
                                sosa=0
                                while sosa<len(Z[job][4][coza]):
                                    if Z[job][4][coza][sosa]==rosa:
                                        if abs(Z[job][1][coza][sosa])==1:
                                            plutop=0
                                    sosa+=1
                                coza+=1
                            rosa+=1

                    if abs(blueover)>0 and abs(blueunder)>0:
                        imaxblue=max(iblueover,iblueunder)
                        iminblue=min(iblueover,iblueunder)
                        rosa=iminblue+1
                        while rosa<imaxblue:
                            coza=0
                            while coza<len(Z[job][4]):
                                sosa=0
                                while sosa<len(Z[job][4][coza]):
                                    if Z[job][4][coza][sosa]==rosa:
                                        if abs(Z[job][1][coza][sosa])==2:
                                            plutop=0
                                    sosa+=1
                                coza+=1
                            rosa+=1

                    if abs(Z[job][1][sob][0])==abs(Z[job][1][sob][1]) and abs(Z[job]
[1][sob][1])==abs(Z[job][1][sob][2]) and abs(Z[job][1][sob][2])==abs(Z[job][1][sob]
[3]):
                        decide=0
                    elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob]
[2]==-Z[job][1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==Z[job][1][sob]
[2]/abs(Z[job][1][sob][2]):
                        decide=1
                    elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob]
[1]==-Z[job][1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==Z[job][1][sob]
[1]/abs(Z[job][1][sob][1]):
                        decide=1
                    elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob]
[2]==-Z[job][1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==-Z[job][1][sob]
[3]/abs(Z[job][1][sob][3]):
                        decide=2
                    elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob]
[1]==-Z[job][1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==-Z[job][1][sob]
[2]/abs(Z[job][1][sob][2]):
                        decide=2
                    elif Z[job][1][sob][0]==Z[job][1][sob][3] and Z[job][1][sob][1]==-
Z[job][1][sob][2]:
                        decide=3

                    if decide>0 and plutop==1:
                        b=[Z[job][4][sob][0],Z[job][4][sob][1],Z[job][4][sob][2],
Z[job][4][sob][3]]
                        bmax=max(b)
                        bmin=min(b)
                        dmax=[]
                        bay=0
                        while bay<len(Z[job][4]):
                            if len(Z[job][4][bay])>0:
                                dmax.append(max(Z[job][4][bay]))
                            bay+=1
                        decidemax=max(dmax)
                        dmin=[]
                        bay=0
                        while bay<len(Z[job][4]):
                            if len(Z[job][4][bay])>0:
                                dmin.append(min(Z[job][4][bay]))
                            bay+=1
                        decidemin=min(dmin)

                        if decidemax==bmax:
                            if decide ==1:
                                Z[job][6][1]=Z[job][6][1]+1
                            elif decide ==2:
                                call=0
                                while call<len(Z[job][1][sob]):
                                    if Z[job][1][sob][call]==1:
                                        posguy=Z[job][4][sob][call]
                                    elif Z[job][1][sob][call]==-1:
                                        negguy=Z[job][4][sob][call]
                                    call+=1
                                if posguy>negguy:
                                    Z[job][6][3]=Z[job][6][3]+1
                                else:
                                    Z[job][6][5]=Z[job][6][5]+1
                            elif decide ==3:
                                if abs(Z[job][1][sob][1])==1:
                                    Z[job][6][0]=Z[job][6][0]+1
                                elif abs(Z[job][1][sob][1])==2:
                                    Z[job][6][2]=Z[job][6][2]+1
                            del Z[job][1][sob]
                            del Z[job][4][sob]
                            del Z[job][5][sob]
                            sob=0
                        elif decidemin==bmin:
                            if decide==1:
                                Z[job][6][1]=Z[job][6][1]+1
                            elif decide==2:
                                call=0
                                while call<len(Z[job][1][sob]):
                                    if Z[job][1][sob][call]==1:
                                        posguy=Z[job][4][sob][call]
                                    elif Z[job][1][sob][call]==-1:
                                        negguy=Z[job][4][sob][call]
                                    call+=1
                                if posguy>negguy:
                                    Z[job][6][3]=Z[job][6][3]+1
                                else:
                                    Z[job][6][5]=Z[job][6][5]+1
                            elif decide==3:
                                if abs(Z[job][1][sob][1])==1:
                                    Z[job][6][0]=Z[job][6][0]+1
                                elif abs(Z[job][1][sob][1])==2:
                                    Z[job][6][2]=Z[job][6][2]+1
                            del Z[job][1][sob]
                            del Z[job][4][sob]
                            del Z[job][5][sob]
                            sob=0
                        else:
                            sob+=1
                    else:
                        sob+=1
                else:
                    sob+=1
            job+=1

        TWOS=[]
        app=0
        while app<len(Z[z][1]):
            if len(Z[z][1][app])<=2:
                app+=1
            sapp=0
            while app<len(Z[z][1]) and sapp<len(Z[z][1][app]):
                if abs(Z[z][1][app][sapp])==2:
                    TWOS.append([Z[z][1][app][sapp],Z[z][4][app][sapp],app,sapp])
                sapp+=1
            app+=1
        POS=[]
        NEG=[]
        BOT=[]
        one=0
        while one<len(TWOS):

            BOT.append(TWOS[one][1])

            if TWOS[one][0]==2:
                POS.append(TWOS[one][1])
            else:
                NEG.append(TWOS[one][1])
            one+=1

        if len(POS)>0:
            minpos=min(POS)
            maxpos=max(POS)

        if len(NEG)>0:
            minneg=min(NEG)
            maxneg=max(NEG)

        if len(BOT)>0:
            maxbot=max(BOT)
            minbot=min(BOT)
        while len(POS)>0 and minbot==minpos:
            if len(BOT)>0:
                BOT.remove(minbot)
            if len(POS)>0:
                POS.remove(minpos)
            if len(BOT)>0:
                minbot=min(BOT)
            if len(POS)>0:
                minpos=min(POS)
    z+=1

crescendo=0
while crescendo<len(Z):
    jay=0
    DEX=[]
    while jay<len(Z[crescendo][4]):
        jj=0
        while jj<len(Z[crescendo][1][jay]):
            if abs(Z[crescendo][1][jay][jj])==2:
                DEX.append(Z[crescendo][4][jay][jj])
            jj+=1
        jay+=1

    babygary=0
    while babygary<len(Z[crescendo][1]):
        bg=-1
        while bg<len(Z[crescendo][1][babygary])-1:
            if abs(Z[crescendo][1][babygary][bg])==2 and Z[crescendo][1][babygary]
[bg]==Z[crescendo][1][babygary][bg+1]:
                remise=0
                smol=min(Z[crescendo][4][babygary][bg],Z[crescendo][4][babygary]
[bg+1])
                lorg=max(Z[crescendo][4][babygary][bg],Z[crescendo][4][babygary]
[bg+1])
                john=smol+1
                while john<lorg:
                    dex=0
                    while dex<len(DEX):
                        if DEX[dex]==john:
                            remise=1
                        dex+=1
                    john+=1
                if remise==0:
                    del Z[crescendo][1][babygary][bg+1]
                    del Z[crescendo][4][babygary][bg+1]
                    del Z[crescendo][5][babygary][bg+1]
                    del Z[crescendo][1][babygary][bg]
                    del Z[crescendo][4][babygary][bg]
                    del Z[crescendo][5][babygary][bg]
                    bg=-1
                    d=0
                    while d<len(DEX):
                        if DEX[d]==smol or DEX[d]==lorg:
                            del DEX[d]
                        else:
                            d+=1
                else:
                    bg+=1
            else:
                bg+=1
        babygary+=1
    crescendo+=1

job=0
while job<len(Z):
    sob=0
    while sob<len(Z[job][1]):
        decide=0
        if len(Z[job][1][sob])==0:
            Z[job][6][4]=Z[job][6][4]+1
            del Z[job][1][sob]
            del Z[job][4][sob]
            del Z[job][5][sob]
            sob=0
        elif len(Z[job][1][sob])==2 and Z[job][1][sob][0]==-Z[job][1][sob][1]:
            if abs(Z[job][1][sob][0])==1:
                Z[job][6][0]=Z[job][6][0]+1
            elif abs(Z[job][1][sob][0])==2:
                Z[job][6][2]=Z[job][6][2]+1
            del Z[job][1][sob]
            del Z[job][4][sob]
            del Z[job][5][sob]
            sob=0

        elif len(Z[job][1][sob])==4:
            plutop=1
            rosa=0
            redover=0
            redunder=0
            blueover=0
            blueunder=0
            while rosa<4:
                if Z[job][1][sob][rosa]==1:
                    redover=Z[job][1][sob][rosa]
                    iredover=Z[job][4][sob][rosa]
                    qredover=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==-1:
                    redunder=Z[job][1][sob][rosa]
                    iredunder=Z[job][4][sob][rosa]
                    qredunder=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==2:
                    blueover=Z[job][1][sob][rosa]
                    iblueover=Z[job][4][sob][rosa]
                    qblueover=Z[job][5][sob][rosa]
                elif Z[job][1][sob][rosa]==-2:
                    blueunder=Z[job][1][sob][rosa]
                    iblueunder=Z[job][4][sob][rosa]
                    qblueunder=Z[job][5][sob][rosa]
                rosa+=1
            if abs(redover)>0 and abs(redunder)>0:
                imaxred=max(iredover,iredunder)
                iminred=min(iredover,iredunder)
                rosa=iminred+1
                while rosa<imaxred:
                    coza=0
                    while coza<len(Z[job][4]):
                        sosa=0
                        while sosa<len(Z[job][4][coza]):
                            if Z[job][4][coza][sosa]==rosa:
                                if abs(Z[job][1][coza][sosa])==1:
                                    plutop=0
                            sosa+=1
                        coza+=1
                    rosa+=1

            if abs(blueover)>0 and abs(blueunder)>0:
                imaxblue=max(iblueover,iblueunder)
                iminblue=min(iblueover,iblueunder)
                rosa=iminblue+1
                while rosa<imaxblue:
                    coza=0
                    while coza<len(Z[job][4]):
                        sosa=0
                        while sosa<len(Z[job][4][coza]):
                            if Z[job][4][coza][sosa]==rosa:
                                if abs(Z[job][1][coza][sosa])==2:
                                    plutop=0
                            sosa+=1
                        coza+=1
                    rosa+=1

            if abs(Z[job][1][sob][0])==abs(Z[job][1][sob][1]) and abs(Z[job][1][sob]
[1])==abs(Z[job][1][sob][2]) and abs(Z[job][1][sob][2])==abs(Z[job][1][sob][3]):
                decide=0
            elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob][2]==-Z[job]
[1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==Z[job][1][sob][2]/abs(Z[job]
[1][sob][2]):
                decide=1
            elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==Z[job][1][sob][1]/abs(Z[job]
[1][sob][1]):
                decide=1
            elif Z[job][1][sob][0]==-Z[job][1][sob][1] and Z[job][1][sob][2]==-Z[job]
[1][sob][3] and Z[job][1][sob][1]/abs(Z[job][1][sob][1])==-Z[job][1][sob][3]/
abs(Z[job][1][sob][3]):
                decide=2
            elif Z[job][1][sob][0]==-Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2] and Z[job][1][sob][0]/abs(Z[job][1][sob][0])==-Z[job][1][sob][2]/
abs(Z[job][1][sob][2]):
                decide=2
            elif Z[job][1][sob][0]==Z[job][1][sob][3] and Z[job][1][sob][1]==-Z[job]
[1][sob][2]:
                decide=3

            if decide>0 and plutop==1:
                b=[Z[job][4][sob][0],Z[job][4][sob][1],Z[job][4][sob][2], Z[job][4]
[sob][3]]
                bmax=max(b)
                bmin=min(b)
                dmax=[]
                bay=0
                while bay<len(Z[job][4]):
                    if len(Z[job][4][bay])>0:
                        dmax.append(max(Z[job][4][bay]))
                    bay+=1
                decidemax=max(dmax)
                dmin=[]
                bay=0
                while bay<len(Z[job][4]):
                    if len(Z[job][4][bay])>0:
                        dmin.append(min(Z[job][4][bay]))
                    bay+=1
                decidemin=min(dmin)
                if decidemax==bmax:
                    if decide ==1:
                        Z[job][6][1]=Z[job][6][1]+1
                    if decide ==2:
                        call=0
                        while call<len(Z[job][1][sob]):
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                elif decidemin==bmin:
                    if decide==1:
                        Z[job][6][1]=Z[job][6][1]+1
                    elif decide==2:
                        call=0
                        while call<len(Z[job][1][sob]):
                            if Z[job][1][sob][call]==1:
                                posguy=Z[job][4][sob][call]
                            elif Z[job][1][sob][call]==-1:
                                negguy=Z[job][4][sob][call]
                            call+=1
                        if posguy>negguy:
                            Z[job][6][3]=Z[job][6][3]+1
                        else:
                            Z[job][6][5]=Z[job][6][5]+1
                    elif decide==3:
                        if abs(Z[job][1][sob][1])==1:
                            Z[job][6][0]=Z[job][6][0]+1
                        elif abs(Z[job][1][sob][1])==2:
                            Z[job][6][2]=Z[job][6][2]+1
                    del Z[job][1][sob]
                    del Z[job][4][sob]
                    del Z[job][5][sob]
                    sob=0
                else:
                    sob+=1
            else:
                sob+=1
        else:
            sob+=1
    job+=1

brandon=0
while brandon<len(Z):
    sigma=0
    while len(Z[brandon][1])>0:
        endd=len(Z[brandon][1][sigma])-1
        while endd>0 and Z[brandon][1][sigma][0]==Z[brandon][1][sigma][endd]:
            del Z[brandon][1][sigma][endd]
            del Z[brandon][4][sigma][endd]
            del Z[brandon][5][sigma][endd]
            del Z[brandon][1][sigma][0]
            del Z[brandon][4][sigma][0]
            del Z[brandon][5][sigma][0]
            endd-=2
        homie=0
        while homie==0:
            if len(Z[brandon][1][sigma])>0 and Z[brandon][1][sigma][0]==Z[brandon][1]
[sigma][1]:
                del Z[brandon][1][sigma][1]
                del Z[brandon][4][sigma][1]
                del Z[brandon][5][sigma][1]
                del Z[brandon][1][sigma][0]
                del Z[brandon][4][sigma][0]
                del Z[brandon][5][sigma][0]
            endd=len(Z[brandon][1][sigma])-1
            if  len(Z[brandon][1][sigma])>0 and Z[brandon][1][sigma]
[endd-1]==Z[brandon][1][sigma][endd]:
                del Z[brandon][1][sigma][endd]
                del Z[brandon][4][sigma][endd]
                del Z[brandon][5][sigma][endd]
                del Z[brandon][1][sigma][endd-1]
                del Z[brandon][4][sigma][endd-1]
                del Z[brandon][5][sigma][endd-1]
                endd-=2
            if len(Z[brandon][1][sigma])>0 and Z[brandon][1][sigma][0]==Z[brandon][1]
[sigma][endd]:
                while Z[brandon][1][sigma][0]==Z[brandon][1][sigma][endd]:
                    del Z[brandon][1][sigma][endd]
                    del Z[brandon][4][sigma][endd]
                    del Z[brandon][5][sigma][endd]
                    del Z[brandon][1][sigma][0]
                    del Z[brandon][4][sigma][0]
                    del Z[brandon][5][sigma][0]
                    endd-=2
            else:
                homie=1
        while squeakin<len(Z[brandon][1][sigma])-1:
            if Z[brandon][1][sigma][squeakin]==Z[brandon][1][sigma][squeakin+1]:
                del Z[brandon][1][sigma][squeakin+1]
                del Z[brandon][4][sigma][squeakin+1]
                del Z[brandon][5][sigma][squeakin+1]
                del Z[brandon][1][sigma][squeakin]
                del Z[brandon][4][sigma][squeakin]
                del Z[brandon][5][sigma][squeakin]
                squeakin-=1
            else:
                squeakin+=1
        if len(Z[brandon][1][sigma])==0:
            Z[brandon][6][4]=Z[brandon][6][4]+1
            del Z[brandon][1][sigma]
            del Z[brandon][4][sigma]
            del Z[brandon][5][sigma]

        elif len(Z[brandon][1][sigma])==2:
            if abs(Z[brandon][1][sigma][0])==1:
                Z[brandon][6][0]=Z[brandon][6][0]+1
                del Z[brandon][1][sigma]
                del Z[brandon][4][sigma]
                del Z[brandon][5][sigma]
            elif abs(Z[brandon][1][sigma][0])==2:
                Z[brandon][6][2]=Z[brandon][6][2]+1
                del Z[brandon][1][sigma]
                del Z[brandon][4][sigma]
                del Z[brandon][5][sigma]
        elif len(Z[brandon][1][sigma])==4:
            if Z[brandon][1][sigma][0]==-Z[brandon][1][sigma][1] and Z[brandon][1]
[sigma][2]==-Z[brandon][1][sigma][3] and Z[brandon][1][sigma][1]/abs(Z[brandon][1]
[sigma][1])==Z[brandon][1][sigma][2]/abs(Z[brandon][1][sigma][2]):
                Z[brandon][6][1]=Z[brandon][6][1]+1
            elif Z[brandon][1][sigma][0]==-Z[brandon][1][sigma][3] and Z[brandon][1]
[sigma][1]==-Z[brandon][1][sigma][2] and Z[brandon][1][sigma][0]/abs(Z[brandon][1]
[sigma][0])==Z[brandon][1][sigma][1]/abs(Z[brandon][1][sigma][1]):
                Z[brandon][6][1]=Z[brandon][6][1]+1

            elif Z[brandon][1][sigma][0]==-Z[brandon][1][sigma][1] and Z[brandon][1]
[sigma][2]==-Z[brandon][1][sigma][3] and Z[brandon][1][sigma][1]/abs(Z[brandon][1]
[sigma][1])==-Z[brandon][1][sigma][3]/abs(Z[brandon][1][sigma][3]):
                if abs(Z[brandon][1][sigma][0])==1:
                    t1=Z[brandon][4][sigma][0]
                    t2=Z[brandon][4][sigma][1]
                    if t1>t2:
                        if t1==1:
                            Z[brandon][6][3]=Z[brandon][6][3]+1
                        else:
                            Z[brandon][6][5]=Z[brandon][6]
[5]+1
                    elif t2>t1:
                        if t2==1:
                            Z[brandon][6][3]=Z[brandon][6][3]+1
                        else:
                            Z[brandon][6][5]=Z[brandon][6][5]+1
                else:
                    t1=Z[brandon][4][sigma][2]
                    t2=Z[brandon][4][sigma][3]
                    if t1>t2:
                        if t1==1:
                            Z[brandon][6][3]=Z[brandon][6][3]+1
                        else:
                            Z[brandon][6][5]=Z[brandon][6][5]+1
                    elif t2>t1:
                        if t2==1:
                            Z[brandon][6][3]=Z[brandon][6][3]+1
                        else:
                            Z[brandon][6][5]=Z[brandon][6][5]+1

            elif Z[brandon][1][sigma][0]==-Z[brandon][1][sigma][3] and Z[brandon][1]
[sigma][1]==-Z[brandon][1][sigma][2] and Z[brandon][1][sigma][0]/abs(Z[brandon][1]
[sigma][0])==-Z[brandon][1][sigma][2]/abs(Z[brandon][1][sigma][2]):
                if abs(Z[brandon][1][sigma][0])==1:
                    t1=Z[brandon][4][sigma][0]
                    t2=Z[brandon][4][sigma][3]
                    if t1>t2:
                        if t1==1:
                            Z[brandon][6][3]=Z[brandon][6][3]+1
                        else:
                            Z[brandon][6][5]=Z[brandon][6][5]+1
                    elif t2>t1:
                        if t2==1:
                            Z[brandon][6][3]=Z[brandon][6][3]+1
                        else:
                            Z[brandon][6][5]=Z[brandon][6][5]+1
                else:
                    t1=Z[brandon][4][sigma][1]
                    t2=Z[brandon][4][sigma][2]
                    if t1>t2:
                        if t1==1:
                            Z[brandon][6][3]=Z[brandon][6][3]+1
                        else:
                            Z[brandon][6][5]=Z[brandon][6][5]+1
                    elif t2>t1:
                        if t2==1:
                            Z[brandon][6][3]=Z[brandon][6][3]+1
                        else:
                            Z[brandon][6][5]=Z[brandon][6][5]+1
            del Z[brandon][1][sigma]
            del Z[brandon][4][sigma]
            del Z[brandon][5][sigma]
    brandon+=1


#POST-PROCESSING: PROGRAM COMBINE

z=0
while z<len(Z):
    if Z[z][6][3]>0:
        B=copy_2d(Z[z])
        B[0]=B[0]-2
        B[2]=-B[2]
        B[6][0]=B[6][0]+1
        B[6][2]=B[6][2]+1
        B[6][3]=B[6][3]-1
        C=copy_2d(Z[z])
        C[0]=C[0]-4
        C[2]=-C[2]
        C[6][1]=C[6][1]+1
        C[6][3]=C[6][3]-1
        Z.insert(z,C)
        Z.insert(z,B)
        del Z[z+2]
    else:
        z+=1

z=0
while z<len(Z):
    if Z[z][6][5]>0:
        B=copy_2d(Z[z])
        B[0]=B[0]+4
        B[2]=-B[2]
        B[6][1]=B[6][1]+1
        B[6][5]=B[6][5]-1
        C=copy_2d(Z[z])
        C[0]=C[0]+2
        C[2]=-C[2]
        C[6][0]=C[6][0]+1
        C[6][2]=C[6][2]+1
        C[6][5]=C[6][5]-1
        Z.insert(z,C)
        Z.insert(z,B)
        del Z[z+2]
    else:
        z+=1

z=0
```

```
while z<len(Z):
    if Z[z][6][4]>0:
        B=copy_2d(Z[z])
        B[0]=B[0]-2
        B[2]=-B[2]
        B[6][4]=B[6][4]-1
        C=copy_2d(Z[z])
        C[0]=C[0]+2
        C[2]=-C[2]
        C[6][4]=C[6][4]-1
        Z.insert(z,C)
        Z.insert(z,B)
        del Z[z+2]
    else:
        z+=1

z=0
while z<len(Z):
    if Z[z][6][2]>0:
        Z[z][6][0]=Z[z][6][0]+1
        Z[z][6][2]=Z[z][6][2]-1
    else:
        z+=1

z=0
while z<len(Z):
    a=z+1
    while a<len(Z):
        if Z[a][0]==Z[z][0]:
            if Z[z][6][0]==Z[a][6][0] and Z[z][6][1]==Z[a][6][1]:
                Z[z][2]=Z[z][2]+Z[a][2]
                del Z[a]
            else:
                a+=1
        else:
            a+=1
    z+=1

z=0
while z<len(Z):
    if Z[z][2]==0:
        del(Z[z])
    else:
        z+=1

OUTPUT: Z
```

BIBLIOGRAPHY

[1] Christian Blanchet, Nathan Habegger, Gregor Masbaum, and Pierre Vogel. Topological quantum feld theories derived from the Kauffman bracket. *Topology*, 31:685–699, 1992.

[2] Doug Bullock. The $(2, \infty)$ skein module of the complement of a $(2, 2p+1)$ torus knot. *J. Knot Theory Ramifications*, 4:619–632, 1995.

[3] Charles Frohman and Răzvan Gelca. Skein modules and the noncommutative torus. *Transactions of the American Mathematical Society*, 352(10):4877–4888, 2000.

[4] Charles Frohman, Razvan Gelca, and Walter Lofaro. The A-polynomial from the noncommutative viewpoint. *Transactions Amer. Math. Soc.*, 354:735–747, 2001.

[5] Razvan Gelca. Noncommutative trigonometry and the A-polynomial of the trefoil knot. *Math. Proc. of the Cambridge Philosophical Society*, 1, 2002.

[6] Razvan Gelca and Fumikazu Nagasato. Some results about the Kauffman bracket skein module of the twist knot exterior. *J. Knot Theory Ramif.*, 8:1095–1106, 2006.

[7] Răzvan Gelca and Jeremy Sain. The computation of the non-commutative generalization of the A-polynomial of the figure-eight knot. *Journal of Knot Theory and Its Ramifications*, 13(06):785–808, 2004.

[8] Razvan Gelca and Alejandro Uribe. Quantum mechanics and non-abelian theta functions for the gauge group $SU(2)$. *Fundamenta Mathematica*, 228:97–137, 2015.

[9] Razvan Gelca and Hongwei Wang. The action of the Kauffman bracket skein algebra of the torus on the Kauffman bracket skein module of the 3-twist knot complement. *J. Knot Theory and Ramif.*, to appear.

[10] Vaughan FR Jones. Polynomial invariants of knots via von Neumann algebras. *Bull. Amer. Math. Soc.*, 12:103–111, 1985.

[11] Louis H Kauffman. State models and the Jones polynomial. *Topology*, 26:395–407, 1987.

[12] Robion Kirby and Paul Melvin. The 3-manifold invariants of Witten and Reshetikhin-Turaev for $sl(2, \mathbb{C})$. *Inventiones Mathematicae*, 105:547–597, 1991.

[13] Thang TQ Le. The colored Jones polynomial and the A-polynomial of knots. *Adv. in Math.*, 207:782–804, 2006.

[14] WB Raymond Lickorish. The skein method for three-manifold invariants. *Journal of Knot Theory and Its Ramifications*, 2(02):171–194, 1993.

[15] Józef H Przytycki. Skein modules of 3-manifolds. *Bull. Pol. Acad. Sci*, pages 91–100, 1991.

[16] Nicolai Reshetikhin and Vladimir G Turaev. Invariants of 3-manifolds via link polynomials and quantum groups. *Inventiones mathematicae*, 103(1):547–597, 1991.

[17] Edward Witten. Quantum field theory and the Jones polynomial. *Communicantions in Mathematical Physics*, 121:351–399, 1989.