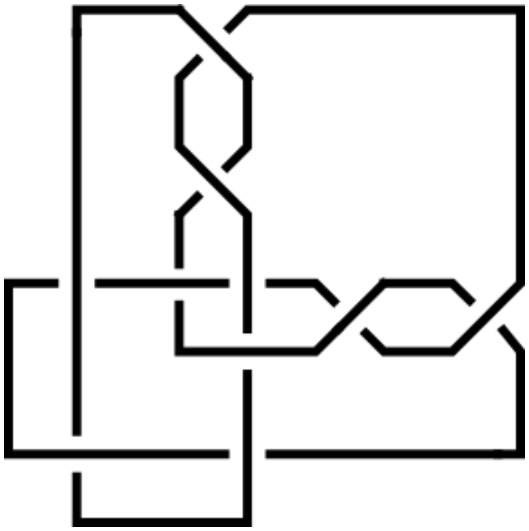
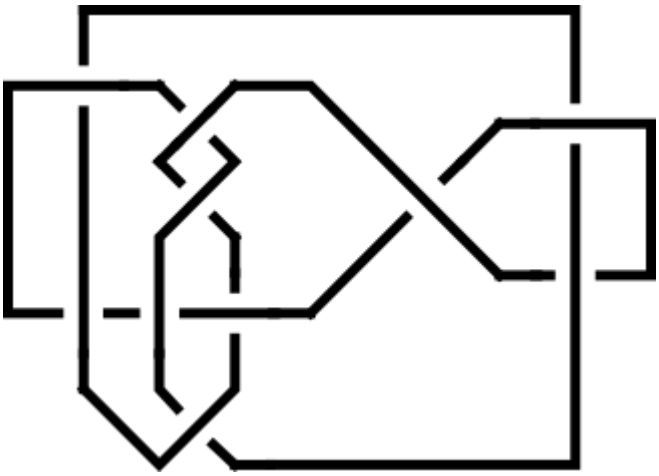
university of groningen

faculty of science and engineering

Building a Table of Knots





Master Project Mathematics

June 2025

Student: R. B. de Vries

First supervisor: Prof. dr. R. van der Veen

Second supervisor: Dr. N. Martynchuk

CONTENTS

1 ABSTRACT

In this thesis, the goal is to produce a table of knots of up to 10 crossings. To achieve this, we first cover basic knot theory which defines equivalence of knots, various moves that can be applied to knots, properties knots can have and representing knots numerically. After this, we cover knot invariants like the crossing number, the Jones polynomial and the Theta invariant. Then we introduce plantri, a graph generating tool that will be used to generate our starting dataset. Using all these tools, we will build a knot table using a computer in Python. In the end, our final result is an almost perfect knot table. For knots with up to nine crossings (inclusive), our results match the number of distinct knots listed in other tables, such as KnotInfo [1]. For ten crossings, our results almost match. We find four extra knots, which, after comparison with KnotInfo, are determined to be reducible.

2 INTRODUCTION

Knots are closed, non-self-intersecting curves embedded in $\mathbb{R}^3$ and have zero thickness. As they are closed curves, they are quite different from objects most people would call knots. For example, sailing knots and climbing knots are not considered mathematical knots. Knots have been used and have appeared in human history for a long time. As early as the prehistoric era, the hunter-gatherers have used knots to create basic tools and ropes. Later on, they were used in climbing or sailing and even later they have appeared in religious symbolism like the endless knot from Tibetan Buddhism or as decoration in the form of Celtic knots.

Actual knot theory only started to develop around 1833 with Carl Friedrich Gauss, who developed the Gauss linking integral used for computing the linking number of two knots [2]. Around 1867, the physicist Peter Tait was experimenting with smoke rings. Sir William Thomson observed this and created the idea that atoms were actually knotted vortices in the aether. This idea becomes known as the vortex theory of the atom. Different chemical elements would under this theory correspond to knots and links. This lead them to construct a table of unique knots similar to how the periodic table was created. They believed that the knotted structures could explain why atoms absorb and emit light at specific discrete wavelengths. Tait also formulated three conjectures that are now known as the Tait conjectures [3]. Tait tabulated all knots up to nine crossings together with Thomas Kirkman in [4] and the knots with 10 crossings were tabulated in [5] by Tait, Kirkman and Newton Little.

Unfortunately, in 1974, Kenneth Perko discovered a pair of knots in Tait's table that were actually the same knot [6]. This pair of knots, labeled 10_{161} and 10_{162} in Tait's table, is now known as the Perko pair. This pair of knots can also be found on the front page of this thesis. In the early decades of the 20th century, following the developments in topology, mathematicians like Henri Poincaré and topologists like Max Dehn, James Waddell Alexander II and Kurt Reidemeister started to more closely investigate knots. In this period, Kurt Reidemeister demonstrated that two knot diagrams belonging to the same knot can be related by a sequence of the three Reidemeister moves [7]. His theorem became the basis for finding knot invariants. Knot invariants are expressions that do not change under the application of Reidemeister moves. Therefore, they are a great tool for telling knots apart. J. W. Alexander invented the Alexander Polynomial, a knot invariant. This invariant is a polynomial with integer coefficients. Max Dehn developed Dehn surgery, which allowed him to relate knots to the general theory of 3-manifolds. Another important invariant is the Jones polynomial, developed by Vaughan Jones in 1984 [8]. This polynomial led to other important knot invariants like the bracket polynomial, HOMFLY polynomial, and Kauffman polynomial.

So far all tabulations have been done by hand, but in 1998 with the help of computers, Jim Hoste, Jeff Weeks, and Morwen Thistlethwaite tabulated knots up to and including 16 crossings [9]. The three did separate research and used different algorithms, but agreed on the count of
1 701 936 prime knots including the unknot. More recently, in 2020, Benjamin Burton tabulated all knots up to and including 19 crossings and found 352 152 252 distinct non-trivial prime knots [10]. Modern knot theory has applications in a lot of fields. In biology and biochemistry, there are applications in DNA topology, protein folding and virus structure as all three may form knotted structures. Similarly in chemistry, molecules may also consist of knotted or linked structures. For physics, knot theory is related to topological quantum field theory through Chern-Simons theory as this a another way to calculate invariants like the Jones polynomial.

The goal for us is to also produce a knot table, but of a more modest size. The goal is to tabulate all knots up to at least 10 crossings using a computer. In order to do this we need to through some basic knot theory in chapter 3. Here we cover the various moves we will use, properties a knot can have and ways to write down knots using only numbers. After this, in chapter 4, we will talk about what a knot invariant is and give examples of invariants that we will use. In chapter 5, we will discuss the dataset we start with, which comes from the program plantri. In chapter 6, we will describe the process used to build the knot table. In chapter 7 we provide the code used to generate the knot table and give a simple explanation on how the code works.

Lastly, we have chapter 8 where we show the entire knot table.

Building the knot tables required various steps. We start with a dataset of graphs gathered from the program plantri. Each dataset is a set of graphs of a specific amount of vertices. These graphs can be turned into knots by adding signs to each vertex. Our goal is then to reduce each dataset by filtering out duplicates and filtering out knots that can be represented with a smaller amount of crossings. Each dataset contains links, which we also want to filter out. This filtering will happen in various ways. There are two types of moves that we can use to transform the knot: crossing number preserving moves and crossing number reducing moves. For the preserving moves we will use the third Reidemeister move and flypes. For the reducing moves we will use the first and second Reidemeister move and the two-one pass. We will also use knot invariants like the Jones polynomial and Theta invariant to group our knots. This will allow for more efficient comparing and allows us to figure out which knot is the mirror image of another knot. Finally, we will draw the knots and put them in a table together with a name and information about the the chirality of the knot, the Jones polynomial and an evaluated Theta invariant.

3 KNOTS

A knot, in the mathematical sense, is quite different from the everyday idea of a knot in a rope or shoelace. Mathematically speaking, a knot is a closed, non-self-intersecting curve that is embedded in three-dimensional space. You can make a mathematical knot from your everyday knot by fusing the ends together so that it forms a closed loop. Another important difference is that mathematical knots have zero thickness and arbitrary length. The latter allows it to stretch and shrink as much as we want to. This idea gives the following formal definition:

Definition 1 (Knot [11])**.** A knot is defined as a smooth embedding of the circle S^1 into three-dimensional space $\mathbb{R}^3$. That is, a knot is a map $K : S^1 \hookrightarrow \mathbb{R}^3$ such that K is injective and infinitely differentiable.

This embedding represents a simple, closed, non-self-intersecting curve in space, and unlike everyday knots made of string or rope, it has no endpoints. One can also take multiple knots and intertwine them to form a link.

Definition 2 (Link [11])**.** A link is defined as a smooth embedding of a finite collection of disjoined circles into three-dimensional space $S^1 \sqcup \cdots \sqcup S^1 \to \mathbb{R}^3$.

Since the goal is to make a knot table and not a link table, we will not concern ourselves too much with links.

Two knots K_1 and K_2 should be equivalent, if K_1 can be continuously deformed into K_2. This means that you may not tear, glue, or allow the curve to intersect itself. Such an equivalence is called an ambient isotopy.

Definition 3 (Equivalence [11])**.** Two knots are equivalent if there exists an ambient isotopy between them.

This definition ensures that physical properties like thickness and length of the strands are irrelevant. What remains is the topological structure of the knot itself. An interesting consequence of knots having zero thickness is that there exists mathematical knots that are equivalent to the unknot (figure 2) but a physical representation (with non-zero thickness) can not be untied to the unknot. Such a knot is called a gordian knot [12].

Given the three-dimensional nature of knots, visualizing and working with them in practice typically requires a planar representation. For this, we will use a knot diagram. A knot diagram is a projection of the knot onto $\mathbb{R}^2$. A normal projection loses information about the knot at the crossings, specifically whether a strand goes over or under another strand. To remedy this, a convention is used which is shown in figure 1. Here, the strand labeled A traverses over the strand labeled B at the crossing.

Definition 4 (Orientation)**.** A knot is oriented if there is a continuous choice of direction along the knot.

Orientation is often represented with arrows that points in the direction of the orientation. This allowed us to distinguish two different types of crossings, positive and negative crossings. This is also illustrated in 1. We are not concerned with oriented knots and will say that the two different orientations of the knot are in fact the same knot. That being said, it is quite useful to put an orientation on the knot anyway. This will, for example, be required to compute invariants like the Theta invariant.

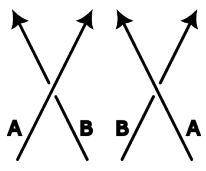


Figure 1: Here we have a positive crossing on the left and a negative crossing on the right

Using this convention, we can now show examples of some knots. Although it may not look like it on first glance, the middle and right knots are equivalent and are both an unknot.

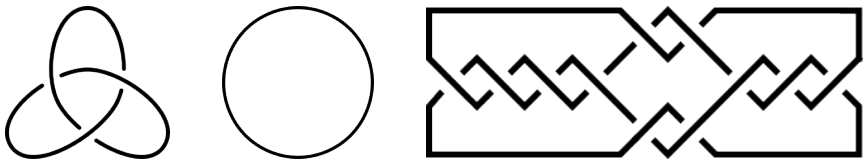


Figure 2: Here we have the trefoil knot on the left, the unknot in the middle and the Goeritz unknot on the right

In 1927, Kurt Reidemeister showed that if you have two planar diagrams of the representing equivalent knots, then you can transform one into the other using a finite sequence of only three basic operations [7]. These simple operations are called the Reidemeister moves.

Theorem 1 (Reidemeister Moves [7])**.** *Two (unoriented) knots* K_1 *and* K_2*, are equivalent if and only if a diagram of* K_1 *can be transformed into a diagram of* K_2 *by a sequence of ambient isotopies of the plane and local moves of the following three types, which are called the Reidemeister moves:*

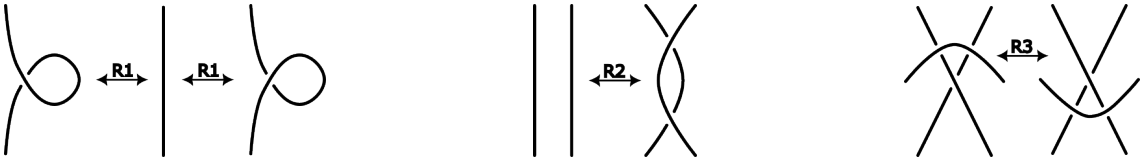


Figure 3: Reidemeister Moves

These moves can be described as:

R1: A twist or untwist of a strand in either direction, creating or removing a crossing.

R2: Moving a strand over another strand, creating or removing two crossings.

R3: Moving a strand to the other side of a crossing.

A proof on why only these simple moves are sufficient to show knot equivalence can be found in [7]. Even though these are the minimal amount of moves needed to get from one knot to another, it is useful to look at more methods of modifying knots.

Definition 5 (Two-one pass)**.** A two-one pass is a move that reduces the amount of crossings by one. This happens by moving the vertical strand on the left under a segment of the knot to the right.

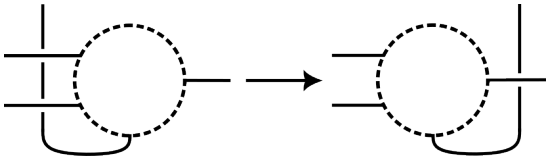


Figure 4: Two-one pass

The dotted circle represents a segment of the knot.

Definition 6 (Flype)**.** A flype is a crossing number preserving move that flips a segment of the knot upside down and moves a crossing from the left of this segment to the right.

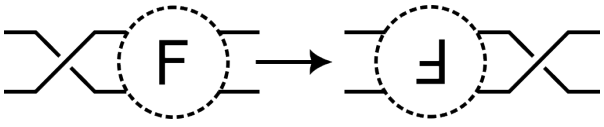


Figure 5: Example of a flype. The big F in the dotted region represents the fact that this segment flips upside down after the move.

Definition 7 (Connected Sum)**.** For the connected sum of two knots, take an edge of each knot. Then remove these edges and replace them by new edges connecting the two knots. This is also illustrated in the following figure:

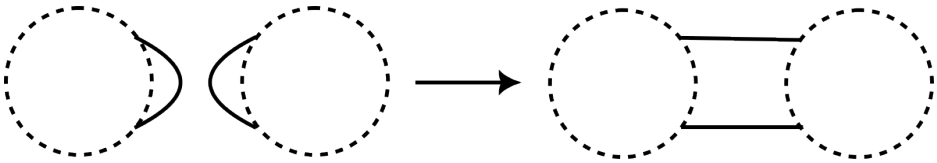


Figure 6: Connected sum

There are some helpful properties knots can have. These definitions will help us with building the knot table.

Definition 8 (Prime knot)**.** A knot is prime if it can not be written as the connected sum of two knots that are not the unknot.

For knot tables, it it conventional to only list prime knots. If a knot is not equal to its mirror image it is also conventional to only list one of them. A note will also be made when a knot is amphichiral.

Definition 9 (Mirror image)**.** For a knot K, we obtain its mirror image $\overline{K}$ by changing every overcrossing to an undercrossing and changing every undercrossing to an overcrossing.

Definition 10 (Amphichiral)**.** A knot K is amphichiral if $K = \overline{K}$, i.e. if k is equivalent to its mirror image.

The next two theorems are two of Tait's conjectures. Although they are still called conjectures they have been proven already. For example, a proof for the first one can be found in [13] and a proof of the second can be found in [14].

Theorem 2 (The alternating Tait conjecture [3])**.** *A reduced alternating diagram has the minimal number of crossings among all diagrams of the given knot.*

This conjecture sounds useful, however it only works in one direction. Unfortunately there exist non-alternating but reduced knots for as few as eight crossings. This mean we can not check for reducibility by only checking if the knot is alternating. Therefore this will not be used to test whether or not a given knot is reduced when building the knot table. The next Tait conjecture will be significantly more useful.

Theorem 3 (The Tait flyping conjecture [3])**.** *Given any two reduced alternating diagrams* K_1, K_2 *of an oriented, prime alternating knot:* K_1 *may be transformed to* K_2 *by means of a sequence of flypes.*

Using this theorem we will be able to compare two knots and check if they are the same knot or not.

The planar representation of a knot can be compacted into a code. There are several ways to encode a knot that each have their upsides and downsides. The relevant codes for this thesis are Alexander-Briggs notation, Gauss code and planar diagram notation. Later on we will also cover the notation plantri uses, but it encodes a graph rather than a knot.

Definition 11 (Alexander-Briggs notation [15])**.** The Alexander-Briggs notation consists of 2 numbers, a_b. Here, a is the amount of crossings and b is an index number.

The index b has no special meaning, but conventionally the order of the Rolfsen Table is used [16]. Here, the knot is assumed to have a minimal amount of crossings. As example, the trefoil knot (figure 2) has, at minimum, 3 crossings and happens to be the only knot with 3 crossings. This means the the trefoil knot is labeled 3_1.

Around 1820, Carl Friedrich Gauss laid the foundation to what later became the Gauss code[2].

Definition 12 (Gauss Code)**.** Given a oriented knot, give every crossing a unique number between 1 and the amount of crossings. Pick a starting point on the knot and traverse the knot in the direction of its orientation. For each crossing we reach we add its label to a list if is an overcrossing. If we reach an undercrossing, we add its label times -1. The resulting list is called the Gauss code.

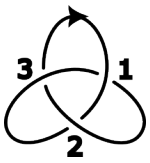


Figure 7: The trefoil knot, with labeled crossings and an orientation

Example 1 (Gauss code of the trefoil knot)**.** For the oriented and labeled trefoil knot in figure (7), let us choose the starting point to be where the arrowhead is. We then first come across an overcrossing at 1 and then an undercrossing at 2 etc. This gives us the following Gauss code:

$$1, -2, 3, -1, 2, -3$$

It is conventional to choose the starting point such that the Gauss code starts with positive 1 and label the crossings in the order see them, but this is not required. Any starting point, labeling and orientation can give you a Gauss code. This means there are up to $3 \cdot 4 = 12$ different Gauss codes for the trefoil knot. This is because there are 3 crossings and for each crossing we can start in 4 different directions. Another issue with the Gauss code is that it can not distinguish between positive and negative crossings. As a result, it can not distinguish between a knot and its mirror image.

Instead of labeling the crossings, one could also label the edges between the crossings. This is what Planar diagram notation does.

Definition 13 (Planar diagram notation)**.** Planar diagram notation, or PD code for short, is defined on oriented knots. Pick a starting point, then label all edges in sequence in the direction of the orientation. For each crossing, we make a 4-tuple starting with the edge corresponding to the incoming undercrossing and listing the remaining edges of the crossing in counterclockwise order. The PD code is the resulting list of all 4-tuples.

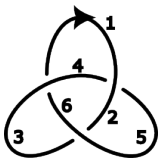


Figure 8: The oriented trefoil knot with its edges labeled

Example 2 (PD code of the trefoil knot)**.** For the oriented and labeled trefoil knot in figure (8) we start with the crossing in the top right. Edge 4 is the incoming undercrossing for that crossing, therefore its 4-tuple starts with 4. If we go counterclockwise around that crossing, we find 2, then 5 and lastly 1. The 4-tuple becomes $[4, 2, 5, 1]$. Doing this for the remaining crossings as well results in the following PD code:

$$[[4, 2, 5, 1], [2, 6, 3, 5], [6, 4, 1, 3]] \tag{1}$$

Just like Gauss codes, PD codes are not unique. This can be somewhat remedied by choosing a canonical representation. If we fix the labels, we can sort the PD code based on the first element of each crossing. This turns (1) into:

$$[[2, 6, 3, 5], [4, 2, 5, 1], [6, 4, 1, 3]]$$

We can also look at all combinations of labels and orientations and then choose the lexicographically smallest combination. This will be our canonical representation of a knot. For the trefoil this is:

$$[[1,5,2,4],[3,1,4,6],[5,3,6,2]]$$

4 KNOT INVARIANTS

An invariant is property of an object such that it does not change under certain operations. For knots we define invariants to not change under Reidemeister moves. This ensures that two equivalent knots give the same result when passed through an invariant function.

Definition 14 (Knot invariant)**.** A knot invariant is a quantity or property that is the same for all diagrams of a given knot, regardless of how the knot is twisted or deformed. i.e. it assigns the same value to all knots that are topologically equivalent.

Invariants are useful to tell two knots apart. By definition we have that if two knots produce two different results from an invariant function, then the two knots can not be equivalent. Invariants can have different strengths. An example of an invariant is the Crossing number.

Definition 15 (Crossing number)**.** The crossing number is the minimum number of crossings in any diagram.

This is the invariant that Alexander-Briggs notation (definition 11) is based on. This is a weak invariant as the amount of knots with the same crossing number is rather large for almost all crossing numbers. The crossing number seems simple, but it is rather hard to compute. For complex knots, how can you be sure that you really have the minimal amount of crossings? This is a hard question and it is the main question for this thesis and knot tabulations in general.

Definition 16 (Skein relation)**.** A skein relation relates link diagrams (usually three) that are identical except in a small region where they differ by a single crossing. The standard types are:

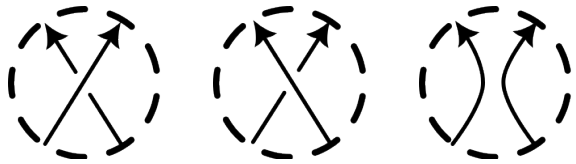


Figure 9: L_+ on the left, L_- in the middle and L_0 on the right.

The dotted circle represents the rest of the knot. A skein relation is a formula using these diagrams.

Skein relations can be used to recursively define polynomial invariants of knots by expressing a complicated knot in terms of simpler ones. An example of this is the Kauffman bracket. The Kauffman bracket is a function that takes in a knot and produces a polynomial. It was introduced by Louis Kauffman in 1987 [17].

Definition 17 (Kauffman bracket [17])**.** For a knot K, its Kauffman bracket $\langle K \rangle$ is a polynomial in A and is defined by three rules.

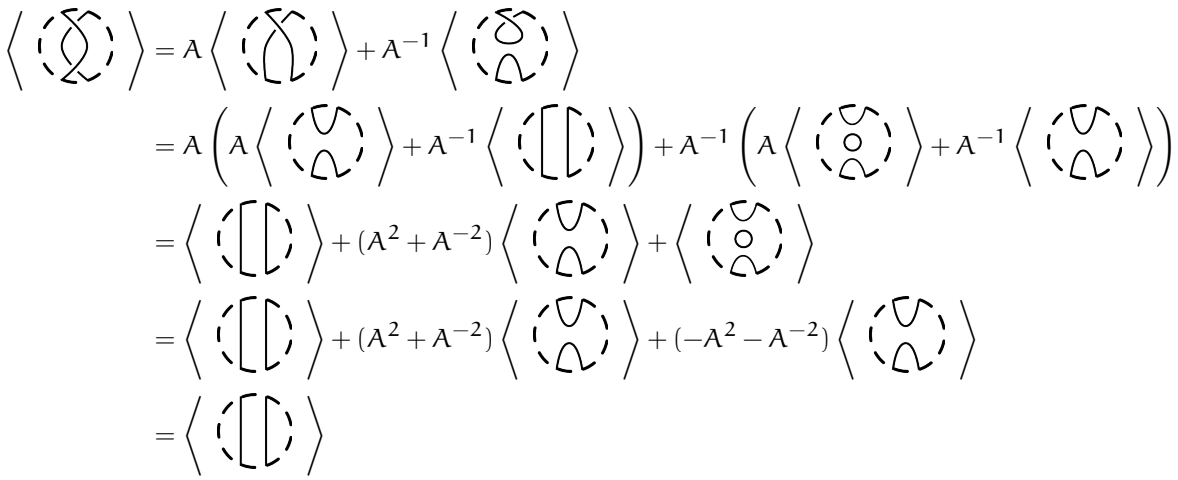
1. $\langle \bigcirc \rangle = 1$, where $\bigcirc$ is the unknot.
2. $\langle \bigcirc \sqcup K \rangle = (-A^2 - A^{-2})\langle K \rangle$
3. $\left\langle \right\rangle = A \left\langle \right\rangle + A^{-1} \left\langle \right\rangle$

These rules should be applied until all crossings are resolved and what remains is a polynomial.

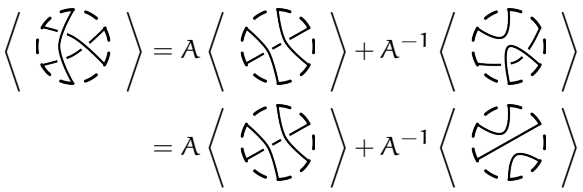
The third rule is transforming one knot into two simpler knots. At worst, for a knot with n crossings you will have to do 2^n calculations. When doing this manually, this process can be sped up by grouping knots that are the same and by applying the second Reidemeister move. The latter works, because the Kauffman bracket is invariant under the second and third Reidemeister move as we will prove soon. A downside of the Kauffman bracket is that it is not invariant under the first Reidemeister move.

Theorem 4. *The Kauffman bracket is invariant under the second and third Reidemeister move.*

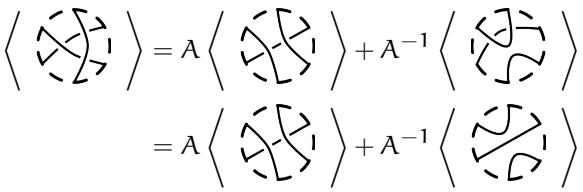
Proof. If we can show that applying the Kauffman bracket to both sides of the Reidemeister moves gives an equality, we are done. First, the second Reidemeister move. It consists of two crossings which we can expand with the third rule. We start by expanding the lower crossing, then the upper crossing.



Hence, the Kauffman bracket is invariant under the second Reidemeister move. For the third Reidemeister move we can do something similar and check if both sides of the move evaluate to the same. For the first side, we will evaluate the top-left crossing. As we have proven the second Reidemeister move, we can use it in the proof.



For the other side, we evaluate the bottom-left crossing and also use the second Reidemeister move.



As they lead to the same result, the Kauffman bracket is invariant under the third Reidemeister move as well. □

The fact that the Kauffman bracket is not invariant under the first Reidemeister move can be remedied by involving the writhe of a knot.

Definition 18 (Writhe [17])**.** For an oriented knot diagram K its writhe $w(K)$ is defined as the amount of positive crossings minus the amount of negative crossings.

$$w(K) = \#\text{Positive crossings} - \#\text{Negative crossings}$$

Using the writhe, we can define the normalized Kauffman bracket, which is also invariant under the first Reidemeister move.

Definition 19 (Normalized Kauffman bracket [17])**.** For an oriented knot K its normalized Kauffman bracket, $\langle K \rangle_N$ is a polynomial in A and is computed as follows:

$$\langle K \rangle_N = (-A)^{-3w(K)} \langle K \rangle$$

Where $\langle K \rangle$ is the Kauffman bracket and $w(K)$ is the writhe.

For the writhe, one needs to put an orientation on the knot. Luckily, it turns out that the choice of orientation does not matter for the writhe. The normalized Kauffman bracket is useful because it is now invariant under all Reidemeister moves. It is a stronger invariant than the crossing number as fewer knots share the same normalized Kauffman bracket. It is also significantly more practical than the crossing number as its computation is more direct. It is also useful because a simple substitution for A turns it into the Jones polynomial.

Definition 20 (Jones Polynomial [8])**.** The Jones polynomial of an oriented knot K, denoted by $V(K)$, is defined using the following rules:

1. $V(\bigcirc) = 1$, where $\bigcirc$ is the unknot.
2. $t^{-1}V\left(\right) - tV\left(\right) = (t^{1/2} - t^{-1/2})V\left(\right)$

Another way of obtaining the Jones polynomial is via substitution. Some use this as a definition instead and then derive the relations from there. A proof that this substitution works can be found in [13]. Since one can obtain the Jones polynomial via substitution from the normalized Kauffman bracket, they are equally strong invariants.

Theorem 5. *The Jones Polynomial can be obtained from the Normalized Kauffman Bracket by the substitution* $A = t^{-1/4}$.

The reason this substitution is useful is twofold. First of all, unlike the Kauffman bracket or normalized Kauffman bracket, there are tables available that show the Jones polynomial together with the Alexander-Briggs notation. This allows us to verify the knot table we end up building. The main table we will use for verification of our own knot table is the one from KnotInfo [1]. The second reason is that, from a coding perspective, it is significantly easier to compute the Kauffman bracket and writhe. This is because one iteration of the Kauffman bracket reduces your knot into two knots with a lower amount of crossings. The Jones polynomial gives you back a simpler knot and a knot with the opposite crossing. This second knot then has to be reduced with a Reidemeister move, making it harder to resolve.

If you have the Jones polynomial of a knot, then you can obtain the Jones polynomial of the mirror image of the knot by substituting $1/t$ for t. You can see this by both substituting $1/t$ for t in rule two of definition 20 and reversing the crossings. This can then be rewritten into the original rule. An amphichiral knot is equivalent to its mirror image, therefore the Jones polynomial of such a knot and its mirror image are the same. However, the reverse is not necessarily true. For example, the knot 9_{42} has Jones polynomial $t^3 - t^2 + t - 1 + \frac{1}{t} - \frac{1}{t^2} + \frac{1}{t^3}$ but is not amphichiral.

The Jones polynomial is stronger than the crossing number, but not strong enough to be able to distinguish every pair of knots. For a stronger invariant we will use the Theta invariant [18]. This is a complicated invariant developed by Dror Bar-Natan and Roland van der Veen.

In order to define the Theta invariant, the knot needs to be represented differently first. Although this is not strictly necessary, it will prove useful in order to compute rotation numbers as well. A Sage implementation of the Theta invariant can be found on [19] and it expects the sou-code as part of the input. Therefore, not reformatting would require us to modify the code more than necessary.

Definition 21 (sou code)**.** The idea of the sou code is to have the crossings point in a consistent direction. It is a list of triples of the form [sign, overstrand, understrand].

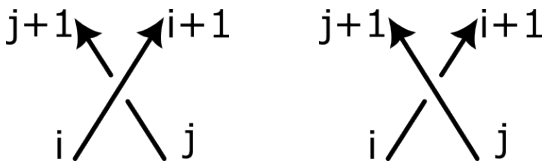


Figure 10: The sou code the the left crossing is [1, i, j] and the sou code of the right crossing is [-1, j, i]

Definition 22 (Rotation numbers)**.** The rotation numbers φ are an array of numbers of size $2 \cdot$ #crossings + 1 . For an upright knot, the n-th index (0-indexed) in the array is the signed number of times the tangent to the n-th edge is horizontal and heading right, with cups counted with 1 signs and caps with -1.

To compute the rotation numbers, we want to start with 1 strand at the bottom, labeled 0, and place crossings such that all crossings are pointing upwards as seen in figure 11. The final edge is also had the label 0, but this one is relabeled to 2 x #crossings.

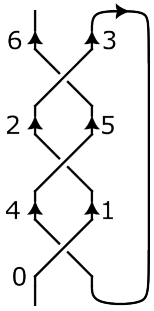


Figure 11: This is the trefoil knot, laid out for the rotation numbers. There is one cap such that its tangent is to the right at edge 3. Therefore the rotation numbers are: φ =[0,0,0,-1,0,0,0]

An important note here is that rotation numbers are not unique. Depending on the ordering of the edges and the way you build the knot, the rotation numbers might change. Fortunately, the Theta invariant is invariant under this. The Theta invariant produces the same result for different pairs of sou-codes and rotation numbers of the same knot.

Definition 23 (Theta invariant [18])**.** The Theta invariant is a Laurent polynomial in two variables and can be constructed using the following formula:

$$\theta(D) := \Delta_1\Delta_2\Delta_3 \left(\sum_c F_1(c) + \sum_{c_0,c_1} F_2(c_0,c_1) + \sum_k F_3(\varphi_k,k) \right)$$

Here, Δ_i are substitutions of the Alexander polynomial.

An implementation of this formula in Sage can be found at [19] and our implementation can be found at 7. The computation is quite involved and due to technical reasons we will be more interested in $\theta(\pi, e)$, where we substitute the first and second variable with π and e respectively. This results in a simple number rather than a lengthy polynomial in two variables.

Example 3 (Theta invariant of the trefoil)**.** Calculating the Theta invariant of the trefoil gives the following expression:

$$\frac{-t_1^4t_2^4 + t_1^4t_2^3 + t_1^3t_2^4 - t_1^4t_2^2 - t_1^2t_2^4 + t_1^3t_2 + t_1t_2^3 - t_1^2 - t_2^2 + t_1 + t_2 - 1}{t_1^2t_2^2}$$

For the simplest knot, we already obtain a complicated expression. If you have a knot with more crossings, the expression becomes even larger. This is the main reason for evaluating Theta. When evaluating this expression at $t_1 = \pi, t_2 = e$, we obtain $\theta(\pi, e)(\text{Trefoil}) \approx -38.292983368$.

5 GRAPH GENERATOR: PLANTRI

Plantri is a program written in C by Gunnar Brinkmann, Heidi Van den Camp and Brendan McKay. Their website and a link to their program can be found at [20]. An easy way to interact with the program without having to compile it yourself can be found at [21], on the combinatorial object server or COS for short.

Plantri is a graph generator capable of generating more than 2,000,000 graphs per second. This speed allows it to exhaustively search and find exactly one member of each isomorphism class. Two graphs are the same if there exists an embedding-preserving isomorphism (possibly reflectional) between them. Plantri is able to generate a lot of different types of graphs some of there include:

- Planar triangulations
- Planar quadrangulations
- Planar simple graphs
- Planar simple bipartite graphs
- Triangulations of a disk
- Planar cubic graphs
- Planar quartic graphs

Within a type it can look at simpleness, connectivity, minimum degree, edge-connectedness and more. The one we are interested in is simple planar quadrangulations, more specifically, their duals.

Definition 24 (Planar quadrangulations and their duals)**.** A planar quadrangulation is a type of graph where every face is enclosed by four edges and 4 vertices. To obtain its dual, you replace every face with a vertex and replace every vertex with a face. Then you place an edge for each pair of faces that were separated from each other by a single edge.

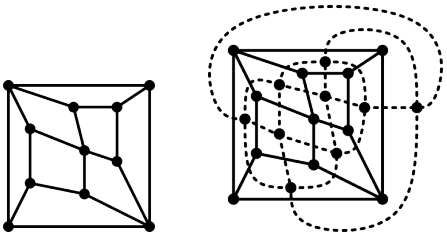


Figure 12: This is an example of a simple quadrangulation on the left. On the right we have the same quadrangulation, but with its dual shown using dotted lines.

If you take a close look at the dual, you will see that every vertex has degree four. This will always be the case when you take the dual of a quadrangulation. This dual is similar to a knot when you forget about the over and undercrossings. This is how the quadrangulations will be useful for us. Plantri can generate a lot of quadrangulations and we can add information about the crossings to make it resemble a knot. But what exactly does plantri output and how does it make these quadrangulations?

The output that the COS website [21] provides and what the program [20] provides are slightly different, but have the same idea. A line of output from COS, when looking at the duals of quadrangulations, looks like this:

$$2 : 1[2\ 2\ 3\ 4]\ 2[1\ 4\ 3\ 1]\ 3[1\ 2\ 4\ 4]\ 4[1\ 3\ 3\ 2]$$

To break it down, $2 :$ is an index. This index is the second entry it shows that is shown by plantri. Then we get four blocks of the form $a[b, c, d, e]$. a is the label of the crossing and b, c, d, e are

the labels of the crossings that are connected to a. These b, c, d, e are ordered clockwise. This is similar to Planar diagram code, but using the vertices instead of the edges.

We will now give an overview on how these simple quadrangulations are generated. More information can be found in this article [22]. The premise is that we start with some basic graphs and apply rules to these graphs that preserve these graphs being simple quadrangulations. The article cover four theorems, each dissecting the rules and starting graph(s) needed to build a class of quadrangulations. For us their first theorem is of importance.

Theorem 6 (Simple quadrangulations [22])**.** *The class of all simple quadrangulations of the sphere is generated from the square, as seen in figure (14), by the following two rules:*

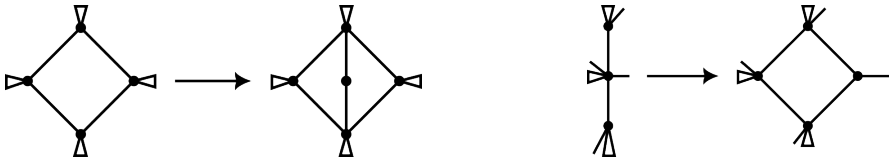


Figure 13: On the left we have the first rule and on the right we have the second rule. A small triangle attached to a vertex denotes an arbitrary amount of edges including 0

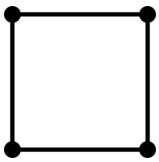


Figure 14: A simple square used as a starting point

Proof. The idea of the proof is to work backwards. They show that any graph in the class of simple quadrangulations falls in one of three cases. Either it is the square, it is the result of rule one or it is the result of rule two. This way for every graph in this class a path can be made starting from the square using either of the two rules. □

Their article also discusses the dual classes. The dual of the class in theorem 6 is the class of 4-regular, 4-edge-connected planar graphs (which are not necessarily simple). The 4-edge-connected property is great to have. Since a connected sum of two graphs is always 2-edge-connected, these are excluded from the class. As a result, when we make knots from these graphs, we will only obtain prime knots.

6 BUILDING A KNOT TABLE

It all starts with gathering the graph data provided by plantri. Specifically, we want dual graphs of the class of planar simple quadrangulations. As discussed earlier, this is the class of 4-regular, 4-edge-connected planar graphs (which are not necessarily simple) and will result in only prime knots when we convert them to knots. We will keep track of the size of our data in the following way, we make a list of the form [#crossings, #knots]. Here, the amount of crossings (or vertices before they become crossings) will be a number between 2 and 10 and the amount of knots refers to amount of data (in the form of knots or graphs) we have. As this step we have the following:

$$[[2,1],[3,1],[4,2],[5,3],[6,9],[7,18],[8,62],[9,198],[10,803]]$$

First of all, there are links among the knots. We can filter those out even before adding crossings to the graphs. We do this by constructing a Gauss code by treating the vertices of the graphs as if they were knot crossings. This significantly reduces the amount of graphs we have left.

$$[[3,1],[4,1],[5,2],[6,3],[7,10],[8,27],[9,101],[10,364]]$$

As you can see, there are no knots with 2 crossings. the one that was present before was actually the Hopf link. Now we add a sign to each vertex of the graph to have them actually represent knots. This exponentially increases the amount of knots. For each graph with n vertices, we obtain 2^n graphs in return. In this step we will also convert all knots to the more standard PD code, this has no influence on the amount of knots.

$$[[3,8],[4,16],[5,64],[6,192],[7,1280],[8,6912],[9,51712],[10,372736]]$$

Next we want to go through various steps of reducing the amount of knots we have left. We will do this by applying easy to check reductions and make them more complex as we go on. This order is important as the more complex methods take more time to perform than the easier methods. The basic idea is to delete all knots that can be reduced to a knot with a smaller amount of crossings. The easiest check is to check if we can successfully apply the second Reidemeister move anywhere on the knot. If we can apply it we can remove the knot. Note that we only have to check if it is possible, we do not have to apply the transformation. This simple check has a big impact. As we can see in the following list, the amount of knots of 10 crossings have been reduced by over 90%.

$$[[3,2],[4,4],[5,6],[6,28],[7,154],[8,952],[9,5694],[10,34424]]$$

For the next step, we apply the third Reidemeister move. For a given knot, we continuously and recursively apply R3 to every triplet of crossings we can and check if this opens an opportunity to apply either the first or second Reidemeister move. This was also successful as it again produces a reduction of almost 90% to knots of 10 crossings. We also chose a canonical representation for knots in this step much like example 2. This had no influence on the amount of knots in this step.

$$[[3,2],[4,2],[5,4],[6,6],[7,26],[8,152],[9,696],[10,3558]]$$

Now that all knots are in our canonical representation, we noticed that there were multiple knots with the same representation. In this step we simply clean up this duplication. This had a small but noticeable impact of about 5% in the case of 10 crossings.

$$[[3,2],[4,1],[5,4],[6,5],[7,26],[8,122],[9,628],[10,3381]]$$

The next idea is to check if a two-one pass is possible. This requires building the knot a lot of times and is therefore a more expensive operation than the previous step. If we can apply the two-one pass, we discard the knot, otherwise we keep it. This gives a reduction of about 32% for knots of 10 crossings.

$$[[3,2],[4,1],[5,4],[6,5],[7,20],[8,95],[9,438],[10,2293]]$$

It is now time to store our knots in a different way. Instead of listing all knots in plain text in a text file, we will make a dictionary based on the Jones polynomial of the knots. This will

group all knots with the same Jones polynomial together. In this step we will also remove mirror images of knots. We can do this by looking at the Jones polynomial. Recall that if we substitute $1/t$ for t, we get the Jones polynomial of the mirror image. Unfortunately, this method can not remove all mirror images. For example, for knot 9_{42} we have that its Jones polynomial is $t^3 - t^2 + t - 1 + \frac{1}{t} - \frac{1}{t^2} + \frac{1}{t^3}$ and this knot is not amphicheiral. This means that this knot and its mirror image go in the same group when one can not be turned into the other. Another operation we perform in this step is reducing the group sizes by applying flyping transformations to the first element of the group. If this results in an element that is on the group, we can remove that element. This new method of storing knots allows us to more precisely state our data. We will now assign three numbers to each dataset. The list will be of the form [#crossings, #groups, #knots] where groups will be the amount of different groups and knots will be sum of all knots of all groups. The amount of groups will also allow us to compare with different tables and see our progress. So far, knots up to and including 7 crossings have hit the limit. Flyping the first element and comparing the the rest of the group gave a reduction of about 55% for 10 crossings.

$$[[3,1,1],[4,1,1],[5,2,2],[6,3,3],[7,7,7],[8,23,42],[9,59,180],[10,196,1015]]$$

As mentioned in the previous section, there are some flaws with the previous method of using the Jones polynomial. It will put a knot and its mirror image in the same group if Jones(t) = Jones($1/t$). Another issue is that two different knots can have the same Jones polynomial. For example, this is the case for knots 10_{25} and 10_{56}. We can solve these issues by looking at the Theta invariant. For each group we compute the Theta invariant of all knots and split them into more groups based on Theta. If we have a group with Theta invariant a and another with $-a$, we know that they are mirror images of each other and we can delete one of the groups. This process results in more groups but a marginally lower amount of total knots.

$$[[3,1,1],[4,1,1],[5,2,2],[6,3,3],[7,7,7],[8,23,42],[9,59,159],[10,209,985]]$$

For the last step, we will apply all methods we can at the knots. For each group, we take the first knot. On this knot, we apply the third Reidemeister move in all possible places and apply all possible flypes. We then check for all the resulting knots if it is reducible by either R1, R2 or the two-one pass. If it is reducible we delete the entire group. Note that this deletion does make a big assumption. At this point we assume that all knots of the same group are in fact the same knot. Given the strength of the Jones polynomial and Theta invariant and the amount of crossings we are dealing with, we believe this is an acceptable assumption to make. After checking reducibility, we remove elements from the group that appear as a result from flyping or R3. Then we apply this whole procedure to all knots that appear as a result from applying a flype or R3 to the first knot. Then repeat that process 10 times. This gives us the following final results:

$$[[3,1,1],[4,1,1],[5,2,2],[6,3,3],[7,7,7],[8,21,24],[9,49,70],[10,169,425]]$$

For knots up to 7 crossings, our results perfectly matches different tables like Tait's table and the table found on [1]. For knots with 8 or 9 crossings, we get the correct amount of groups, but we were not able to reduce these groups to 1 element. Given that we already made the assumption that these knots must be the same knot based on their Jones polynomial and Theta invariant, this is still a successful result. Lastly we have the knots with 10 crossings. We found 169 groups of knots, which is 4 more then the 165 knots that can be found on [1]. Through comparing Jones polynomials, we know which four these are. Interestingly enough, all four should be reducible to 8 crossings and three out these four are amphichiral. These knots, under our labeling system, are $10_{94}, 10_{96}, 10_{128}$ and 10_{146} which should be reducible to $8_{21}, 8_{20}, 8_{17}$ and 8_3 respectively.

There are ways to improve upon this result. You can perform a deeper search at the places we performed a depth search. You can also implement more different moves. For example you can make various k-l-pass moves. Another technique you can do is to first add one or more crossings (for example by the first Reidemeister move) and then reduce.

7 CODE

The language of choice is Python. Although Python is slower than a language like C, it is significantly easier to work with. For the scale that we are working with, Python is fast enough. If we would want to expand to significantly larger knots, then rewriting all functions to C would be more advantageous.

The code itself can be found on the following GitHub repository [23]. In order to run the code yourself, you will need to have Python 3.10.0 installed [24] and you will need the following packages: re, ast, copy, sympy, symengine, math, pickle, numpy and pygame.

We start with the initial data that we obtain from either COS [21] or compiling the source code [20]. Note that for the latter one needs a C compiler and delete some Linux specific code if compiling on windows. Specifically the library "times.h" and functions using that library need to be deleted.

```
# Reads Plantri output, returns the data as proper nested lists
def Read_Plantri_Data(s: str) -> list[list[int]]:
    pattern = re.findall(r'(\d+)\[(.*?)\]', s)
    result = []

    for key, values in pattern:
        # Convert space-separated numbers to a list of integers
        numbers = list(map(int, values.split()))
        result.append([int(key), numbers])

    return result
```

From the Plantri data, we need to transform it into something more usable. We want to make a transformation into nested lists:

$$2 : 1[2234]2[1431]3[1244]4[1332] \rightarrow [[1,[2,2,3,4]],[2,[1,4,3,1]],[3,[1,2,4,4]],[4,[1,3,3,2]]]$$

We need the extra brackets in order to find all the data through indexes. The above code block allows us to do this through regular expressions or regex for short. More information on what regex is or what you can do with it can be found here [25].

```
# Knot in plantri-format --> Gauss-like code
def get_gauss(knot: list[list[int]]):
    # Start point
    gauss = [[1, 0]]

    # Start by choosing into the direction of knot[0][1][0] Ususally 2
    crossing = knot[0][1]
    # If 2 appears once
    if crossing.count(crossing[0]) == 1:
        gauss.append([crossing[0], 1])
    # If 2 appears twice
    else:
        if crossing[1] == crossing[0]:
            gauss.append([crossing[0], "-"])
        else:
            gauss.append([crossing[0], "+"])

    done = False
    while not done:
        # Pick the next crossing
        next_crossing = knot[gauss[-1][0] - 1][1] # -1 because 0-indexing
        index = next_crossing.index(gauss[-2][0])
        # If the next value is not unique, find where we came from
        if next_crossing.count(gauss[-2][0]) != 1:
            if gauss[-1][1] == "-":
                if next_crossing[index] == next_crossing[index + 1]:
                    index += 1
            else:
                if next_crossing[index] != next_crossing[index + 1]:
                    index = 3
```

```
        # If where we go to is unique, write it
        if next_crossing.count(next_crossing[(index + 2) % 4]) == 1:
            gauss.append([next_crossing[(index + 2) % 4], gauss[-1][0]])
        # Else, find where we go to
        else:
            if next_crossing[(index + 2) % 4] == next_crossing[(index + 3) % 4]:
                gauss.append([next_crossing[(index + 2) % 4], "-"])
            else:
                gauss.append([next_crossing[(index + 2) % 4], "+"])

        # If we are back where we started, we made a full loop
        if gauss[-1] == gauss[1]:
            done = True
            gauss[0] = gauss[-2]
            return gauss
```

Now we want to filter out all links. We can do this by constructing a Gauss-like code and examining its length. We choose our starting point to be vertex 1 and we look into the direction of vertex 2. Unfortunately, we will have to handle graphs like this one:

$$[[1,[2,2,3,4]],[2,[1,4,3,1]],[3,[1,2,4,4]],[4,[1,3,3,2]]] \tag{2}$$

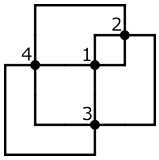


Figure 15: The graph corresponding to code 2

Duplicate elements show up as we could go from one vertex to another in two ways. We fix this by keeping track of whether we came from the left connection or the right connection. When starting with 1 and going up towards 2, we need to keep track of the "left" side, denoted with a "-". Then we can use this information that the next step is towards 3 instead of 4. Below we find the Gauss-like code of this graph.

$$[[1,3],[2,'-'],[3,2],[4,'+'],[2,4],[1,'-'],[4,1],[3,'+'],[1,3],[2,'-']]$$

```
# 1.Plantri_Data --> 2.Plantri_knots
def print_knots():
    for i in range(3,11):
        count = 0
        file = open(f"1.Plantri_Data/{i}.txt", "r")
        new_file = open(f"2.Plantri_knots/{i}.txt", "w")
        for line in file:
            knot = Read_Plantri_Data(line.rstrip()[3:])
            gauss_code = get_gauss(knot)
            if len(gauss_code) == ((len(knot)*2) + 2):
                new_file.write(f"{knot}\n")
                count += 1
```

All data gathered from Plantri are stored in a folder. This folder contains files for each crossing number. The above code block is almost a template on how we generate the next folder. We go through each file, then go through each line. We apply some logic to it and decide if it makes it to the next folder. In this case we do two things. First, we reformat the graph using "Read_Plantri_Data()". If this passes the check, we write it in its new form. The check we perform, checks if the graph forms a link or not. If the Gauss code goes through all vertices twice, it cannot be a link. Note that there is a $+2$ as our Gauss code produces two more elements than necessary.

```
# 2.Plantri_knots --> 3.Plantri_knots_with_sign
# Adds a sign to each crossing (all combinations)
def add_sign_to_crossing():
    def print_binary_sequences(n, knot):
        if n < 0:
            new_file.write(f"{knot}\n")
```

```
        else:
            # Recursively call this function untill the knot is filled in
            knot_1 = copy.deepcopy(knot)
            knot_1[n].append("+")
            print_binary_sequences(n - 1, knot_1)
            knot_2 = copy.deepcopy(knot)
            knot_2[n].append("-")
            print_binary_sequences(n - 1, knot_2)

    # Go through each file and line to apply print_binary_sequences()
    for i in range(3,11):
        file = open(f"2.Plantri_knots/{i}.txt", "r")
        new_file = open(f"3.Plantri_knots_with_sign/{i}.txt", "w")
        for line in file:
            knot = ast.literal_eval(line)
            print_binary_sequences(i-1, knot)
        file.close()
        new_file.close()
```

Next up, we want to make knots. To make a knot from a graph we need to add information about the crossings. A crossing can either be a positive or negative crossing. For a graph of n crossings, we will obtain 2^n knots by producing all combinations. The helper function is inlined into the function as it was simpler to put the "write" statement in the helper function. This function transforms a code like 2 into the following, but with 16 copies in total with each having a different combination of signs.

$$[[1,[2,2,3,4],'+'],[2,[1,4,3,1],'-'],[3,[1,2,4,4],'+'],[4,[1,3,3,2],'-']]$$

```
# Returns a list following the knot, listing if the crossing goes over or under
def over_under_list(knot):
    ou_list = []
    seen = []
    gauss = get_gauss(knot)
    # Loop over the Gauss list (without the two duplicates at the end of the list)
    for i in range(0,len(gauss) - 2):
        # If we have seen this vertex before
        if gauss[i][0] in seen:
            crossing = knot[gauss[i][0] - 1][1]
            sign = knot[gauss[i][0]-1][2]
            idx_first = -1
            idx_second = -1
            last_j = -1
            # Find the vertex in the list that we have seen before
            for j in range(len(gauss) - 2):
                if gauss[j][0] == gauss[i][0]:
                    # If that previous vertex had a duplicate entry, there is a "+" or "-" we need to
                                                                        account for
                    if isinstance(gauss[j+1][1], str):
                        if gauss[j+1][1] == "+":
                            idx_first = crossing.index(gauss[j+1][0])
                            if idx_first != 0:
                                idx_first += 1
                            last_j = j
                            break
                        else:
                            idx_first = crossing.index(gauss[j+1][0])
                            if idx_first == 0:
                                if crossing[1] != gauss[j+1][0]:
                                    idx_first = 3
                            last_j = j
                            break
                    else:
                        idx_first = crossing.index(gauss[j+1][0])
                        last_j = j
                        break
            # If the new vertex has a duplicate entry, there is a "+" or "-" we need to account for
            if isinstance(gauss[i+1][1],str):
                if gauss[i+1][1] == "+":
                    idx_second = crossing.index(gauss[i+1][0])
                    if idx_second != 0:
                        idx_second += 1
```

```
                else:
                    idx_second = crossing.index(gauss[i+1][0])
                    if idx_second == 0:
                        if crossing[1] != gauss[i+1][0]:
                                idx_second = 3
            else:
                idx_second = crossing.index(gauss[i+1][0])

            # Decide if we go over or under for both the previous instance and the current one
            if idx_first == ((idx_second + 1)%4):
                if sign == "+":
                    ou_list[last_j] = [gauss[i][0],"over"]
                    ou_list.append([gauss[i][0],"under"])
                else:
                    ou_list[last_j] = [gauss[i][0],"under"]
                    ou_list.append([gauss[i][0],"over"])
            else:
                if sign == "+":
                    ou_list[last_j] = [gauss[i][0],"under"]
                    ou_list.append([gauss[i][0],"over"])
                else:
                    ou_list[last_j] = [gauss[i][0],"over"]
                    ou_list.append([gauss[i][0],"under"])

        # If we have not seen this vertex
        else:
            # Add it to both lists
            seen.append(gauss[i][0])
            # The 0 here is a dummy value
            ou_list.append([gauss[i][0],0])

    return(ou_list)
```

This function will help us to transform the knots from plantri code to the more standard PD code. From the Gauss-like code, we already know in what order the crossings are found when traversing the knot. While reading the Gauss code, the second time we encounter a particular crossing, we can use the sign of the crossing to determine whether the strand passed over or under at that point.

```
# Converts a Plantri-knot to Planer diagram notation
# vertex based --> edge based
def plantri_to_PD(knot):
    ou_list = over_under_list(knot)
    pd_code = []
    # We want the code to not contain the the number 0
    for i in range(1, len(knot)+1):
        pd_segment = [-1,-1,-1,-1]
        idx_under = -1
        idx_over = -1
        sign = knot[i-1][2]
        # find the other strand of the crossing
        j=0
        while idx_over == -1 or idx_under == -1:
            if ou_list[j][0] == i:
                if ou_list[j][1] == "under":
                    idx_under = j
                else:
                    idx_over = j
            j+=1
        if idx_under == 0:
            idx_under += 2*len(knot)
        if idx_over == 0:
            idx_over += 2*len(knot)
        # Fill in pd-code
        pd_segment[0] = idx_under
        pd_segment[2] = (idx_under )%(2*len(knot)) + 1
        if sign == "+":
            pd_segment[1] = (idx_over )%(2*len(knot)) + 1
            pd_segment[3] = idx_over
        else:
            pd_segment[1] = idx_over
            pd_segment[3] = (idx_over )%(2*len(knot)) + 1
```

```
        pd_code.append(pd_segment)
    return sorted(pd_code)
```

Using the last function we can now transform knots from plantri code to PD code. This involves a structural change by looking at the edges instead of the vertices. You can give each element in the over_under_list() a number according to its index. For example, the first element is 1, the second is 2 etc. These numbers will correspond to the labels of the edges of our PD code. This function finds the two elements of over_under_list(), that corresponds with a crossing and builds the PD code of that crossing with this idea. The information about which strand is under/over and the sign of the crossing determines the exact order in which they have to be listed in the PD code.

```
# 3.Plantri_knots_with_sign --> 4.PD_knots
# converts all knots to PD notation
def PD_knots():
    for i in range(3,11):
        file = open(f"3.Plantri_knots_with_sign/{i}.txt", "r")
        new_file = open(f"4.PD_knots/{i}.txt", "w")
        for line in file:
            knot = ast.literal_eval(line)
            knot = plantri_to_PD(knot)
            new_file.write(f"{knot}\n")
        file.close()
        new_file.close()
```

Now that we can transform knots from plantri code to PD code, we apply this to every knot. This function is similar to print_knots() and will be similar to all function that write from one file to another. We read a line, transform that knot to PD code and write this new code to the new file.

```
# Returns True if a knot in PD notation containts a trivial application of R2
def contains_trivial_R2(knot):
    for i in range(len(knot)):
        if (knot[i][0] + 1)%(2*len(knot)) == (knot[(i+1)%len(knot)][0])%(2*len(knot)):
            temp = knot[i] + knot[(i+1)%len(knot)]
            if len(set(temp)) < 7:
                return True
    return False
```

Now that we have our knots in PD notation, it is time to filter out knots that can be simplified. The above code block checks if the second Reidemeister move can be applied. The first if statement checks for two consecutive understrands, the second if statement checks if the amount of uniquely labeled edges is less then 7. Two consecutive knots always share an edge, therefore the amount of unique elements is at most 7. When they share two edges and if they have two consecutive understrands, the second Reidemeister move is possible. There is one more edge case. What if the two shared edges are not next to each other? In that case they form a loop that is separate from the rest of the knot, a link. This case was already ruled out by print_knots() using the Gauss-code.

```
# 4.PD_knots --> 5.PD_No_R2
# Removes all knots that contain a trivial application of R2
def remove_trivial_R2():
    for i in range(3,11):
        file = open(f"5.PD_knots/{i}.txt", "r")
        new_file = open(f"6.PD_No_R2/{i}.txt", "w")
        for line in file:
            knot = ast.literal_eval(line)
            if not contains_trivial_R2(knot):
                new_file.write(f"{knot}\n")
        file.close()
        new_file.close()
```

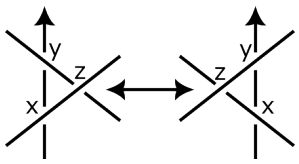
Here we have another file-write block. This block applies contains_trivial_R2() and sorts out the knots that can be reduced.

```
# Applies R3 on the all triplets of crossings that allows a R3
def apply_R3(knot):
    knot_list = []
    len_knot = len(knot)
```

```
for i in range(len_knot):
    # look for 2 consecutive understrands
    if (knot[i][0] + 1)%(2*len_knot) == (knot[(i+1)%len_knot][0])%(2*len_knot):
        # look for third crossing of the triangle
        for j in range(len_knot):
            # third crossing cant be either of the first 2
            if knot[j] != knot[i] and knot[j] != knot[(i+1)%len_knot]:
                # check if it becomes a triangle
                if knot[i][1] in knot[j] or knot[i][3] in knot[j]:
                    if knot[(i+1)%len_knot][1] in knot[j] or knot[(i+1)%len_knot][3] in knot[j]:
                        # apply R3

                        x = knot[i]
                        y = knot[(i+1)%len_knot]
                        z = knot[j]

                        # deduce the specific r3 application we have
                        side = False
                        if (x[1] in z) and (y[1] in z):
                            side = "right"
                        elif (x[3] in z) and (y[3] in z):
                            side = "left"
                        else:
                            print("Side did not resolve properly")
                            break

                        top_left_to_down_right = False
                        if side == "right":
                            if (y[3])%(2*len_knot) == (y[1]+1)%(2*len_knot):
                                top_left_to_down_right = "left"
                            else:
                                top_left_to_down_right = "right"
                        else:
                            if (x[3])%(2*len_knot) == (x[1]+1)%(2*len_knot):
                                top_left_to_down_right = "left"
                            else:
                                top_left_to_down_right = "right"

                        down_left_to_top_right = False
                        if side == "right":
                            if (x[3])%(2*len_knot) == (x[1]+1)%(2*len_knot):
                                down_left_to_top_right = "left"
                            else:
                                down_left_to_top_right = "right"
                        else:
                            if (y[3])%(2*len_knot) == (y[1]+1)%(2*len_knot):
                                down_left_to_top_right = "left"
                            else:
                                down_left_to_top_right = "right"


                        part_of_z_understrand = False
                        if side == "right":
                            if (x[1] == z[0]) or (x[1] == z[2]):
                                part_of_z_understrand = "i"
                            else:
                                part_of_z_understrand = "i+1"
                        else:
                            if (x[3] == z[0]) or (x[3] == z[2]):
                                part_of_z_understrand = "i"
                            else:
                                part_of_z_understrand = "i+1"

                        # apply r3 based on the specific application
                        if side == "right":
                            if down_left_to_top_right == "right":
                                new_x = [y[0], ((x[1])%(2*len_knot)) + 1, y[2], x[1]]
                            else:
                                new_x = [y[0], ((x[1] - 2)%(2*len_knot)) + 1, y[2], x[1]]

                            if top_left_to_down_right == "right":
                                new_y = [x[0], ((y[1])%(2*len_knot)) + 1, x[2], y[1]]
                            else:
```

```
                        new_y = [x[0], ((y[1] - 2)%(2*len_knot)) + 1, x[2], y[1]]

                else:
                    if top_left_to_down_right == "right":
                        new_x = [y[0], x[3], y[2], ((x[3] - 2)%(2*len_knot)) + 1]
                    else:
                        new_x = [y[0], x[3], y[2], ((x[3])%(2*len_knot)) + 1]

                    if down_left_to_top_right == "right":
                        new_y = [x[0], y[3], x[2], ((y[3] - 2)%(2*len_knot)) + 1]
                    else:
                        new_y = [x[0], y[3], x[2], ((y[3])%(2*len_knot)) + 1]

                if side == "right":
                    if part_of_z_understrand == "i":
                        if down_left_to_top_right == "right":
                            new_z = [x[3], y[1], x[1], y[3]]
                        else:
                            new_z = [x[1], y[3], x[3], y[1]]
                    else:
                        if top_left_to_down_right == "right":
                            new_z = [y[3], x[3], y[1], x[1]]
                        else:
                            new_z = [y[1], x[1], y[3], x[3]]
                else:
                    if part_of_z_understrand == "i":
                        if top_left_to_down_right == "right":
                            new_z = [x[3], y[3], x[1], y[1]]
                        else:
                            new_z = [x[1], y[1], x[3], y[3]]
                    else:
                        if down_left_to_top_right == "right":
                            new_z = [y[3], x[1], y[1], x[3]]
                        else:
                            new_z = [y[1], x[3], y[3], x[1]]

                # replace the previous three crossings with the new ones
                new_knot = knot.copy()
                new_knot[i] = new_x
                new_knot[(i+1)%len_knot] = new_y
                new_knot[j] = new_z

                knot_list.append(sorted(new_knot))
    return knot_list
```

Unfortunately, applying the third Reidemeister move is not as simple as the second. This is mostly because we actually have to do the transformation instead of only locating it. Furthermore, we want to find every possible application of the third Reidemeister move. To find a place where we can apply the third Reidemeister move, we first look for two consecutive understrands. Then go through all remaining crossings to check if they form a triangle. With this, we mean that there exists a strand in the first crossing and there exists a strand in the second crossing that both are in another crossing.



After finding a place to perform R3, we need to gather data on how the actions should be performed. We always knot where the two consecutive understrands are, they are part of crossings x and y. Their direction is also consistent. All other factors can vary. The third crossing (z) can be on the left or right, z can be a positive or negative crossing and the other two strands can go in either direction. All these factors impact how the PD code change. Therefore we need a lot of if statements to hone in on the specific case we are in and then apply R3 correctly.

```
# Returns True if a knot in PD notation contains a trivial application of R1
def contains_R1(knot):
    for i in range(len(knot)):
```

```
        if len(set(knot[i])) < 4:
            return True
    return False
```

It might be possible that after applying the third Reidemeister move, the first one becomes possible. If a crossing contains the same strand twice, then R1 is possible.

```
# Make a list containing all representations of a given knot
# Accounts for relabeling and change of orientation
def all_representations(knot: list):
    all_list = [knot]
    len_knot = len(knot)
    # Accounting for relabeling
    for i in range(1, 2*len_knot):
        knot_copy = copy.deepcopy(all_list[-1])
        knot_copy = [[(x % (2*len_knot)) + 1 for x in sublist] for sublist in knot_copy]
        all_list.append(sorted(knot_copy))

    knot_copy = copy.deepcopy(all_list[-1])
    # Change orientation
    reflected = [[(2*len_knot + 1) - x for x in sublist] for sublist in knot_copy]
    other_orientation = [sublist[2:] + sublist[:2] for sublist in reflected]
    all_list.append(sorted(other_orientation))

    # Accounting for relabeling after changing the orientation
    for i in range(1, 2*len_knot):
        knot_copy = copy.deepcopy(all_list[-1])
        knot_copy = [[(x % (2*len_knot)) + 1 for x in sublist] for sublist in knot_copy]
        all_list.append(sorted(knot_copy))

    return all_list
```

PD code is not unique, it depends on the starting edge and direction. When comparing two knots, we want a canonical way to represent a specific set of crossings. We do this by first making all possible representations.

```
# Transformes a knot into its canonical representation
def canonical_representation(knot: list):
    return sorted(all_representations(knot))[0]
```

Then we sort the list of all representations and choose the first element. This canonical representation is the lexicographically smallest one.

```
# Continuously applies apply_R3 to a knot to find instances where R1 or R2 is possible
# Returns False if reducible
# Returns the knot in canonical form if not reducible
def deep_R3_search(knot: list, search_depth: int):
    depth = 1
    knot = canonical_representation(knot)
    knot_list = [knot]

    # Search untill the desired depth
    while (depth < search_depth):
        depth += 1
        new_list = []
        # Fill list with all unique application of R3
        for new_knot in knot_list:
            R3_list = apply_R3(new_knot)
            for R3_knot in R3_list:
                new_R3_knot = canonical_representation(R3_knot)
                if new_R3_knot in new_list:
                    continue
                new_list.append(new_R3_knot)
        knot_list = new_list

        # If the knot is reducible (by R1 or R2) we are done
        for new_knot in knot_list:
            if (contains_R1(new_knot)) or (contains_trivial_R2(new_knot)):
                return False

    # We assume the knot to be non-reducible
    return knot
```

One application of apply_R3() might not be enough to find all possible knots that can be obtained through R3. This function continuously applies R3 to a knot. If any of the resulting knots is reducible by R2 or R1, we knot that all of them are and we can discard this knot. Otherwise we will return the canonical representation of the knot.

```
# 5.PD_No_R2 --> 6.Reduced_knots
# Attempts to find reducible knots through R3
# Prints all knots that can not be reduced (before reaching search_dept) in canonical form
def reduce_knots_by_R3():
    search_depth = 10
    for i in range(3,11):
        file = open(f"6.PD_No_R2/{i}.txt", "r")
        new_file = open(f"7.Reduced_knots/{i}.txt", "w")
        for line in file:
            knot = ast.literal_eval(line)
            search = deep_R3_search(knot, search_depth)
            if search == False:
                continue
            else:
                new_file.write(f"{search}\n")
        file.close()
        new_file.close()
```

The above codeblock applies the previous function to all knots and stores the results in a new folder.

```
# 6.Reduced_knots --> 7.no_dupes
# Some knots had the same canonical representation and were therefore the same knot
def remove_duplicates():
    for i in range(3,11):
        knot_list = []
        file = open(f"6.Reduced_knots/{i}.txt", "r")
        new_file = open(f"7.no_dupes/{i}.txt", "w")
        for line in file:
            knot = ast.literal_eval(line)
            if knot in knot_list:
                continue
            knot_list.append(knot)
            new_file.write(f"{knot}\n")
        file.close()
        new_file.close()
```

Now that all knots are stored in a canonical representation, we found that there are duplicate entries. These will be filtered out with the above code block. Every knot that is seen will be stored in knot_list. Then if we find a knot that is already in that list, we skip writing it to the file.

```
# Constucts knot to check if a 2-1 pass is valid from a specific location
def is_two_one_pass_valid(knot: list, front: list, end_list: list):
    len_front = len(front)
    end_len = len_front - 2
    next_front = front

    while len(front) != 0:
        draw_list = []
        skip_next = False
        front = next_front
        len_front = len(front)

        # We are done if our front is of lenght 1
        if len_front == end_len:
            return True
        # or if our front contains a number that it shouldnt (exampe: otherside of startfront)
        for number in front:
            if number in end_list:
                return False

        # create next front and draw list
        for i in range(len_front):
            # skip next number if we have placed cap/cross
            if skip_next:
                skip_next = False
                continue
```

```
        if i < len(front) - 1:
            # can we place a cap?
            if front[i] == front[i+1]:
                draw_list.append("cap")
                skip_next = True
                next_front.pop(i)
                next_front.pop(i)
                continue

            # no placable knots = only caps and lines can be drawn
            if len(knot) == 0:
                draw_list.append("line")
                continue

            found = False
            for crossing in knot:
                # find other half edge
                if front[i] in crossing:
                    found = True
                    # can we place cross?
                    if front[i+1] in crossing:
                        idx_i = crossing.index(front[i])
                        idx_i2 = crossing.index(front[i+1])
                        if idx_i +1 != idx_i2:
                            next_front[i] = crossing[(idx_i+1)%4]
                            next_front[i+1] = crossing[(idx_i2-1)%4]
                        else:
                            next_front[i] = crossing[(idx_i-1)%4]
                            next_front[i+1] = crossing[(idx_i2+1)%4]

                        # make a cross
                        if crossing[0] == next_front[i] or crossing[2] == next_front[i]:
                            draw_list.append("cross_under")
                        else:
                            draw_list.append("cross_over")

                        skip_next = True
                        knot.remove(crossing)
                        break
                    # if a crossing can be placed, but not a cross
                    else:
                        draw_list.append("line")
                        break
            # if no crossings can be placed
            if not found:
                draw_list.append("line")
                continue
        # At the end of the front, only a line can be drawn
        else:
            draw_list.append("line")

    # If we can only draw lines with the rules above, we need to place a fork
    if len(set(draw_list)) == 1 and draw_list[0] == "line":
        done = False
        for i in range(len(front)):
            if done:
                done = False
                break
            for crossing in knot:
                if front[i] in crossing:
                # find other half edge
                    idx = crossing.index(front[i])
                    next_front.pop(i)
                    next_front.insert(i, crossing[(idx - 1)%4])
                    next_front.insert(i, crossing[(idx - 2)%4])
                    next_front.insert(i, crossing[(idx - 3)%4])
                    knot.remove(crossing)
                    draw_list[i] = "fork"
                    done = True
                    break
```

For our next reduction step, we check if we can apply a two-one pass. The above function does that by starting with a front of length three, then building the knot. How we build the knot exactly will be explained later on when we draw the knot to the screen. If we end up with a front of size one before we use an edge we should not use, we return true. To understand why we start with a front of size three, we have the following example:

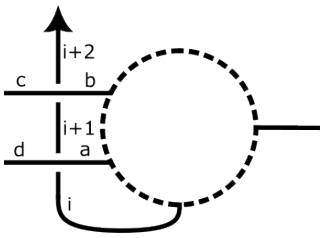


Figure 16: Example of when a two-one pass might be possible

To be a two-one pass we need 2 strands going in and one strand going out. But actually we have a third one going in as part of our front. We have a, b and i going in and only one going out. If either c or d is part of the dotted circle we know that a two-one pass is not possible.

```
# Checks all cases where 2-1 might be applicable
# returns True if 2-1 pass can be applied, False otherwise
def check_two_one_pass(knot):
    len_knot = len(knot)
    for i in range(len_knot):
        # look for 2 consecutive understrands
        if (knot[i][0] + 1)%(2*len_knot) == (knot[(i+1)%len_knot][0])%(2*len_knot):
            knot_copy = copy.deepcopy(knot)
            crossing_1 = knot_copy[i]
            crossing_2 = knot_copy[(i+1)%len_knot]
            knot_copy.remove(crossing_1)
            knot_copy.remove(crossing_2)
            # Given two consecutive understrands, there are four cases to check for 2-1 pass
            if is_two_one_pass_valid(copy.deepcopy(knot_copy), [crossing_1[0], crossing_1[1], crossing_2
                                                                [1]], [crossing_1[3], crossing_2[3]]):
                return True
            if is_two_one_pass_valid(copy.deepcopy(knot_copy), [crossing_1[1], crossing_2[1], crossing_2
                                                                [2]], [crossing_1[3], crossing_2[3]]):
                return True
            if is_two_one_pass_valid(copy.deepcopy(knot_copy), [crossing_2[2], crossing_2[3], crossing_1
                                                                [3]], [crossing_1[1], crossing_2[1]]):
                return True
            if is_two_one_pass_valid(copy.deepcopy(knot_copy), [crossing_2[3], crossing_1[3], crossing_1
                                                                [0]], [crossing_1[1], crossing_2[1]]):
                return True
    return False
```

There are four cases in which a two-one pass might be possible given two consecutive understrands. From figure 16, the other cases are where a, b and $i+2$ are part of the front (vertically flipped), c, d and i are part of the front (horizontally flipped) and c, d and $i+2$ are part of the front (horizontally and vertically flipped). This function applies all four cases to each pair of consecutive understrands. If at any point is_two_one_pass_valid() returns True, we knot that the knot can be reduced and we do not have to check any other cases.

```
# 7.no_dupes --> 8.pass_reduced
def reduce_by_pass():
    for i in range(3,11):
        file = open(f"7.no_dupes/{i}.txt", "r")
        new_file = open(f"8.pass_reduced/{i}.txt", "w")
        for line in file:
            knot = ast.literal_eval(line)
            if check_two_one_pass(knot):
                continue
            new_file.write(f"{knot}\n")
        file.close()
        new_file.close()
```

Now that we have a function to check for two-one pass we apply it to all knots and only write the knots that are not reducible by this method.

```
# Returns the Kauffman bracket Polynomial
def kauffman(knot: list):
    # Break down crossings to line segments
    for crossing in knot:
        if len(crossing) == 4:
            # recursivly call kauffman
            knot_a = copy.deepcopy(knot)
            knot_a.remove(crossing)
            knot_a.append([crossing[0], crossing[1]])
            knot_a.append([crossing[2], crossing[3]])

            knot_b = copy.deepcopy(knot)
            knot_b.remove(crossing)
            knot_b.append([crossing[1], crossing[2]])
            knot_b.append([crossing[3], crossing[0]])
            poly = A*kauffman(knot_a) + (A**(-1)) * kauffman(knot_b)
            return expand(poly)
    # Count the amount of circles you can form with the line segments
    circles = 0
    while len(knot) > 0:
        if len(set(knot[0])) == 1:
            circles += 1
            knot.pop(0)
            continue
        for i in range(1,len(knot)):
            if len(set(knot[0] + knot[i])) == 3:
                new_crossing = list((set(knot[0]) | set(knot[i])) - (set(knot[0]) & set(knot[i])))
                knot.pop(i)
                knot.pop(0)
                knot.append(new_crossing)
                break
            elif len(set(knot[0] + knot[i])) == 2:
                circles += 1
                knot.pop(i)
                knot.pop(0)
                break
    return ((-(A**2) - (A**(-2)))**(circles - 1))
```

The above code calculates the Kauffman bracket. It does this by first applying the skein relation that defines the Kauffman bracket recursively. Then, if it cannot apply the skein relation, it counts the amount of circles that can be formed from the line segments. When looking at definition 17, it first applies rule 3 and when it exhausted the applications of rule 3, it applies rule 2. It more so combines rules 2 and 1. If the only shapes left are disjoint circles, we have the following that holds:

$$\langle \bigsqcup^{n} \bigcirc \rangle = (-A^2 - A^{-2})^{n-1}$$

```
# Returns the Normalized Kauffman bracket Polynomial
def kauffman_normalized(knot: list):
    writhe = 0
    len_knot = len(knot)
    # Calculate the Writhe of a knot
    for crossing in knot:
        if (crossing[1])%(2*len_knot) == (crossing[3]+1)%(2*len_knot):
            writhe += 1
        else:
            writhe -= 1
    kauff_part = kauffman(knot)
    return expand((-(A**3))**(-writhe) * kauff_part)
```

If we now add the Writhe to the calculation by using definition 18 we obtain the normalized Kauffman bracket from definition 19.

```
# Returns the Jones polynomial of a knot, using the normalized kauffman bracket
def jones(knot: list):
    kauff = kauffman_normalized(knot)
    return kauff.subs(A, (t)**(Rational(-1,4)))
```

Via a simple substitution, we obtain the Jones polynomial like in Theorem 5.

```
# knot in pd code --> knot in [sign_i,overpass_i,underpass_i] parts
def pd_to_sign_over_under(knot):
    sou = []
    len_knot = 2*len(knot)
    for crossing in knot:
        if (crossing[1]-1)%len_knot+1 == (crossing[3])%len_knot+1:
            sou.append([1,(crossing[3]-1)%len_knot , (crossing[0]-1)%len_knot])
        else:
            sou.append([-1, (crossing[1]-1)%len_knot,(crossing[0]-1)%len_knot])
    return sou
```

Next up we want to compute the Theta invariant, but its implementation required us to reformat our knot to sou code (definition 21). Instead of 1 being the smallest edge, 0 is now the smallest edge. This requires us to move all numbers one backwards. This Code structures the crossings to be of the form [sign_i,overpass_i,underpass_i]. The main reason is that we want all crossings to always point upwards when building the knot in the next step for the computation of their rotation numbers.

```
# knot in sou-code --> Rotation numbers
def rotation_numbers(knot: list):
    len_knot = len(knot)
    rot_list = [0 for _ in range(2 * len_knot + 1)]

    # Removeing the sign and changing the orden for negative crossings
    # This makes looping through the crossings to place them easier
    for crossing in knot:
        if crossing[0] == -1:
            crossing[1], crossing[2] = crossing[2], crossing[1]
        crossing.pop(0)

    # Startpoint
    front = [[0, "u"]]
    knot_done = False

    len_front = len(front)

    next_front = front
    while not knot_done:
        # Our stop condition is haveing a front of size 1 AND haveing used all knots
        if (len(front) == 1) and (len(knot) == 0):
            knot_done = True
            break

        draw_list = []
        skip_next = False
        front = next_front
        len_front = len(front)

        # create front and lists what to draw
        for i in range(len_front):
            # skip next number if we have placed cap/cross
            if skip_next:
                skip_next = False
                continue

            if i < len(front) - 1:
                # can we place a cap?
                if front[i][0] == front[i+1][0]:
                    draw_list.append("cap")
                    skip_next = True
                    #If cap is to the right, we update the rotation number
                    if front[i][1] == "d":
                        rot_list[front[i][0]] -= 1
                    next_front.pop(i)
                    next_front.pop(i)
                    continue

                # no placable knots = only caps and lines can be drawn
                if len(knot) == 0:
                    draw_list.append("line")
```

```
                    continue

                found = False
                for crossing in knot:
                    # find other half edge
                    if front[i][0] in crossing:
                        found = True
                        # can we place cross?
                        if front[i+1][0] in crossing:
                            next_front[i] = [(crossing[0])%(2*len_knot) + 1, "u"]
                            next_front[i+1] = [(crossing[1])%(2*len_knot) + 1, "u"]

                            # make a cross
                            draw_list.append("cross")

                            skip_next = True
                            knot.remove(crossing)
                            break
                        # if a crossing can be placed, but not a cross
                        else:
                            draw_list.append("line")
                            break
                # if no crossings can be placed
                if not found:
                    draw_list.append("line")
                    continue
            # At the end of the front, only a line can be drawn
            else:
                draw_list.append("line")

        # If we can only draw lines with the rules above, we need to place a cup + cross
        if len(set(draw_list)) == 1 and draw_list[0] == "line":
            done = False
            for i in range(len(front)):
                if done:
                    done = False
                    break
                for crossing in knot:
                    if front[i][0] in crossing:
                        # Do we place cup + cross on the right?
                        if front[i][0] == crossing[0]:
                            next_front[i] = [(crossing[1]%(2*len_knot)) + 1, "u"]
                            next_front.insert(i, [(crossing[0]%(2*len_knot)) + 1, "u"])
                            next_front.insert(i, [crossing[1], "d"])
                        else:
                            next_front[i] = [(crossing[0]%(2*len_knot)) + 1, "u"]
                            next_front.insert(i+1, [crossing[0], "d"])
                            next_front.insert(i+1, [(crossing[1]%(2*len_knot)) + 1, "u"])
                            rot_list[crossing[0]] += 1

                        knot.remove(crossing)

                        done = True
                        break
    return rot_list
```

Just like the two-one pass, we need to build the knot. While the general structure of building the knot is the same there are some notable differences. The start and end conditions are different. The two-one pass starts with three values and wants to end with one. The two-one pass can also end prematurely. The rotation numbers start and end with one value and always start with the same value. The function cannot end prematurely. Another difference is that the two-one pass places a fork if it cannot place anything else. The rotation numbers cannot do this as the fork does not have a clear pointing up direction. Therefore we place a cup and cross instead. Lastly, every time a cup/cap is placed and points to the right, we need to update the rotation numbers. All these differences makes it so we cannot simply write a function that works for both applications. Even though the structure is similar, we need to duplicate a lot of code here.

```
# Computes the Theta invariant at t_1 = pi, t_2 = e
def theta(knot: list):
    Cs = pd_to_sign_over_under(knot)
    phi = rotation_numbers(copy.deepcopy(Cs))
```

```
    n = len(Cs)
    size = 2 * n + 1

    # Initialize matrix A1
    A1 = eye(size)

    # Populate A1 matrix
    for c in Cs:
        sign, i, j = c
        A1[i, i+1] = -t1**sign
        A1[i, j+1] = t1**sign - 1
        A1[j, j+1] = -1

    # Determinants
    total_phi = sum(phi)
    total_signs = sum(c[0] for c in Cs)
    Delta1 = t1**((-total_phi - total_signs)/2) * A1.det()
    Delta2 = Delta1.subs(t1, t2)
    Delta3 = Delta1.subs(t1, t1 * t2)

    # Inverses
    G1 = A1.inv()
    G2 = G1.subs(t1, t2)
    G3 = G1.subs(t1, t1 * t2)

    # Functions
    def F11(s, i, j):
        return s * (
            Rational(1, 2) + t2**s * G1[i, i] * G2[j, i] +
            ((t1**s - 1) * t2**(2*s) * G1[j, i] * G2[j, i]) / (t2**s - 1) - G1[i, i] * G2[j, j] -
            (t1**s - 1) * t2**s * G1[j, i] * G2[j, j] / (t2**s - 1) -
            G3[i, i] - (t2**s - 1) * G2[j, i] * G3[i, i] + 2 * G2[j, j] * G3[i, i] +
            (t1**s * t2**s - 1) * G3[j, i] / (t2**s - 1) -
            t2**s * (t1**s * t2**s - 1) * G1[i, i] * G3[j, i] / (t2**s - 1) -
            (t1**s - 1) * (t2**s + 1) * (t1**s * t2**s - 1) * G1[j, i] * G3[j, i] / (t2**s - 1) +
            (t1**s * t2**s - 1) * G2[i, j] * G3[j, i] / (t2**s - 1) +
            (t1**s * t2**s - 1) * G2[j, i] * G3[j, i] +
            (t2**s - 2) * (t1**s * t2**s - 1) * G2[j, j] * G3[j, i] / (t2**s - 1) +
            G1[i, i] * G3[j, j] + (t1**s - 1) * t2**s * G1[j, i] * G3[j, j] / (t2**s - 1) -
            G2[i, i] * G3[j, j] -t2**s * G2[j, i] * G3[j, j]
        )

    def F12(s0, i0, j0, s1, i1, j1):
        numerator = (t1**s0 - 1) * (t1**s1 * t2**s1 - 1) * G1[j1, i0] * G3[j0, i1] * (
            t2**s0 * G2[i1, i0] - G2[i1, j0] - t2**s0 * G2[j1, i0] + G2[j1, j0]
        )
        return s1 * numerator / (t2**s1 - 1)

    def Gam1(ph, k):
        return ph * G3[k, k] - ph / 2

    # Theta computation
    theta = 0
    for c in Cs:
        theta += F11(c[0], c[1], c[2])
    for c1 in Cs:
        for c2 in Cs:
            theta += F12(c1[0], c1[1], c1[2], c2[0], c2[1], c2[2])
    for k in range(size):
        theta += Gam1(phi[k], k)

    # Final expression
    theta = theta * Delta1 * Delta2 * Delta3
    return theta.subs({t1: pi, t2: e}).evalf()
```

The above code is mostly a direct translation from [19]. There are two main differences. First, we input a knot in PD code and then compute the sou code and corresponding rotation numbers instead of having them as inputs. Secondly, we evaluate the Theta invariant at $t_1 = \pi$ and $t_2 = e$. The reason for this is that the Theta invariant returns a very big expression and calling something like expand() on this expression takes a very long time to resolve. Simplifying the expression is therefore not an option and keeping the expression non-simplified is also inconvenient. The

main drawback of evaluating the expression is that we will have to deal with floating point imprecision. This makes is so comparing two knots becomes harder. For two knots that are the same, but with a different representation, their Theta invariants will slightly off. We will do some testing to figure out a bound on the acceptable error between two similar knots.

```
# Find a bound for theta to be able to compare theta values
def find_theta_bound():
    smallest = 1000000
    max_diff = 0
    file = open(f"8.pass_reduced/{7}.txt", "r")
    for line in file:
        knot = ast.literal_eval(line)
        knot_list = all_representations(knot)
        theta_list = []
        # Find theta of all prepresentations
        for knot in knot_list:
            theta_list.append(theta(knot))
        # Find the biggest difference
        diff = max(theta_list) - min(theta_list)
        jones_poly = jones(knot)
        # Keep track of biggest difference and smallest absolute
        if abs(min(theta_list)) < smallest:
            if jones_poly.subs(t, 1/t) != jones_poly:
                smallest = abs(min(theta_list))
        if diff > max_diff:
            max_diff = diff
    print("max diff:", diff)
    print("Smallest absolut:", smallest)
```

We know for sure that different representations of the same knot are indeed the same knot and should produce the same Theta invariant. We will use this to find a suitable bound which we can use for comparing Theta values. We arbitrarily choose to check the biggest difference in Theta of all representations of knots of 8 crossings. We also checked what the smallest absolute Theta value is. Note that we exclude values for which the Jones polynomial equals the Jones polynomial with $1/t$ substituted as in this case Theta equals 0. The biggest difference came back as: $3.43425199389458e-07$ and the smallest number in absolute value is: 29.9376760185094. Given this data, a bound of $1e-2$ is small enough that it will not be bigger than any theta invariant and it is big enough to encompass all representations of a knot even when flypes and R3 are involved.

```
# Given knot and startpoint, rebuild the knot and apply flype transformation
def apply_flype(knot: list[list[int]], front: list, end_list, directions, sign):
    start_front = copy.deepcopy(front)
    crossings_to_flip = []
    len_knot = len(knot) + 1
    len_front = len(front)
    end_len = 2
    first_step = True
    next_front = front

    while len(front) != 0:
        draw_list = []
        skip_next = False
        front = next_front
        len_front = len(front)

        # Check conditions to stop looping
        if len_front == end_len:
            if first_step == False:
                end_front = front
                break
        first_step = False
        for number in front:
            if number in end_list:
                return False

        # Create front and lists what to draw
        for i in range(len_front):
            if skip_next:
                skip_next = False
```

```
            continue

        if i < len(front) - 1:
            # Can we apply a cap?
            if front[i] == front[i+1]:
                draw_list.append("cap")

                skip_next = True
                next_front.pop(i)
                next_front.pop(i)
                continue

            # If there are no crossings left to place, we can only place a cap or a line
            if len(knot) == 0:
                draw_list.append("line")
                continue


            found = False
            for crossing in knot:
                # find other half edge
                if front[i] in crossing:
                    found = True
                    # Can we place a cross?
                    if front[i+1] in crossing:
                        # place the cross
                        idx_i = crossing.index(front[i])
                        idx_i2 = crossing.index(front[i+1])
                        if idx_i +1 != idx_i2:
                            next_front[i] = crossing[(idx_i+1)%4]
                            next_front[i+1] = crossing[(idx_i2-1)%4]
                        else:
                            next_front[i] = crossing[(idx_i-1)%4]
                            next_front[i+1] = crossing[(idx_i2+1)%4]


                        if crossing[0] == next_front[i] or crossing[2] == next_front[i]:
                            draw_list.append("cross_under")
                        else:
                            draw_list.append("cross_over")

                        skip_next = True
                        crossings_to_flip.append(crossing)
                        knot.remove(crossing)
                        break
                    # if a crossing can be placed, but not a cross
                    else:
                        draw_list.append("line")
                        break
            # if no crossings can be placed
            if not found:
                draw_list.append("line")
                continue
        # At the end of the front, only a line can be drawn
        else:
            draw_list.append("line")

    # If we can only draw lines with the rules above, we need to place a fork
    if len(set(draw_list)) == 1 and draw_list[0] == "line":
        done = False
        for i in range(len(front)):
            if done:
                done = False
                break
            for crossing in knot:
                if front[i] in crossing:
                # find other half edge
                    idx = crossing.index(front[i])
                    next_front.pop(i)
                    next_front.insert(i, crossing[(idx - 1)%4])
                    next_front.insert(i, crossing[(idx - 2)%4])
                    next_front.insert(i, crossing[(idx - 3)%4])
                    crossings_to_flip.append(crossing)
```

```
                    knot.remove(crossing)
                    draw_list[i] = "fork"
                    done = True
                    break

    # Apply the flype transformation
    for crossing in crossings_to_flip:
        # Flip the order
        crossing[1], crossing[3] = crossing[3], crossing[1]
        # Ensure that the Flip produces a valid PD code
        if (crossing[1]%(2*len_knot)) + 1 ==  crossing[3]:
            crossing = crossing[1:] + crossing[:1]
        else:
            crossing = crossing[3:] + crossing[:3]

        # All crossings that are flipped need to be relabled aswell
        if directions == "rr":
            for idx, number in enumerate(crossing):
                crossing[idx] = (number - 2)%(2*len_knot) + 1
        elif directions == "ll":
            for idx, number in enumerate(crossing):
                crossing[idx] = (number)%(2*len_knot) + 1
        else:
            right_dir_idx = -1
            start_front_idx = -1
            if directions == "rl":
                start_front_idx = 1
            else:
                start_front_idx = 0

            for i in range(2*len_knot):
                if (start_front[start_front_idx] + i-1)%(2*len_knot) + 1 in end_front:
                    right_dir_idx = end_front.index((start_front[start_front_idx] + i-1)%(2*len_knot) +
                                                                                    1)
                    break
                # Skip over a specific case that is significantly harder to implement then the rest
                if (start_front[start_front_idx] + i-1)%(2*len_knot) + 1 == start_front[(start_front_idx
                                                                                + 1 )%2]:
                    return False

            if start_front[start_front_idx] < end_front[right_dir_idx]:
                for idx, number in enumerate(crossing):
                    if start_front[start_front_idx] <= number <= end_front[right_dir_idx]:
                        crossing[idx] = (number - 2)%(2*len_knot) + 1
                    else:
                        crossing[idx] = (number)%(2*len_knot) + 1
            else:
                for idx, number in enumerate(crossing):
                    if number >= start_front[start_front_idx] or end_front[right_dir_idx]>= number:
                    # if number <= start_front[start_front_idx] or end_front[right_dir_idx]<= number:
                        crossing[idx] = (number - 2)%(2*len_knot) + 1
                    else:
                        crossing[idx] = (number)%(2*len_knot) + 1

        knot.append(crossing)

    # Add the crossing on the other side of the flipped crossings
    if directions == "rr":
        if sign == "+":
            knot.append([(end_front[1] - 2)%(2*len_knot) + 1, end_front[0], end_front[1], (end_front[0]
                                                                - 2)%(2*len_knot) + 1])
        else:
            knot.append([(end_front[0] - 2)%(2*len_knot) + 1, (end_front[1] - 2)%(2*len_knot) + 1,
                                                               end_front[0], end_front[1]])
    elif directions == "ll":
        if sign == "+":
            knot.append([end_front[1], (end_front[0])%(2*len_knot) + 1, (end_front[1])%(2*len_knot) + 1,
                                                                end_front[0]])
        else:
            knot.append([end_front[0], end_front[1], (end_front[0])%(2*len_knot) + 1, (end_front[1])%(2*
                                                               len_knot) + 1])
    elif directions == "rl" or directions == "lr":
        if sign == "+":
```

```
            if right_dir_idx == 0:
                knot.append([(end_front[0] - 2)%(2*len_knot) + 1,(end_front[1])%(2*len_knot) + 1,
                                                                end_front[0], end_front[1]])
            else:
                knot.append([end_front[0], end_front[1], (end_front[0])%(2*len_knot) + 1, (end_front[1]
                                                                - 2)%(2*len_knot) + 1])
        else:
            if right_dir_idx == 0:
                knot.append([end_front[1], (end_front[0]-2)%(2*len_knot) + 1, (end_front[1])%(2*len_knot
                                                                ) + 1, end_front[0]])
            else:
                knot.append([(end_front[1]-2)%(2*len_knot) + 1, end_front[0], end_front[1], (end_front[0
                                                                ])%(2*len_knot) + 1])

    return canonical_representation(knot)
```

For the next operation, we implement the flype 6. We start with a knot, a starting point and end condition. In essence, it is the same process we went through to implement the two-one pass. This means we also have to build a segment of the knot again. For this operation we need to keep track of what crossings we use in the flype as they will have to be turned upside down and will need to be relabeled.

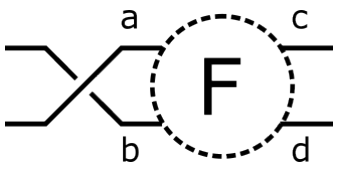


Figure 17: Example of where a flype can be applied, with some edges labeled

The of the edges a and b can be oriented in either direction and c and d will have the same orientation in some order. For example, a, d can be pointing to the right while b, c are pointing to the left. The crossing on the left disappears and we will need to construct a new crossing on the right. This mean we need to relabel some crossings. We will only need to relabel crossings inside the flyped segment if we leave out one specific case. If edge a is connected to edge b inside the flyped region we would need to also relabel crossings that are not part of the flype. For this reason we choose to leave out this case.

```
# For a knot, go through all crossings and attempt to apply a flype everywhere
def flype_list(knot:list[list[int]]):
    result = [knot]
    len_knot = len(knot)
    for crossing in knot:
        knot_1 = copy.deepcopy(knot)
        knot_1.remove(crossing)
        # Each crossing can be build out in four separate ways
        if (crossing[1] + 1)%(2*len_knot) == crossing[3]%(2*len_knot):
            flype_1 = apply_flype(copy.deepcopy(knot_1), [crossing[2], crossing[3]], [crossing[0],
                                                                crossing[1]], "rr", "-")
            flype_2 = apply_flype(copy.deepcopy(knot_1), [crossing[1], crossing[2]], [crossing[0],
                                                                crossing[3]], "rl", "-")
            flype_3 = apply_flype(copy.deepcopy(knot_1), [crossing[0], crossing[1]], [crossing[2],
                                                                crossing[3]], "ll", "-")
            flype_4 = apply_flype(copy.deepcopy(knot_1), [crossing[3], crossing[0]], [crossing[1],
                                                                crossing[2]], "lr", "-")
        else:
            flype_1 = apply_flype(copy.deepcopy(knot_1), [crossing[1], crossing[2]], [crossing[0],
                                                                crossing[3]], "rr", "+")
            flype_2 = apply_flype(copy.deepcopy(knot_1), [crossing[0], crossing[1]], [crossing[2],
                                                                crossing[3]], "rl", "+")
            flype_3 = apply_flype(copy.deepcopy(knot_1), [crossing[3], crossing[0]], [crossing[1],
                                                                crossing[2]], "ll", "+")
            flype_4 = apply_flype(copy.deepcopy(knot_1), [crossing[2], crossing[3]], [crossing[0],
                                                                crossing[1]], "lr", "+")

        # Only add the result if it isnt there already
        if flype_1 not in result and flype_1 != False:
            result.append(flype_1)
        if flype_2 not in result and flype_2 != False:
```

```
            result.append(flype_2)
        if flype_3 not in result and flype_3 != False:
            result.append(flype_3)
        if flype_4 not in result and flype_4 != False:
            result.append(flype_4)

    return result
```

We want to apply the previous function to all directions of all crossings of a knot to obtain a "complete" list of all flyping possibilities.

```
# List of knots --> reduced list of knots such that the first element is not flype related to the others
def flype_reduce(knot_list: list[list[list[int]]]):
    unique_list = [knot_list[0]]

    max_depth = 5
    depth = 0
    all_flypes = [knot_list[0]]
    have_been_flyped = []
    while depth < max_depth:
        # Fill list of all flypes (of flypes) of the first knot
        for knot in all_flypes:
            # We dont want to do work twice
            if knot in have_been_flyped:
                continue
            have_been_flyped.append(knot)

            flyped = flype_list(knot)
            for flype_knot in flyped:
                if flype_knot in all_flypes:
                    continue
                all_flypes.append(flype_knot)
        # Delete knots in our original list if they are flype related to the first knot
        for knot_list_knot in knot_list:
            if knot_list_knot in all_flypes:
                knot_list.remove(knot_list_knot)
        # If all other knots from our original list are flype related to the first, we are done
        if len(knot_list) == 0:
            break
        depth += 1
    # Returns a list containing the first knot and all knots that are not flype related to it
    unique_list = unique_list + knot_list
    return unique_list
```

For the above function, we will input a list of knots such that they have the same Jones polynomial. The way we do this will be clear from the next function. Then we continuously apply flypes until there is either one knot left or we have searched deep enough. The second stop condition prevents us from searching forever.

```
# 8.pass_reduced --> 9.jones_dict
# Creates a dict of the form {jones: [knots]}
# If not amphichiral, it writes either the knot or mirror image, not both
# Flype reduces the list of knots
def jones_dict_store():
    for i in range(3,11):
        file = open(f"8.pass_reduced/{i}.txt", "r")
        # create a dict:  {jones: [knots]}
        knot_dict = {}
        for line in file:
            knot = ast.literal_eval(line)
            knot_jones = jones(knot)
            if knot_jones not in knot_dict:
                knot_dict[knot_jones] = [knot]
            else:
                knot_dict[knot_jones] = [knot] + knot_dict.get(knot_jones)

        file.close()

        knot_dict_2 = {}
        write_value = 0
        for key, value in knot_dict.items():
            # If mirror image is already in dict, skip it
```

```
            if key.subs(t, 1/t) in knot_dict_2:
                continue

            # Flype reduce
            if len(value) > 1:
                write_value = flype_reduce(value)
            else:
                write_value = value

            knot_dict_2[key] = write_value

        new_file = open(f"9.jones_dict/{i}", "ab")
        # "pickle" the dict into a binary file
        dump(knot_dict_2, new_file)
        new_file.close()
```

It is time for a different method of storing our knots. For this function, we take all knots (of a specific amount of crossings) and put them in a dictionary. This will group knots based on their Jones polynomial. After doing this, we will also remove mirror images of knots. We do this by looking if the Jones polynomial evaluated at $1/t$ is already in the dictionary or not. Furthermore, we will also attempt to apply the flype operation to reduce the list of knots that have the same Jones polynomial. Storing this dictionary in a file will also work differently. Instead of storing the knots line for line as plain text in a text file, we will use the pickle package to store the dictionary in a binary file.

```
# Reads binary file and returns the dict stored there
def jones_dict_load(i)-> dict:
    dbfile = open(f"9.jones_dict/{i}", 'rb')
    db = load(dbfile)
    dbfile.close()
    return db
```

Since we are now storing the knots in a dictionary in a binary file, we need a function that can retrieve the dictionary from the binary file. This also uses the pickle package.

```
# 9.jones_dict --> 10.theta_reduced
# Split values over multiple keys if they dont have the same Theta value
def split_by_theta():
    for i in range(3, 11):
        jones_dict = jones_dict_load(i)
        theta_reduced_dict = {}
        for key, value in jones_dict.items():
            # For 1 value, we dont need to compute theta
            if len(value) == 1:
                theta_reduced_dict[key] = value
                continue

            # Compute list of all theta values
            theta_list = []
            for knot in value:
                theta_list.append([theta(knot), knot])
            theta_list = sorted(theta_list)

            # Construct goupes for different Theta's
            groups = []
            current_group = [theta_list[0]]
            for val in theta_list[1:]:
                if abs(val[0] - current_group[-1][0]) < theta_bound:
                    current_group.append(val)
                else:
                    groups.append(current_group)
                    current_group = [val]

            groups.append(current_group)

            # if 1 group, we dont need to split it
            if len(groups) == 1:
                theta_reduced_dict[key] = value
                continue

            # Check for mirror image
```

```
            seen_theta = [groups[0][0][0]]
            for group in groups[1:]:
                for seen_theta_value in seen_theta:
                    if abs(group[0][0] + seen_theta_value) < theta_bound:
                        groups.remove(group)
                        break

            # Remove Theta to have all values in dict have the same structure
            new_groups = []
            for group in groups:
                new_group = []
                for knot in group:
                    new_group.append(knot[1])
                new_groups.append(new_group)

            # Create multiple entries from the groups
            j = 1
            for group in new_groups:
                theta_reduced_dict[(key, j)] = group
                j+=1

        new_file = open(f"10.theta_reduced/{i}", "ab")
        # "pickle" the dict into a binary file
        dump(theta_reduced_dict, new_file)
        new_file.close()
```

It is possible that two different knots have the same Jones polynomial and it is possible for a knot to be not amphichiral but still have that same Jones polynomial for the knot and its mirror image. Both of these issues can be reduced significantly by looking at the Theta invariant. For each list of knot of the same Jones polynomial, we compute the Theta invariant and sort the knots based on this into new groups. If the second issue occurs, we remove the group. We can figure out when this happens because if the Theta invariant equals a, then the mirror image equals $-a$.

```
# Reads binary file and returns the dict stored there
def theta_reduced_dict_load(i)-> dict:
    dbfile = open(f"10.theta_reduced/{i}", 'rb')
    db = load(dbfile)
    dbfile.close()
    return db
```

We again need a function to retrieve the dictionary from the binary file.

```
# Continously apply R3 and flypes
# Returns False if one of the knots is reducible
# Otherwise Returns a list such that te first element can not be found to be the same at the rest
def mixed_reduce(knot_list: list):
    unique_list = [knot_list[0]]

    max_depth = 10
    depth = 0
    all_applied = [knot_list[0]]
    have_been_flyped_and_r3 = []
    while depth < max_depth:
        for knot in all_applied:
            # Check if we have covered this knot already
            if knot in have_been_flyped_and_r3:
                continue
            have_been_flyped_and_r3.append(copy.deepcopy(knot))

            # Apply all R3s and flypes
            flyped = flype_list(copy.deepcopy(knot))
            r_3 = apply_R3(copy.deepcopy(knot))
            flyped_and_r3 = flyped + r_3

            # Check if reducable in crossing amount
            for f_r_knot in flyped_and_r3:
                if contains_trivial_R2(f_r_knot) or contains_R1(f_r_knot) or check_two_one_pass(f_r_knot
                                                                                                ):
                    return False

            # Dont append duplicates
            for knot_2 in flyped_and_r3:
```

```
                if knot_2 in all_applied:
                    continue
                all_applied.append(knot_2)

        for knot_list_knot in knot_list:
            if knot_list_knot in all_applied:
                knot_list.remove(knot_list_knot)

        # If we have reduced to 1 element (the one in unique_list)
        if len(knot_list) == 0:
            break

        depth += 1
    unique_list = unique_list + knot_list
    return unique_list
```

For the next reduction step, we will use all our methods at the same time. The main benefit is that we are mixing the applications of flyping and the third Reidemeister move. This allows us to more broadly search permutations of a knot. It is very possible that there are representations of a knot that can only be obtained from a combination of flypes and R3 and not just from one of them. A big and unfortunate assumption we are making here is that we are treating each list as the same knot. We are only applying transformations to the first knot in the list are removing the entire list if that single knot is reducible. We think that this is a reasonable assumption to make given the crossing numbers. All knots in a list have the same Jones polynomial and the same Theta invariant. Given the strength of these invariants and the relatively low crossing number, this is acceptable.

```
# 10.theta_reduced --> 11.mix_reduce
# applies the mixed_reduce algorithm to all sets of knot in the theta_reduced dict
def apply_mixed_reduce():
    for i in range(10, 11):
        knot_dict = theta_reduced_dict_load(i)
        new_dict = {}
        for key, value in knot_dict.items():
            new_value = mixed_reduce(copy.deepcopy(value))
            # if reducible, skip it
            if new_value == False:
                continue

            new_dict[key] = new_value

        new_file = open(f"11.mix_reduce/{i}", "ab")
        # "pickle" the dict into a binary file
        dump(new_dict, new_file)
        new_file.close()
```

We want to apply the above method on all values of the dictionary. We then want to store the result into another dictionary and use pickle to store it into a binary file.

```
# Reads binary file and returns the dict stored there
def mix_reduced_dict_load(i)-> dict:
    dbfile = open(f"11.mix_reduce/{i}", 'rb')
    db = load(dbfile)
    dbfile.close()
    return db
```

We again need a function to retrieve the dictionary from the binary file.

Besides sorting through all knots, we also want a way to visualize them and draw them to the screen. For this, we will use the PyGame package. The resulting figures are used in chapter 8 to show the knots. To build a knot, we need 5 different puzzle pieces: A starting piece, a line, a cap, a cross and a fork.

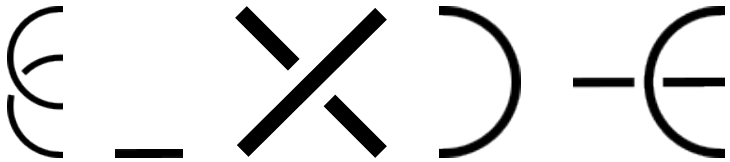


Figure 18: From left to right we have the starting piece, the line, the cross, the cap and the fork.

In order to draw these pieces to the screen, we use the following functions.

```
# Draw a semi-circle (with gap)
def draw_semicircle(p1, p2, direction, gap = "no"):
    # unpack Coordinates
    x1, y1 = p1
    _, y2 = p2
    # Find midpoint and radius
    mid_y = (y1 + y2) / 2
    radius = abs(y2 - y1) / 2
    rect = pygame.Rect(x1 - radius, mid_y - radius, radius*2, radius*2)
    start_angle = radians(90)
    end_angle = radians(270)
    # Switch the start and endpoint if we want the semicircle on the right side
    if direction == "right":
        start_angle, end_angle = end_angle, start_angle
    # Place a gap given an angle if needed
    if gap == "no":
        pygame.draw.arc(screen, "black", rect, start_angle, end_angle, 4)
    else:
        gap_size = radians(8)
        pygame.draw.arc(screen, "black", rect, start_angle, gap - gap_size, 4)
        pygame.draw.arc(screen, "black", rect, gap + gap_size, end_angle, 4)
```

Given two points (which lie on top of each other), this function draws a semi-circle to the screen. A gap can be placed in the semi-circle in order to use it as a starting piece.

```
# Draw an ellipse (with gap)
def draw_ellips(p1, p2, direction, gap = "no"):
    # unpack Coordinates
    x1, y1 = p1
    _, y2 = p2
    # Find midpoint and radius
    mid_y = (y1 + y2) / 2
    radius = abs(y2 - y1) / 2
    rect = pygame.Rect(x1 - (radius/2), mid_y - radius, radius, radius*2)
    start_angle = radians(90)
    end_angle = radians(270)
    # Switch the start and endpoint if we want the semicircle on the right side
    if direction == "right":
        start_angle, end_angle = end_angle, start_angle
    # Place a gap given an angle if needed
    if gap == "no":
        pygame.draw.arc(screen, "black", rect, start_angle, end_angle, 4)
    else:
        gap_size = radians(8)
        pygame.draw.arc(screen, "black", rect, start_angle, gap - gap_size, 4)
        pygame.draw.arc(screen, "black", rect, gap + gap_size, end_angle, 4)
```

Similarly, we can draw an ellipse as well, this is used to draw the fork.

```
# Draws the starting piece
def draw_0_4(p1, p2, p3, p4, sign):
    draw_semicircle(p1, p3, "left", gap = radians(150))
    draw_semicircle(p2, p4, "left")
```

The code above draws the starting piece from four coordinates (which lie on top of each other) using semicircles. Note that we fix the location of the gap, this is justified as we can rotate the first crossing until it appears correct.

```
# Draws a cross
def draw_cross(p1, p2, p3, p4):
    middle = tuple(numpy.divide(numpy.add(p1, p3),(2,2)))
    gapsize = tuple(numpy.divide(numpy.subtract(p3, p1), (10,10)))
    pygame.draw.line(screen, "black", p1, tuple(numpy.subtract(middle, gapsize)), 4)
    pygame.draw.line(screen, "black", tuple(numpy.add(middle, gapsize)), p3, 4)
    pygame.draw.line(screen, "black", p2, p4, 4)
```

Using four points, we can draw a cross. We do not have to specify the sign of the crossing as we can rotate the inputs when calling the function to get the other sign.

```
# Draws a fork
def draw_fork(p1,p2,p3,p4, gap):
    # The gap can eiher be part of the line or the ellipse
    if gap == "line":
        middle = tuple(numpy.divide(numpy.add(p1, p3),(2,2)))
        gapsize = tuple(numpy.divide(numpy.subtract(p3, p1), (10,10)))
        pygame.draw.line(screen, "black", p1, tuple(numpy.subtract(middle, gapsize)), 4)
        pygame.draw.line(screen, "black", tuple(numpy.add(middle, gapsize)), p3, 4)
        draw_ellips(p2,p4, "left")
    else:
        pygame.draw.line(screen, "black", p1, p3, 4)
        draw_ellips(p2,p4, "left", radians(180))
```

For the fork, we will need to specify where the gap is located. We also only place a fork in the orientation that can be found in figure 18.

```
# Front --> screen Coordinates
def update_point_list(len_front, front_number):
    startheight = 0
    if len_front%2: # odd
        startheight = HEIGHT/2 + floor(len_front/2)*spacing
    else: # even
        startheight = HEIGHT/2 + (len_front-1)*(spacing/2)
    points = [(margin_left+ spacing*front_number, startheight - i*spacing) for i in range(len_front)]
    return points
```

Our next function takes points from the front and converts these to screen coordinates. These coordinates are used in our previous functions to draw all the pieces at the correct location. What exactly the "front" is will be explained in the next function.

```
def draw_knot(knot):
    current_crossing = knot[0]
    front = current_crossing
    len_front = len(front)
    knot.remove(current_crossing)

    front_number = 1
    points = update_point_list(len(front), front_number)

    draw_0_4(points[0], points[1], points[2], points[3], "+")
    if show_numbers:
        for idx, point in enumerate(points):
                text_surface = font.render(str(front[idx]), True, "black")
                screen.blit(text_surface, (point[0] - 10, point[1] - 30))

    next_front = front
    next_points = points
    while len(front) != 0:
        draw_list = []
        skip_next = False
        front_number += 1
        points = next_points
        front = next_front
        len_front = len(front)
        global max_front_size, max_front_number
        if len_front > max_front_size:
            max_front_size = len_front
```

```
        if front_number > max_front_number:
            max_front_number = front_number

        # create front and lists what to draw
        for i in range(len_front):
            if skip_next:
                skip_next = False
                continue

            if i < len(front) - 1:
                # can we place a cap?
                if front[i] == front[i+1]:

                    draw_list.append("cap")
                    skip_next = True
                    next_front.pop(i)
                    next_front.pop(i)
                    continue

                # If there are no crossings left to place, we can only place a cap or a line
                if len(knot) == 0:
                    draw_list.append("line")
                    continue

                found = False
                for crossing in knot:
                    # find other half edge
                    if front[i] in crossing:
                        # place the cross
                        found = True
                        if front[i+1] in crossing:
                            idx_i = crossing.index(front[i])
                            idx_i2 = crossing.index(front[i+1])
                            if idx_i +1 != idx_i2:
                                next_front[i] = crossing[(idx_i+1)%4]
                                next_front[i+1] = crossing[(idx_i2-1)%4]
                            else:
                                next_front[i] = crossing[(idx_i-1)%4]
                                next_front[i+1] = crossing[(idx_i2+1)%4]

                            if crossing[0] == next_front[i] or crossing[2] == next_front[i]:
                                draw_list.append("cross_under")
                            else:
                                draw_list.append("cross_over")

                            skip_next = True
                            knot.remove(crossing)
                            break
                        # if a crossing can be placed, but not a cross
                        else:
                            draw_list.append("line")
                            break
                # if no crossings can be placed
                if not found:
                    draw_list.append("line")
                    continue
            # At the end of the front, only a line can be drawn
            else:
                draw_list.append("line")

        # If we can only draw lines with the rules above, we need to place a fork
        if len(set(draw_list)) == 1 and draw_list[0] == "line":
            done = False
            for i in range(len(front)):
                if done:
                    done = False
                    break
                for crossing in knot:
                    if front[i] in crossing:
                    # find other half edge
                        idx = crossing.index(front[i])
                        next_front.pop(i)
                        next_front.insert(i, crossing[(idx - 1)%4])
```

```
                    next_front.insert(i, crossing[(idx - 2)%4])
                    next_front.insert(i, crossing[(idx - 3)%4])
                    knot.remove(crossing)
                    if crossing[0] == next_front[i] or crossing[2] == next_front[i]:
                        draw_list[i] = "fork_semi"
                    else:
                        draw_list[i] = "fork_line"
                    done = True
                    break

        # draw front labels
        next_points = update_point_list(len(next_front), front_number)
        if show_numbers:
            for idx, point in enumerate(next_points):
                text_surface = font.render(str(next_front[idx]), True, "black")
                screen.blit(text_surface, (point[0] - 10, point[1] - 30))

        # draw from list
        i = 0
        front_offset = 0
        for operation in draw_list:
            if operation == "line":
                pygame.draw.line(screen,"black", points[i], next_points[i+front_offset], 4)
                i+=1
            elif operation == "cross_over":
                draw_cross(points[i], points[i+1], next_points[i+1], next_points[i + front_offset])
                i+=2
            elif operation == "cross_under":
                draw_cross(next_points[i + front_offset], points[i], points[i+1], next_points[i+1])
                i+=2
            elif operation == "cap":
                draw_semicircle(points[i], points[i+1], "right")
                front_offset -= 2
                i+=2
            elif operation == "fork_line":
                draw_fork(points[i], next_points[i],next_points[i+1], next_points[i+2], "line")
                front_offset += 2
                i+= 1
            elif operation == "fork_semi":
                draw_fork(points[i], next_points[i],next_points[i+1], next_points[i+2], "semi")
                front_offset += 2
                i+= 1
```

Given a knot in PD notation, the above function draws this knot to the screen. A knot is always build from left to right. Every knot starts with the starting piece, for this piece the first crossing is used. This also sets up a front of size four. The front is a set of numbers representing which half-edges still need a connection. This front should be read from the bottom to the top. As an example, we take the trefoil which has PD-code $[[1,4,2,5],[3,6,4,1],[5,2,6,3]]$. For the starting piece we take the first crossing.

Figure 19: Starting piece for the trefoil

The front at this point is $[1,4,2,5]$. Now it is time to start placing puzzle pieces. We go through the front from left to right (or bottom to top in the figure) and check if we can place a puzzle piece. We will need to keep track of what pieces we are placing. If two consecutive numbers are the same we can place a cap. This will reduce the size of the next front by two. 1 and 4 are not the same number, therefore we go to the next check. Next we check if we can place a cross. 1 and 4 are both part of the second crossing which is $[3,6,4,1]$, therefore we will remember that we first need to place a crossing. Before we start drawing we check if we can place more puzzle pieces. We do not have to check for 4 as it is already part of a puzzle piece, so we move on to 2. 2 and 5 are not the same, therefore we can not place a cap. 2 and 5 are part of the same crossing, being $[5,2,6,3]]$ therefore we can place another crossing. At this point we have worked through

the whole front and we move on to the drawing step.

We need to place two crossings in a row and update the new front accordingly. This gives the following result.

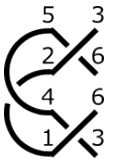
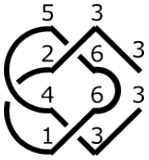


Figure 20: Building up the trefoil

Our front now looks like this: $[3, 6, 6, 3]$. Now we need to go through the whole procedure again until we are done drawing. Since we have no more crossings to place, we can only place lines and caps. Since 6 and 6 are the only two consecutive numbers we know that we will need to draw a line, a cap and another line in that order. Since we draw a cap, the total size of the front goes down by two. As we want to maintain the same distance between points the whole time, the two lines we draw are going to be diagonal.



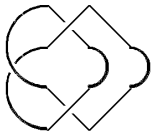
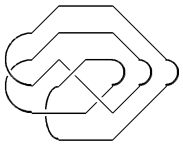
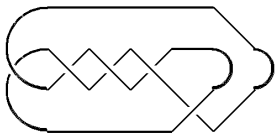
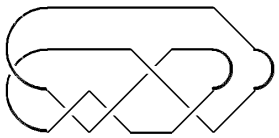
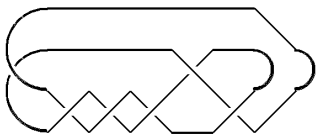
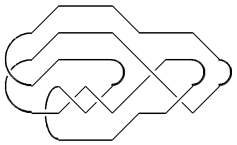
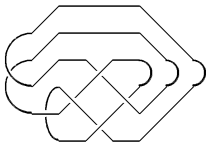
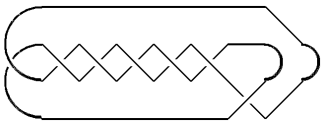
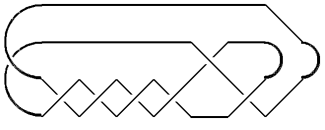
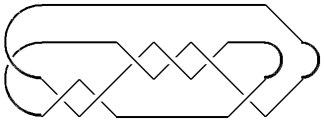
Now our front is $[3, 3]$ and there is one more step to complete the drawing, which is to place a cap.

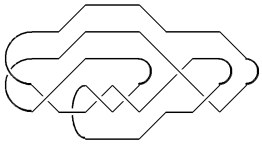
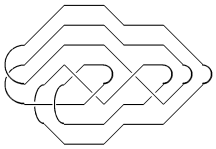
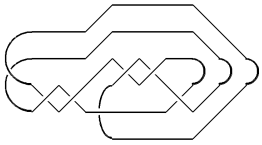
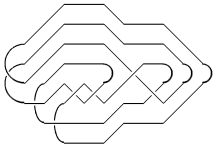
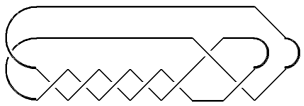
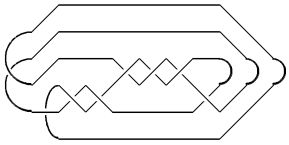
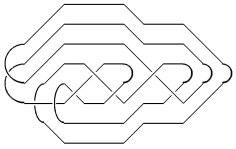
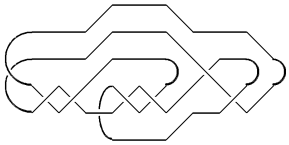
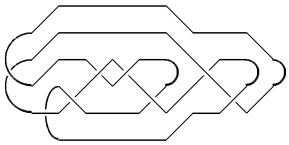
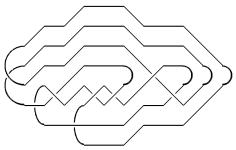
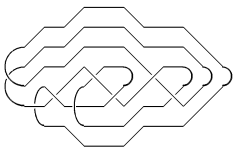
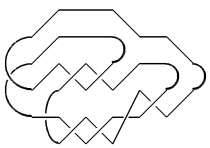
As the front is now of size 0 and there are no more crossings left to place, we know that we have successfully drawn a knot.

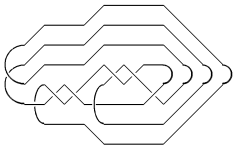
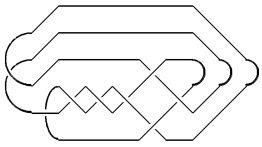
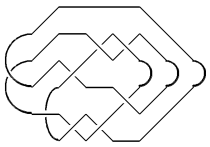
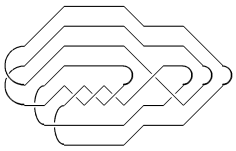
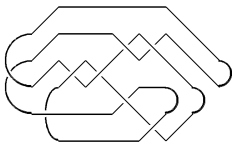
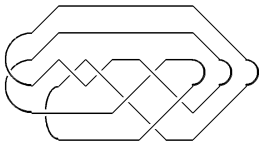
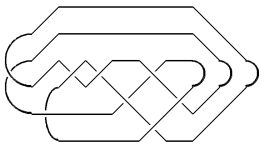
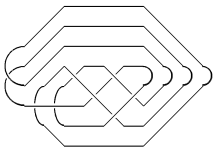
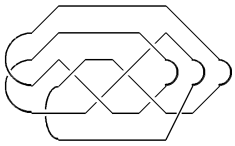
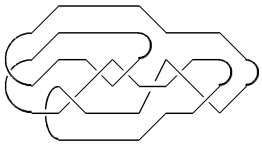
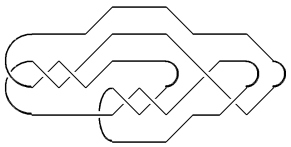
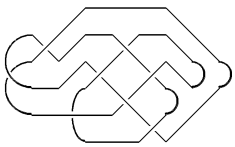
There is one more case to consider. It is entirely possible that looking through the front, we can not place any caps or crosses. In this case we do not want to draw only lines. If we were to do that we would not make any progress as in the next step we would be in the same situation. In this case we place a singular fork at the earliest location that allows it.

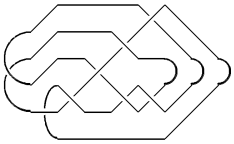
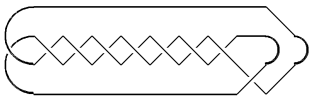
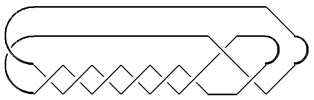
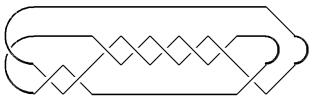
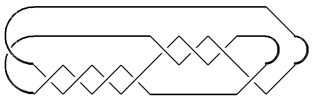
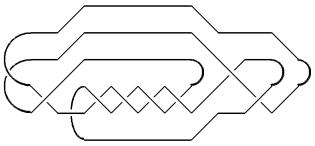
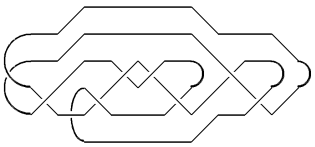
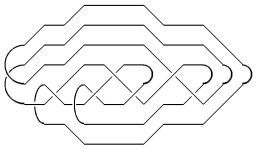
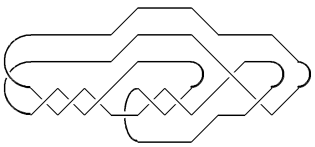
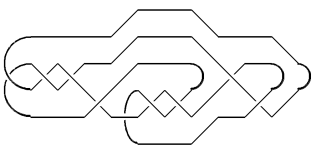
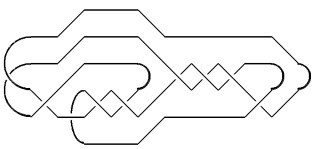
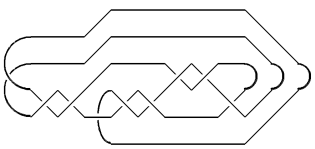
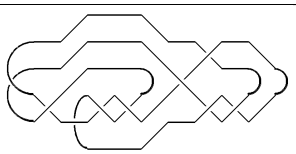
8 THE KNOT TABLE

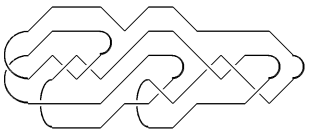
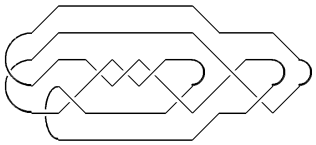
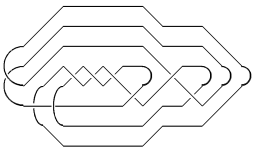
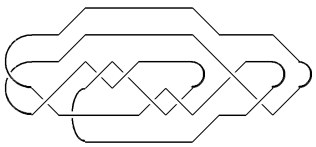
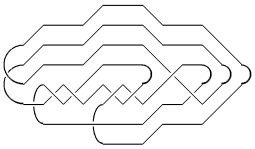
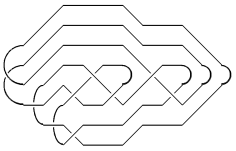
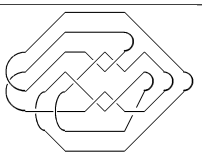
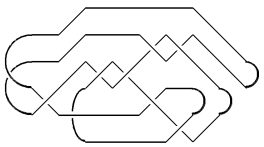
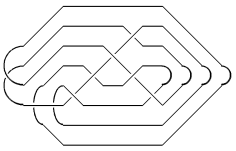
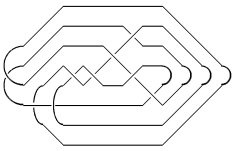
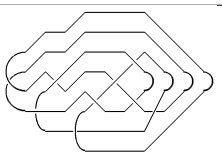
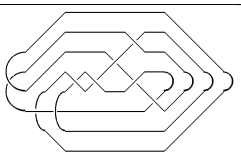
Our final result is the following knot table. Here, "Name" might not be the same as the Alexander-Briggs notation of the knot. Furthermore, Theta(π, e) will be cut off at nine digits of precision. This is because of floating point imprecision at this scale.

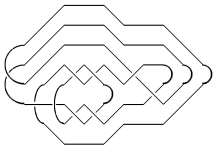
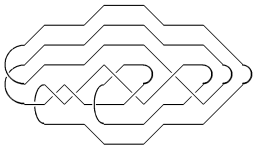
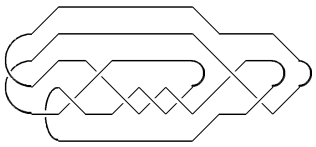
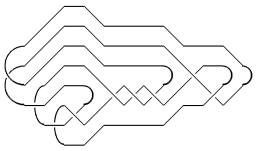
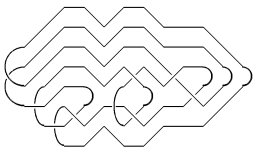
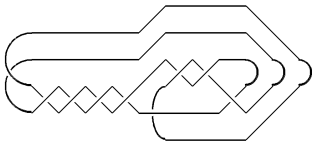
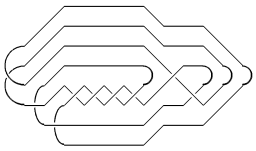
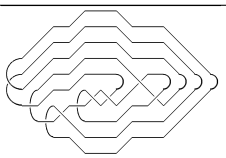
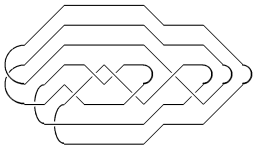
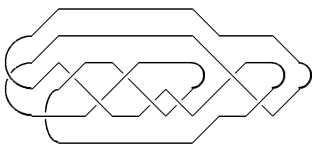
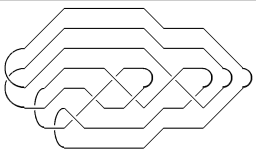
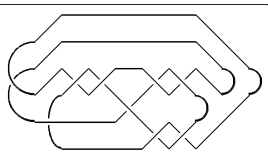
Name	Figure	amphichiral	Jones	Theta(π, e)
3_1		no	$\frac{1}{t} + \frac{1}{t^3} - \frac{1}{t^4}$	-38.292983368
4_1		yes	$t^2 - t + 1 - \frac{1}{t} + \frac{1}{t^2}$	0
5_1		no	$-t^7 + t^6 - t^5 + t^4 + t^2$	5292.97026695
5_2		no	$-t^6 + t^5 - t^4 + 2t^3 - t^2 + t$	215.28181040
6_1		no	$t^4 - t^3 + t^2 - 2t + 2 - \frac{1}{t} + \frac{1}{t^2}$	-1.22415649
6_2		no	$t^5 - 2t^4 + 2t^3 - 2t^2 + 2t - 1 + \frac{1}{t}$	-201.98343175
6_3		yes	$-t^3 + 2t^2 - 2t + 3 - \frac{2}{t} + \frac{2}{t^2} - \frac{1}{t^3}$	0
7_1		no	$-t^{10} + t^9 - t^8 + t^7 - t^6 + t^5 + t^3$	575615.28973434
7_2		no	$-t^8 + t^7 - t^6 + 2t^5 - 2t^4 + 2t^3 - t^2 + t$	694.70337479
7_3		no	$-t^9 + t^8 - 2t^7 + 3t^6 - 2t^5 + 2t^4 - t^3 + t^2$	29631.54625102

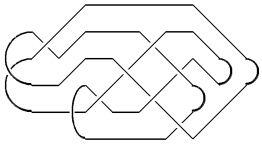
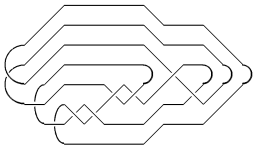
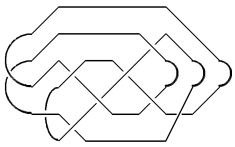
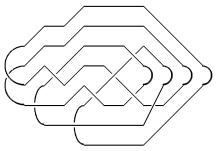
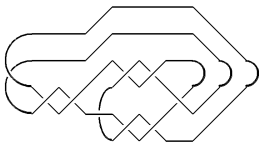
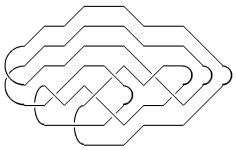
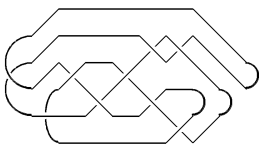
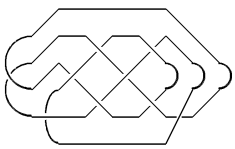
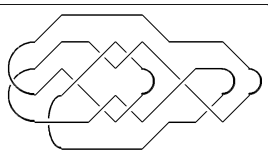
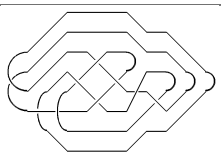
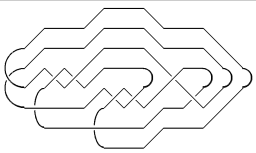
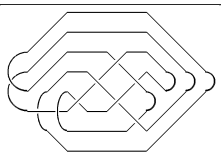
7_4		no	$-t^9+2t^8-3t^7+3t^6-3t^5+3t^4-t^3+t^2$	22335.53751413
7_5		no	$-t^6+2t^5-3t^4+4t^3-3t^2+3t-2+\frac{1}{t}$	-111.09637014
7_6		no	$-t^8+t^7-2t^6+3t^5-2t^4+3t^3-2t^2+t$	1362.90288293
7_7		no	$t^4-2t^3+3t^2-4t+4-\frac{3}{t}+\frac{3}{t^2}-\frac{1}{t^3}$	35.84467038
8_1		no	$t^6-t^5+t^4-2t^3+2t^2-2t+2-\frac{1}{t}+\frac{1}{t^2}$	-29.93767602
8_2		yes	$t^4-t^3+2t^2-3t+3-\frac{3}{t}+\frac{2}{t^2}-\frac{1}{t^3}+\frac{1}{t^4}$	0
8_3		yes	$t^4-2t^3+4t^2-5t+5-\frac{5}{t}+\frac{4}{t^2}-\frac{2}{t^3}+\frac{1}{t^4}$	0
8_4		no	$t^7-2t^6+3t^5-4t^4+4t^3-4t^2+3t-1+\frac{1}{t}$	-2417.84609239
8_5		no	$t^7-2t^6+3t^5-5t^4+5t^3-4t^2+4t-2+\frac{1}{t}$	-1403.13677418
8_6		no	$t-2+\frac{4}{t}-\frac{5}{t^2}+\frac{6}{t^3}-\frac{5}{t^4}+\frac{4}{t^5}-\frac{3}{t^6}+\frac{1}{t^7}$	690.83311215
8_7		no	$-t^3+2t^2-3t+5-\frac{4}{t}+\frac{4}{t^2}-\frac{3}{t^3}+\frac{2}{t^4}-\frac{1}{t^5}$	-878.67075625
8_8		no	$1-\frac{1}{t}+\frac{2}{t^2}-\frac{2}{t^3}+\frac{3}{t^4}-\frac{3}{t^5}+\frac{2}{t^6}-\frac{2}{t^7}+\frac{1}{t^8}$	30248.57284443

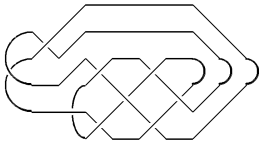
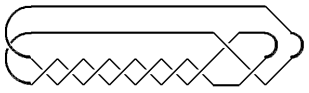
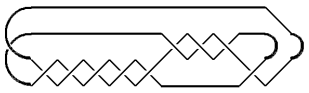
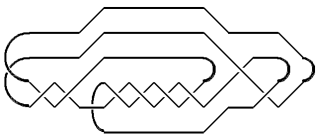
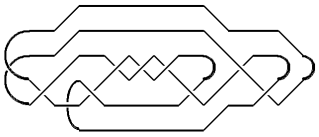
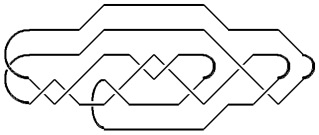
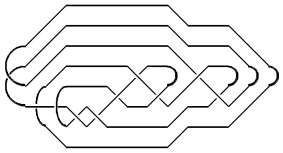
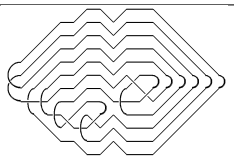
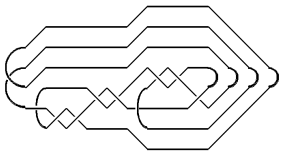
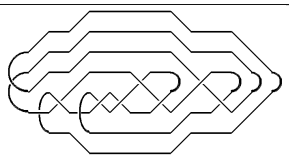
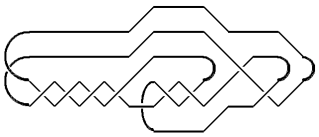
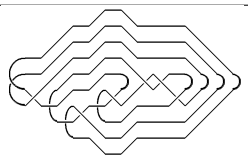
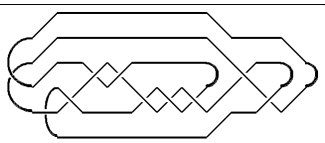
8_9		no	$t^5 - 2t^4 + 3t^3 - 3t^2 + 3t - 3 + \frac{2}{t} - \frac{1}{t^2} + \frac{1}{t^3}$	-3643.00640449
8_{10}		no	$-t^6 + 2t^5 - 3t^4 + 4t^3 - 4t^2 + 4t - 2 + \frac{2}{t} - \frac{1}{t^2}$	49265.59363454
8_{11}		yes	$t^4 - 2t^3 + 3t^2 - 4t + 5 - \frac{4}{t} + \frac{3}{t^2} - \frac{2}{t^3} + \frac{1}{t^4}$	0
8_{12}		no	$-t^5 + 2t^4 - 3t^3 + 5t^2 - 5t + 5 - \frac{4}{t} + \frac{3}{t^2} - \frac{1}{t^3}$	915.73958312
8_{13}		no	$t^{10} - 3t^9 + 4t^8 - 6t^7 + 6t^6 - 5t^5 + 5t^4 - 2t^3 + t^2$	43467.86622692
8_{14}		no	$t^7 - 2t^6 + 2t^5 - 3t^4 + 3t^3 - 2t^2 + 2t$	-240.03036400
8_{15}		no	$-t^5 + t^4 - t^3 + 2t^2 - t + 2 - \frac{1}{t}$	155.87748256
8_{16}		no	$-t^8 + t^5 + t^3$	495542.92559803
8_{17}		yes	$t^4 - 4t^3 + 6t^2 - 7t + 9 - \frac{7}{t} + \frac{6}{t^2} - \frac{4}{t^3} + \frac{1}{t^4}$	0
8_{18}		no	$-t^6 + 2t^5 - 4t^4 + 5t^3 - 4t^2 + 5t - 3 + \frac{2}{t} - \frac{1}{t^2}$	60988.26529641
8_{19}		no	$t^8 - 2t^7 + 3t^6 - 4t^5 + 3t^4 - 3t^3 + 3t^2 - t + 1$	-43848.13348998
8_{20}		yes	$t^4 - 3t^3 + 5t^2 - 6t + 7 - \frac{6}{t} + \frac{5}{t^2} - \frac{3}{t^3} + \frac{1}{t^4}$	0

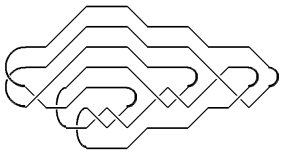
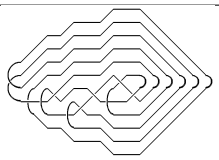
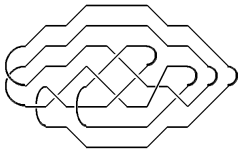
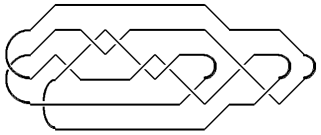
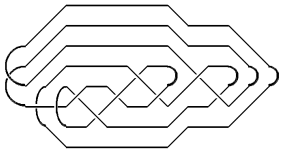
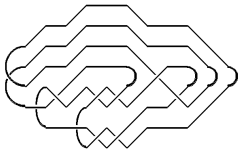
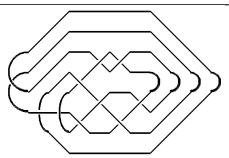
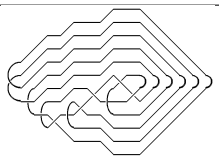
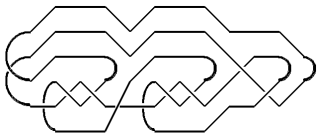
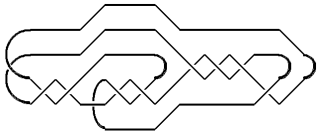
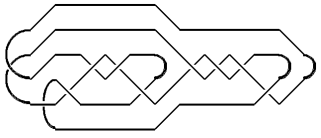
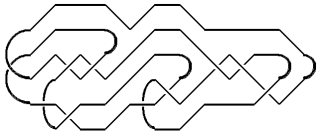
8_{21}		no	$-t^6+3t^5-5t^4+6t^3-6t^2+6t-4+\frac{3}{t}-\frac{1}{t^2}$	21110.55692072
9_1		no	$-t^{13}+t^{12}-t^{11}+t^{10}-t^9$ $+t^8-t^7+t^6+t^4$	55977275.98225386
9_2		no	$-t^{10}+t^9-t^8+2t^7-2t^6+2t^5$ $-2t^4+2t^3-t^2+t$	1703.62860367
9_3		no	$-t^{12}+t^{11}-2t^{10}+3t^9-3t^8+3t^7$ $-2t^6+2t^5-t^4+t^3$	3162227.81013260
9_4		no	$-t^{11}+t^{10}-2t^9+3t^8-3t^7+4t^6$ $-3t^5+2t^4-t^3+t^2$	92977.86917356
9_5		no	$-t^{12}+2t^{11}-3t^{10}+4t^9-5t^8+4t^7$ $-3t^6+3t^5-t^4+t^3$	2295159.90658894
9_6		no	$-t^{11}+2t^{10}-4t^9+6t^8-7t^7+7t^6$ $-6t^5+5t^4-2t^3+t^2$	120350.55747009
9_7		no	$t-2+\frac{4}{t}-\frac{6}{t^2}+\frac{7}{t^3}-\frac{6}{t^4}$ $+\frac{6}{t^5}-\frac{4}{t^6}+\frac{2}{t^7}-\frac{1}{t^8}$	253.90105979
9_8		no	$-t^{11}+2t^{10}-3t^9+4t^8-5t^7+5t^6$ $-4t^5+3t^4-t^3+t^2$	61045.13518824
9_9		no	$-t^{12}+3t^{11}-5t^{10}+6t^9-7t^8+6t^7$ $-5t^6+4t^5-t^4+t^3$	1748450.97142742
9_{10}		no	$-t^{12}+2t^{11}-4t^{10}+5t^9-5t^8+5t^7$ $-4t^6+3t^5-t^4+t^3$	2508341.39935153
9_{11}		no	$-t^{11}+t^{10}-3t^9+5t^8-5t^7+6t^6$ $-5t^5+4t^4-2t^3+t^2$	167592.89099559
9_{12}		no	$-t^{11}+2t^{10}-4t^9+5t^8-6t^7+7t^6$ $-5t^5+4t^4-2t^3+t^2$	140754.54357990

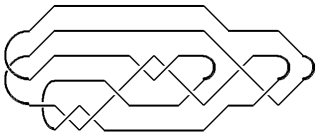
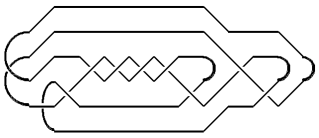
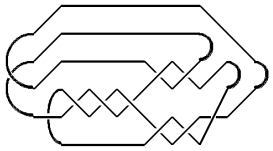
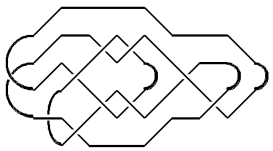
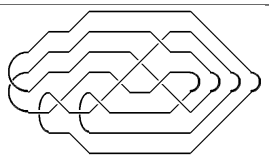
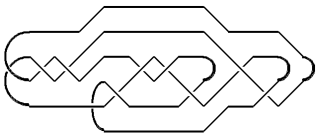
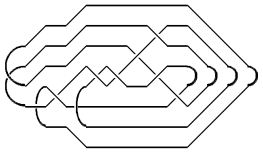
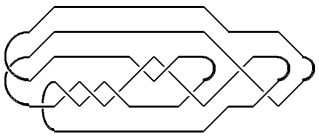
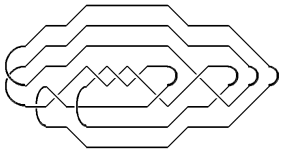
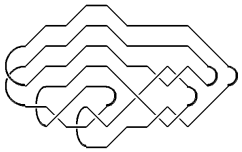
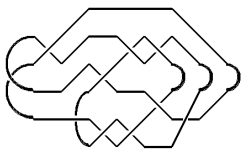
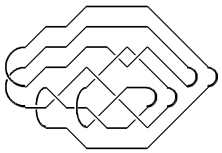
9_{13}		no	$t^4-3t^3+5t^2-7t+8-\frac{7}{t}$ $+\frac{7}{t^2}-\frac{4}{t^3}+\frac{2}{t^4}-\frac{1}{t^5}$	-131.88559406
9_{14}		no	$-t^8+2t^7-3t^6+5t^5-6t^4+6t^3$ $-5t^2+4t-2+\frac{1}{t}$	-426.93709576
9_{15}		no	$-t^9+2t^8-4t^7+5t^6-5t^5+6t^4$ $-4t^3+3t^2-2t+1$	-11099.13825720
9_{16}		no	$-t^{11}+3t^{10}-5t^9+6t^8-8t^7+8t^6$ $-6t^5+5t^4-2t^3+t^2$	98579.47581842
9_{17}		no	$t^4-2t^3+4t^2-6t+7-\frac{7}{t}$ $+\frac{6}{t^2}-\frac{4}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	-61.30247408
9_{18}		no	$-t^5+3t^4-5t^3+7t^2-8t+9$ $-\frac{7}{t}+\frac{5}{t^2}-\frac{3}{t^3}+\frac{1}{t^4}$	791.34483471
9_{19}		no	$-t^8+2t^7-4t^6+6t^5-7t^4+8t^3$ $-6t^2+5t-3+\frac{1}{t}$	-771.86884794
9_{20}		no	$-t^8+3t^7-5t^6+7t^5-8t^4+8t^3$ $-7t^2+5t-2+\frac{1}{t}$	-2087.35617931
9_{21}		no	$-t^8+2t^7-3t^6+4t^5-4t^4+4t^3-3t^2+2t$	-556.44271234
9_{22}		no	$-t^5+2t^4-2t^3+3t^2-3t+3-\frac{2}{t}+\frac{1}{t^2}$	-58.38762160
9_{23}		no	$t^3-t^2+t-1+\frac{1}{t}-\frac{1}{t^2}+\frac{1}{t^3}$	-559.23521393
9_{24}		no	$-t^7+2t^6-2t^5+2t^4-2t^3+2t^2-t+1$	-18628.72913996

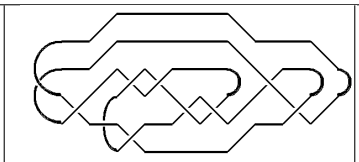
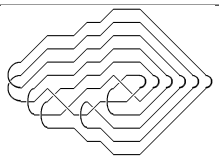
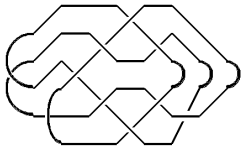
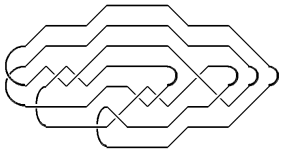
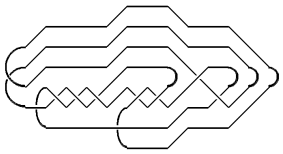
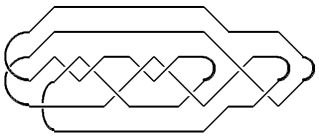
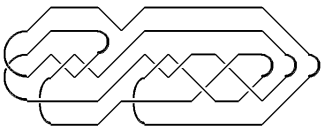
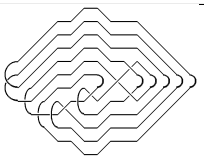
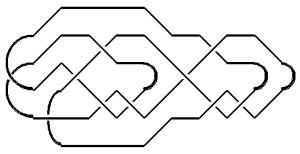
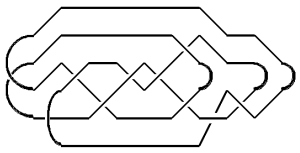
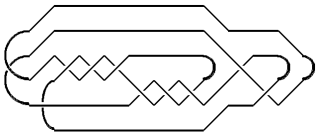
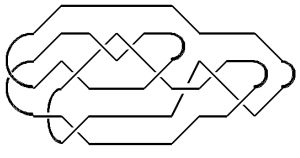
9_{25}		no	$-t^9+2t^8-4t^7+6t^6-6t^5+6t^4$ $-5t^3+4t^2-2t+1$	-5373.75709188
9_{26}		no	$-t^6+2t^5-3t^4+5t^3-5t^2+5t$ $-4+\frac{3}{t}-\frac{2}{t^2}+\frac{1}{t^3}$	-411.14966766
9_{27}		no	$-t^9+3t^8-5t^7+6t^6-7t^5+7t^4$ $-5t^3+4t^2-2t+1$	-6452.27028104
9_{28}		no	$-t^6+3t^5-4t^4+6t^3-7t^2+6t$ $-5+\frac{4}{t}-\frac{2}{t^2}+\frac{1}{t^3}$	1621.28961392
9_{29}		no	$t^7-3t^6+5t^5-8t^4+9t^3-8t^2$ $+8t-5+\frac{3}{t}-\frac{1}{t^2}$	11917.74398254
9_{30}		no	$-t^{10}+t^9-2t^8+3t^7-3t^6+4t^5$ $-3t^4+3t^3-2t^2+t$	4667.81709316
9_{31}		no	$t^6-2t^5+3t^4-5t^3+6t^2-6t$ $+6-\frac{4}{t}+\frac{3}{t^2}-\frac{1}{t^3}$	216.70893245
9_{32}		no	$-t^6+3t^5-5t^4+7t^3-7t^2+7t$ $-6+\frac{4}{t}-\frac{2}{t^2}+\frac{1}{t^3}$	3057.38632237
9_{33}		no	$t^7-3t^6+5t^5-7t^4+8t^3-8t^2$ $+7t-4+\frac{3}{t}-\frac{1}{t^2}$	9566.53412430
9_{34}		no	$t^7-4t^6+6t^5-8t^4+10t^3-9t^2$ $+8t-5+\frac{3}{t}-\frac{1}{t^2}$	19121.63397011
9_{35}		no	$t^4-3t^3+6t^2-8t+9-\frac{9}{t}$ $+\frac{8}{t^2}-\frac{5}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	-721.08356759
9_{36}		no	$t^6-t^5+t^4-2t^3+t^2-t+2$	-3.67246948

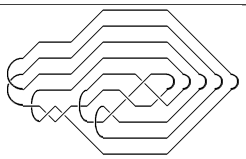
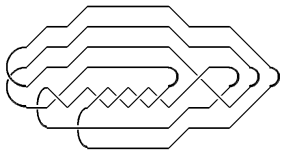
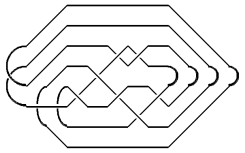
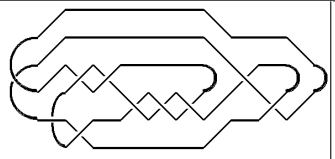
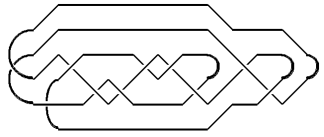
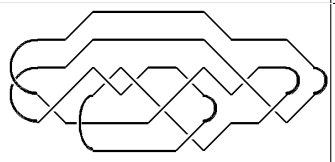
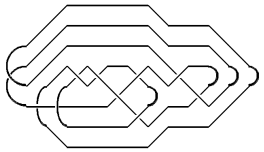
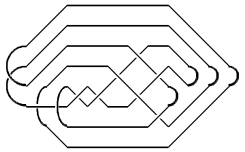
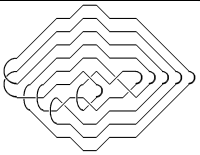
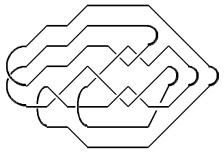
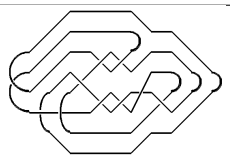
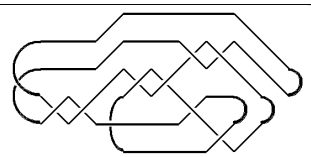
9_{37}		no	$-2t^6+3t^5-4t^4+6t^3-4t^2+4t-3+\frac{1}{t}$	-585.74214114
9_{38}		no	$t^4-2t^3+5t^2-7t+7-\frac{8}{t}$ $+\frac{7}{t^2}-\frac{4}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	-35.03726754
9_{39}		no	$-t^5+4t^4-7t^3+10t^2-12t+12$ $-\frac{10}{t}+\frac{8}{t^2}-\frac{4}{t^3}+\frac{1}{t^4}$	-243.33939840
9_{40}		no	$-t^2+3t-3+\frac{5}{t}-\frac{5}{t^2}+\frac{4}{t^3}-\frac{4}{t^4}+\frac{2}{t^5}$	-9904.81243087
9_{41}		no	$-t^{10}+t^9-3t^8+4t^7-3t^6+5t^5$ $-4t^4+3t^3-2t^2+t$	7153.80817476
9_{42}		no	$t^7-3t^6+6t^5-9t^4+10t^3-10t^2$ $+9t-6+\frac{4}{t}-\frac{1}{t^2}$	1213.17494926
9_{43}		no	$t^4-4t^3+7t^2-9t+11-\frac{10}{t}$ $+\frac{9}{t^2}-\frac{6}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	743.44919110
9_{44}		no	$t^7-4t^6+8t^5-11t^4+13t^3-13t^2$ $+11t-8+\frac{5}{t}-\frac{1}{t^2}$	23.82356102
9_{45}		no	$-2t^9+3t^8-4t^7+5t^6-4t^5+4t^4-2t^3+t^2$	68794.31392834
9_{46}		no	$-t^8+3t^7-6t^6+8t^5-9t^4+10t^3$ $-8t^2+6t-3+\frac{1}{t}$	-713.22313012
9_{47}		no	$t^6-3t^5+5t^4-7t^3+8t^2-8t$ $+8-\frac{5}{t}+\frac{3}{t^2}-\frac{1}{t^3}$	628.06624703
9_{48}		no	$-t^{11}+3t^{10}-6t^9+8t^8-10t^7+10t^6$ $-8t^5+7t^4-3t^3+t^2$	170976.50234378

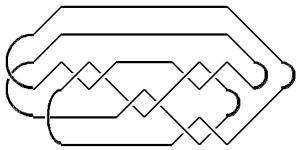
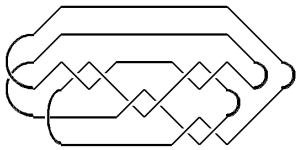
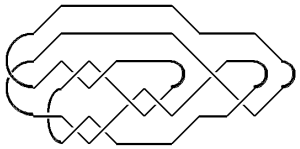
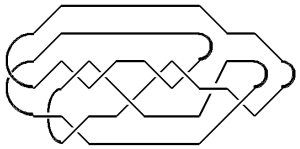
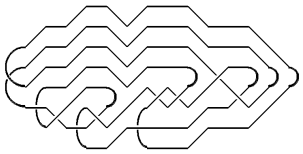
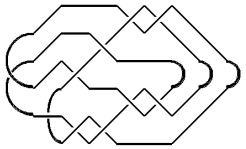
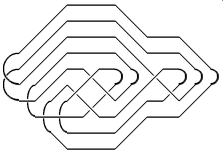
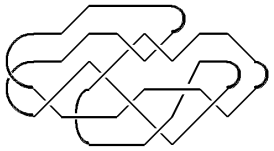
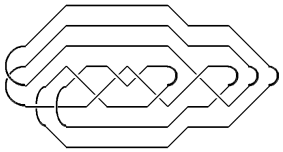
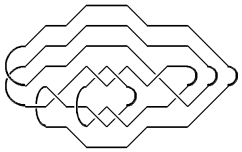
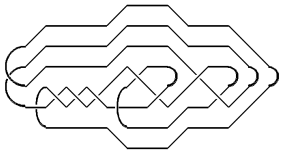
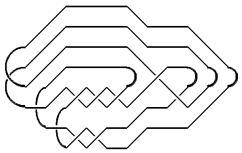
9_{49}		no	$t^3-3t^2+5t-7+\frac{9}{t}-\frac{8}{t^2}$ $+\frac{8}{t^3}-\frac{6}{t^4}+\frac{3}{t^5}-\frac{1}{t^6}$	-9122.14746498
10_1		no	$t^8-t^7+t^6-2t^5+2t^4-2t^3$ $2t^2-2t+2-\frac{1}{t}+\frac{1}{t^2}$	-175.73979853
10_2		no	$t^6-t^5+2t^4-3t^3+3t^2-4t$ $4-\frac{3}{t}+\frac{2}{t^2}-\frac{1}{t^3}+\frac{1}{t^4}$	-298.73539588
10_3		no	$t^{10}-2t^9+3t^8-5t^7+6t^6-6t^5$ $5t^4-4t^3+3t^2-t+1$	-343316.46968421
10_4		no	$t^6-2t^5+4t^4-6t^3+8t^2-9t$ $8-\frac{7}{t}+\frac{5}{t^2}-\frac{2}{t^3}+\frac{1}{t^4}$	-181.91413603
10_5		no	$t^9-2t^8+4t^7-7t^6+8t^5-9t^4$ $9t^3-7t^2+5t-2+\frac{1}{t}$	-15141.74260497
10_6		no	$t^7-3t^6+6t^5-9t^4+11t^3-11t^2$ $10t-8+\frac{5}{t}-\frac{2}{t^2}+\frac{1}{t^3}$	2462.10495444
10_7		no	$t^4-2t^3+4t^2-6t+8-\frac{8}{t}$ $\frac{7}{t^2}-\frac{6}{t^3}+\frac{4}{t^4}-\frac{2}{t^5}+\frac{1}{t^6}$	-123.95241071
10_8		no	$t^4-2t^3+4t^2-6t+8-\frac{8}{t}$ $\frac{7}{t^2}-\frac{6}{t^3}+\frac{4}{t^4}-\frac{2}{t^5}+\frac{1}{t^6}$	28589.81110080
10_9		yes	$-t^5+2t^4-4t^3+7t^2-8t+9$ $-\frac{8}{t}+\frac{7}{t^2}-\frac{4}{t^3}+\frac{2}{t^4}-\frac{1}{t^5}$	0
10_{10}		no	$t^9-2t^8+3t^7-4t^6+5t^5-6t^4$ $5t^3-4t^2+3t-1+\frac{1}{t}$	-10948.06690071
10_{11}		no	$t^{10}-3t^9+6t^8-9t^7+10t^6-11t^5$ $10t^4-7t^3+5t^2-2t+1$	-160596.69332673
10_{12}		no	$t^{10}-3t^9+6t^8-9t^7+10t^6-11t^5$ $10t^4-7t^3+5t^2-2t+1$	-153074.61777697

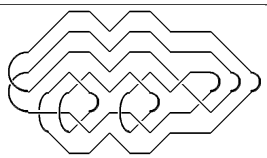
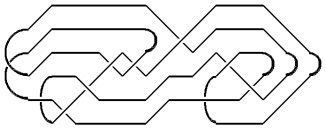
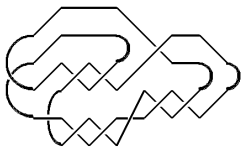
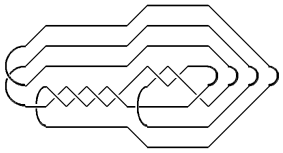
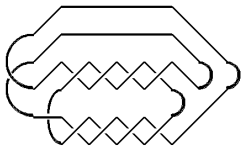
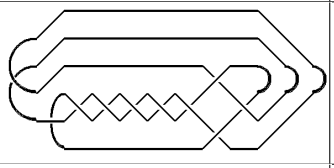
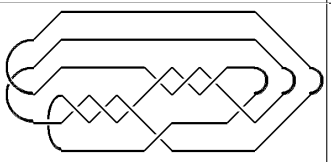
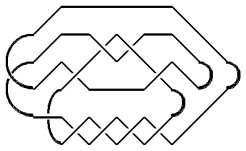
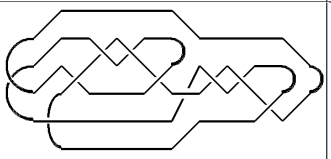
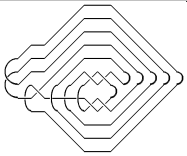
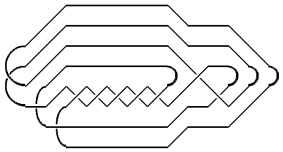
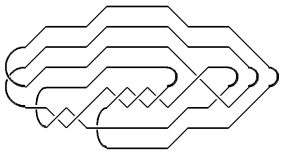
10_{13}		no	$t^7-3t^6+6t^5-8t^4+10t^3-11t^2$ $9t-7+\frac{5}{t}-\frac{2}{t^2}+\frac{1}{t^3}$	1314.68102716
10_{14}		no	$-t^2+3t-6+\frac{10}{t}-\frac{11}{t^2}+\frac{13}{t^3}$ $-\frac{12}{t^4}+\frac{9}{t^5}-\frac{6}{t^6}+\frac{3}{t^7}-\frac{1}{t^8}$	-234867.24463840
10_{15}		no	$-t^2+3t-6+\frac{10}{t}-\frac{11}{t^2}+\frac{13}{t^3}$ $-\frac{12}{t^4}+\frac{9}{t^5}-\frac{6}{t^6}+\frac{3}{t^7}-\frac{1}{t^8}$	-201492.64519930
10_{16}		no	$-t^5+3t^4-6t^3+10t^2-12t+13$ $-\frac{12}{t}+\frac{10}{t^2}-\frac{6}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	-72098.73577109
10_{17}		no	$-t^5+3t^4-6t^3+10t^2-12t+13$ $-\frac{12}{t}+\frac{10}{t^2}-\frac{6}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	-2737.37302647
10_{18}		no	$1-\frac{2}{t}+\frac{5}{t^2}-\frac{7}{t^3}+\frac{9}{t^4}-\frac{10}{t^5}$ $\frac{10}{t^6}-\frac{8}{t^7}+\frac{5}{t^8}-\frac{3}{t^9}+\frac{1}{t^{10}}$	120215.74902230
10_{19}		no	$-t^5+3t^4-6t^3+9t^2-11t+13$ $-\frac{11}{t}+\frac{9}{t^2}-\frac{6}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	-83380.44187352
10_{20}		yes	$-t^5+3t^4-6t^3+9t^2-11t+13$ $-\frac{11}{t}+\frac{9}{t^2}-\frac{6}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	0
10_{21}		no	$t^{10}-3t^9+6t^8-8t^7+9t^6-10t^5$ $8t^4-6t^3+4t^2-t+1$	-318528.13876202
10_{22}		no	$t^7-2t^6+4t^5-6t^4+7t^3-7t^2$ $6t-5+\frac{3}{t}-\frac{1}{t^2}+\frac{1}{t^3}$	-29677.52124377
10_{23}		no	$t^7-2t^6+4t^5-6t^4+7t^3-8t^2$ $7t-5+\frac{4}{t}-\frac{2}{t^2}+\frac{1}{t^3}$	-24334.92940128
10_{24}		no	$-t^8+3t^7-6t^6+8t^5-10t^4+11t^3$ $-9t^2+8t-4+\frac{2}{t}-\frac{1}{t^2}$	337644.89476648

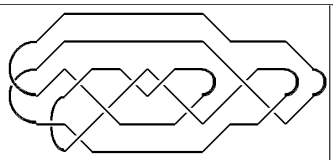
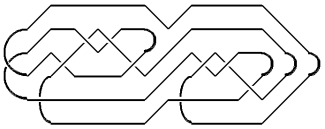
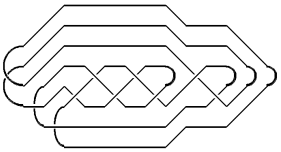
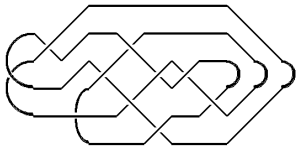
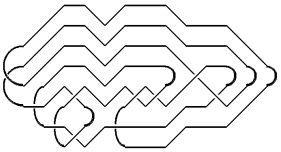
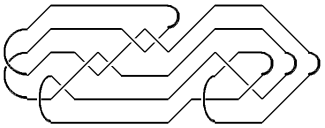
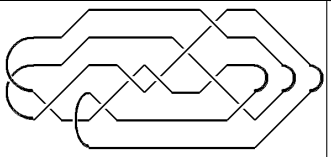
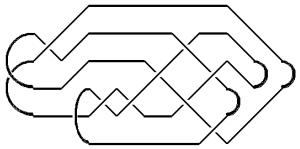
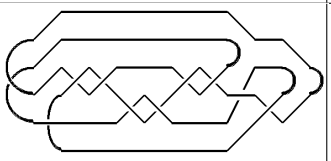
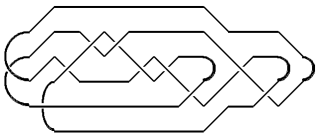
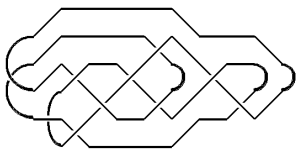
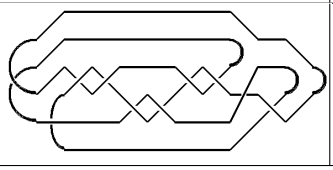
10_{25}		no	$t^9 - 2t^8 + 4t^7 - 8t^6 + 9t^5 - 10t^4$ $11t^3 - 8t^2 + 6t - 3 + \frac{1}{t}$	-8001.75471318
10_{26}		no	$t^9 - 2t^8 + 3t^7 - 5t^6 + 6t^5 - 7t^4$ $7t^3 - 5t^2 + 4t - 2 + \frac{1}{t}$	-5033.33513395
10_{27}		no	$-t^8 + 2t^7 - 4t^6 + 6t^5 - 7t^4 + 8t^3$ $-7t^2 + 6t - 3 + \frac{2}{t} - \frac{1}{t^2}$	398507.45657213
10_{28}		no	$-t^{10} + t^9 - 2t^8 + 2t^7 - t^6 + 2t^5 - t^4 + t^3$	2643717.21993567
10_{29}		no	$-t^5 + 2t^4 - 3t^3 + 4t^2 - 4t$ $5 - \frac{3}{t} + \frac{2}{t^2} - \frac{1}{t^3}$	-1815.70634548
10_{30}		no	$t^{10} - 2t^9 + 4t^8 - 7t^7 + 8t^6 - 9t^5$ $8t^4 - 6t^3 + 5t^2 - 2t + 1$	-267889.33296826
10_{31}		no	$-t + 2 - \frac{2}{t} + \frac{3}{t^2} - \frac{2}{t^3}$ $\frac{3}{t^4} - \frac{2}{t^5} + \frac{1}{t^6} - \frac{1}{t^7}$	-2219.23612141
10_{32}		no	$t^{10} - 2t^9 + 3t^8 - 6t^7 + 7t^6 - 7t^5$ $7t^4 - 5t^3 + 4t^2 - 2t + 1$	-174132.82655781
10_{33}		no	$-t^6 + 2t^5 - 4t^4 + 6t^3 - 6t^2 + 7t$ $-6 + \frac{5}{t} - \frac{3}{t^2} + \frac{2}{t^3} - \frac{1}{t^4}$	250266.25849180
10_{34}		no	$t^3 - 3t^2 + 6t - 8 + \frac{11}{t} - \frac{12}{t^2}$ $\frac{11}{t^3} - \frac{9}{t^4} + \frac{6}{t^5} - \frac{3}{t^6} + \frac{1}{t^7}$	-950.21043709
10_{35}		no	$t^3 - 3t^2 + 6t - 8 + \frac{11}{t} - \frac{12}{t^2}$ $\frac{11}{t^3} - \frac{9}{t^4} + \frac{6}{t^5} - \frac{3}{t^6} + \frac{1}{t^7}$	1339520.47904792
10_{36}		no	$t - 2 + \frac{5}{t} - \frac{7}{t^2} + \frac{9}{t^3} - \frac{10}{t^4}$ $\frac{9}{t^5} - \frac{7}{t^6} + \frac{5}{t^7} - \frac{3}{t^8} + \frac{1}{t^9}$	10751.15764227

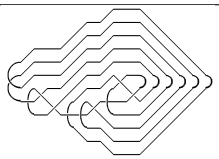
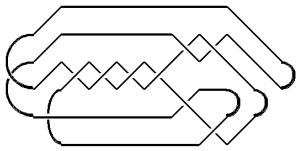
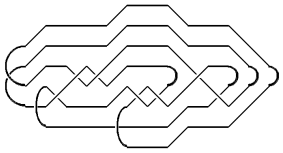
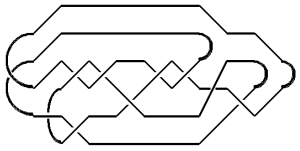
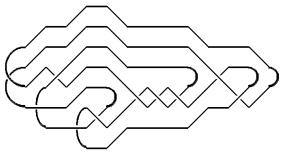
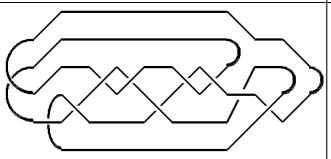
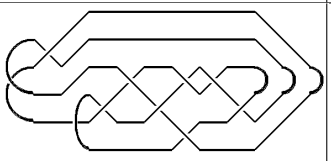
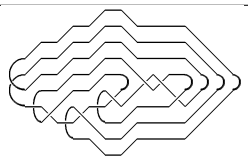
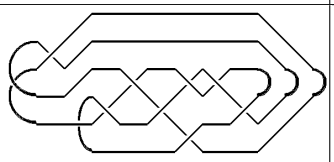
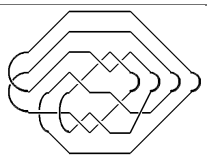
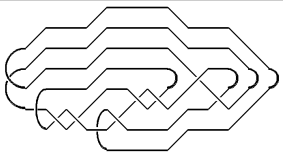
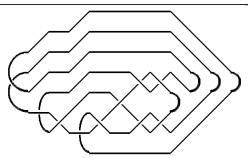
10_{37}		no	$t^6-3t^5+6t^4-8t^3+10t^2-11t$ $10-\frac{8}{t}+\frac{5}{t^2}-\frac{2}{t^3}+\frac{1}{t^4}$	-40.23486115
10_{38}		no	$t^6-2t^5+3t^4-4t^3+4t^2$ $-4t+4-\frac{2}{t}+\frac{1}{t^2}$	4.12577855
10_{39}		no	$t^6-2t^5+3t^4-4t^3+4t^2$ $-4t+4-\frac{2}{t}+\frac{1}{t^2}$	82.12653931
10_{40}		no	$t^3-2t^2+2t-2+\frac{3}{t}-\frac{2}{t^2}+\frac{2}{t^3}-\frac{1}{t^4}$	-8.80546837
10_{41}		no	$t-2+\frac{4}{t}-\frac{6}{t^2}+\frac{8}{t^3}-\frac{8}{t^4}$ $\frac{8}{t^5}-\frac{6}{t^6}+\frac{4}{t^7}-\frac{3}{t^8}+\frac{1}{t^9}$	1570.52521891
10_{42}		no	$t^9-3t^8+5t^7-8t^6+10t^5-11t^4$ $11t^3-8t^2+6t-3+\frac{1}{t}$	-4990.91358347
10_{43}		no	$t^3-3t^2+6t-9+\frac{12}{t}-\frac{13}{t^2}$ $\frac{13}{t^3}-\frac{10}{t^4}+\frac{7}{t^5}-\frac{4}{t^6}+\frac{1}{t^7}$	-14.58008503
10_{44}		no	$-t^8+3t^7-6t^6+10t^5-13t^4+14t^3$ $-13t^2+11t-7+\frac{4}{t}-\frac{1}{t^2}$	3822.91788154
10_{45}		no	$-t^8+3t^7-6t^6+10t^5-13t^4+14t^3$ $-13t^2+11t-7+\frac{4}{t}-\frac{1}{t^2}$	110303.79382269
10_{46}		no	$-t^7+t^6-t^5+t^4+t^2$	884.04250641
10_{47}		no	$-t^2+3t-5+\frac{8}{t}-\frac{9}{t^2}+\frac{10}{t^3}$ $-\frac{9}{t^4}+\frac{7}{t^5}-\frac{4}{t^6}+\frac{2}{t^7}-\frac{1}{t^8}$	-293648.46946884
10_{48}		no	$t^4-3t^3+6t^2-8t+10-\frac{10}{t}$ $\frac{9}{t^2}-\frac{7}{t^3}+\frac{4}{t^4}-\frac{2}{t^5}+\frac{1}{t^6}$	27576.32594090

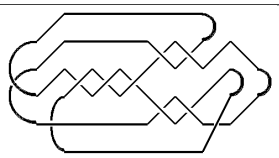
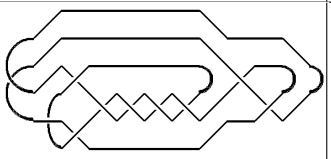
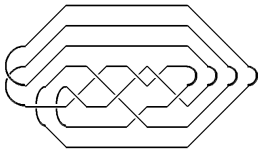
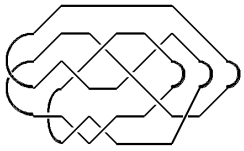
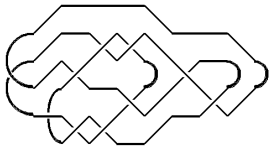
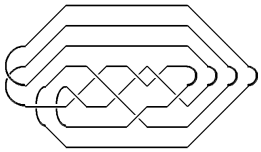
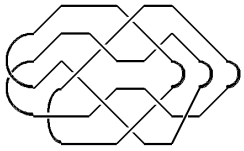
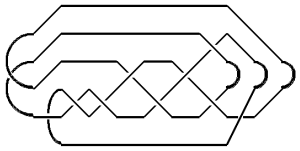
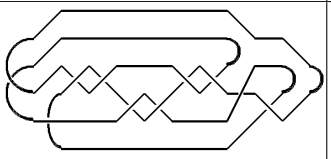
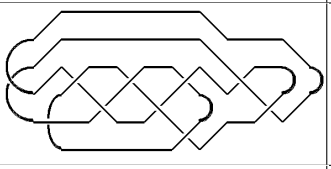
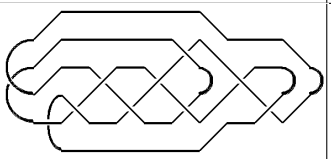
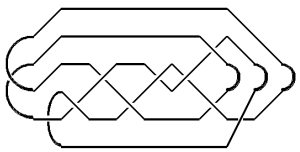
10_{49}		no	$1-\frac{2}{t}+\frac{5}{t^2}-\frac{8}{t^3}+\frac{11}{t^4}-\frac{12}{t^5}$ $\frac{12}{t^6}-\frac{10}{t^7}+\frac{7}{t^8}-\frac{4}{t^9}+\frac{1}{t^{10}}$	51197.10772589
10_{50}		no	$1-\frac{2}{t}+\frac{4}{t^2}-\frac{6}{t^3}+\frac{9}{t^4}-\frac{9}{t^5}$ $\frac{9}{t^6}-\frac{8}{t^7}+\frac{5}{t^8}-\frac{3}{t^9}+\frac{1}{t^{10}}$	91436.61063544
10_{51}		no	$t^9-2t^8+3t^7-5t^6+5t^5$ $-5t^4+5t^3-3t^2+2t$	-1148.53413559
10_{52}		no	$-t^2+3t-6+\frac{9}{t}-\frac{10}{t^2}+\frac{12}{t^3}$ $-\frac{10}{t^4}+\frac{8}{t^5}-\frac{5}{t^6}+\frac{2}{t^7}-\frac{1}{t^8}$	-379415.63038096
10_{53}		no	$-t^2+3t-5+\frac{9}{t}-\frac{11}{t^2}+\frac{12}{t^3}$ $-\frac{11}{t^4}+\frac{9}{t^5}-\frac{6}{t^6}+\frac{3}{t^7}-\frac{1}{t^8}$	-206764.25198795
10_{54}		no	$t^{11}-3t^{10}+3t^9-4t^8+4t^7$ $-3t^6+3t^5-t^4+t^3$	2063846.82187560
10_{55}		no	$-t^5+2t^4-4t^3+6t^2-6t$ $7-\frac{5}{t}+\frac{4}{t^2}-\frac{2}{t^3}$	1316.82945449
10_{56}		no	$t^9-2t^8+2t^7-3t^6+3t^5$ $-3t^4+3t^3-t^2+t$	-830.59272707
10_{57}		no	$-t^2+3t-6+\frac{10}{t}-\frac{12}{t^2}+\frac{14}{t^3}$ $-\frac{12}{t^4}+\frac{10}{t^5}-\frac{7}{t^6}+\frac{3}{t^7}-\frac{1}{t^8}$	-277578.97890084
10_{58}		no	$-t^5+2t^4-4t^3+7t^2-8t+10$ $-\frac{9}{t}+\frac{7}{t^2}-\frac{5}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	1919.40352229
10_{59}		no	$t^3-2t^2+4t-6+\frac{8}{t}-\frac{9}{t^2}$ $\frac{9}{t^3}-\frac{7}{t^4}+\frac{5}{t^5}-\frac{3}{t^6}+\frac{1}{t^7}$	15154.76575137
10_{60}		no	$t^{12}-3t^{11}+5t^{10}-8t^9+9t^8-10t^7$ $10t^6-7t^5+5t^4-2t^3+t^2$	152312.68391019

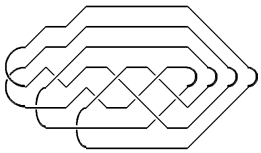
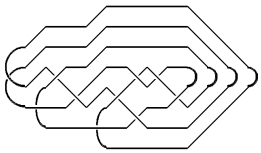
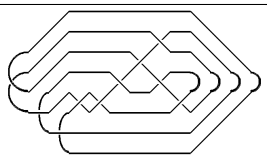
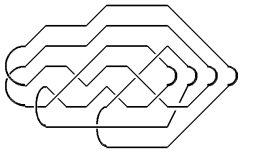
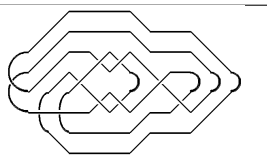
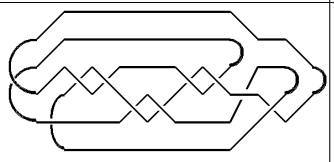
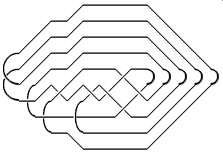
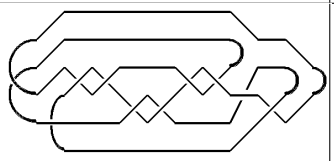
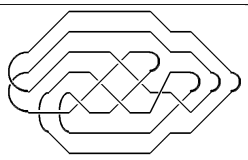
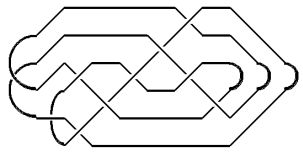
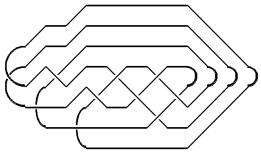
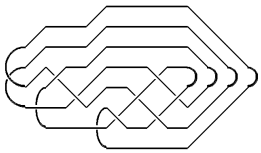
10_{61}		no	$-t^{10}+t^9-t^8+t^7+t^2$	12169.14079597
10_{62}		no	$-t^3+3t^2-4t+6-\frac{6}{t}$ $\frac{5}{t^2}-\frac{4}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	197.54125921
10_{63}		no	$t^9-3t^8+5t^7-8t^6+10t^5-10t^4$ $10t^3-8t^2+5t-2+\frac{1}{t}$	-7092.92877370
10_{64}		no	$t^3-2t^2+3t-4+\frac{5}{t}$ $-\frac{4}{t^2}+\frac{4}{t^3}-\frac{3}{t^4}+\frac{1}{t^5}$	1306.93525273
10_{65}		no	$t^3-3t^2+6t-9+\frac{12}{t}-\frac{12}{t^2}$ $\frac{12}{t^3}-\frac{10}{t^4}+\frac{6}{t^5}-\frac{3}{t^6}+\frac{1}{t^7}$	-36.86502420
10_{66}		no	$t^3-3t^2+6t-9+\frac{12}{t}-\frac{12}{t^2}$ $\frac{12}{t^3}-\frac{10}{t^4}+\frac{6}{t^5}-\frac{3}{t^6}+\frac{1}{t^7}$	1164922.76913909
10_{67}		no	$t^3-2t^2+4t-5+\frac{6}{t}$ $-\frac{6}{t^2}+\frac{5}{t^3}-\frac{4}{t^4}+\frac{2}{t^5}$	-229.38060402
10_{68}		no	$-t^5+4t^4-7t^3+10t^2-13t+14$ $-\frac{12}{t}+\frac{10}{t^2}-\frac{6}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	1232.78998548
10_{69}		no	$t^4-3t^3+6t^2-9t+11-\frac{11}{t}$ $\frac{11}{t^2}-\frac{8}{t^3}+\frac{5}{t^4}-\frac{3}{t^5}+\frac{1}{t^6}$	29565.07033800
10_{70}		no	$-t^6+2t^5-4t^4+6t^3-7t^2+8t$ $-6+\frac{6}{t}-\frac{4}{t^2}+\frac{2}{t^3}-\frac{1}{t^4}$	277813.75102623
10_{71}		no	$-t^3+2t^2-3t+5-\frac{5}{t}+\frac{6}{t^2}$ $-\frac{5}{t^3}+\frac{4}{t^4}-\frac{3}{t^5}+\frac{2}{t^6}-\frac{1}{t^7}$	-5801.28271488
10_{72}		no	$-t^7+2t^6-4t^5+6t^4-7t^3+9t^2$ $-8t+7-\frac{5}{t}+\frac{3}{t^2}-\frac{1}{t^3}$	8466.12767979

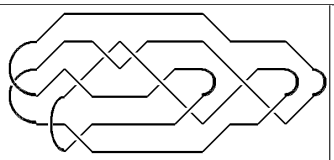
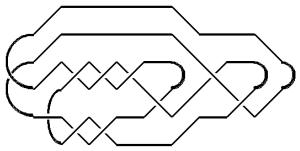
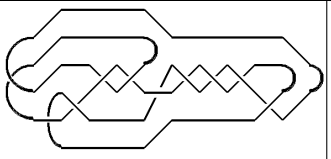
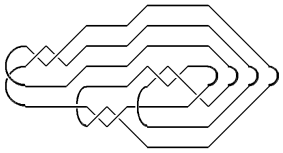
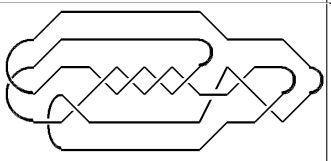
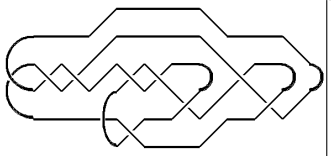
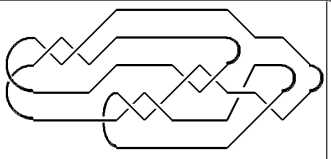
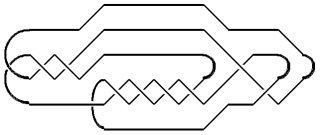
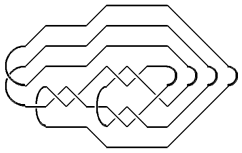
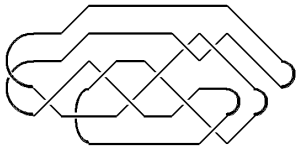
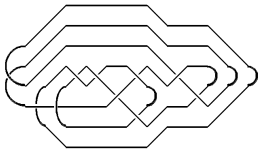
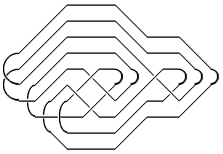
10_{73}		no	$t^{10} - 3t^9 + 5t^8 - 9t^7 + 11t^6 - 11t^5$ $11t^4 - 8t^3 + 6t^2 - 3t + 1$	-19643.61630430
10_{74}		no	$t^6 - 3t^5 + 6t^4 - 10t^3 + 12t^2 - 13t$ $14 - \frac{10}{t} + \frac{7}{t^2} - \frac{4}{t^3} + \frac{1}{t^4}$	-491.64774695
10_{75}		no	$\frac{1}{t} - \frac{1}{t^2} + \frac{2}{t^3} - \frac{2}{t^4} + \frac{3}{t^5} - \frac{3}{t^6}$ $\frac{3}{t^7} - \frac{3}{t^8} + \frac{2}{t^9} - \frac{2}{t^{10}} + \frac{1}{t^{11}}$	3268202.37622739
10_{76}		no	$t^5 - 2t^4 + 3t^3 - 3t^2 + 4t - 4$ $\frac{3}{t} - \frac{3}{t^2} + \frac{2}{t^3} - \frac{1}{t^4} + \frac{1}{t^5}$	-12645.57295491
10_{77}		no	$t^2 - t + 2 - \frac{3}{t} + \frac{4}{t^2} - \frac{4}{t^3}$ $\frac{4}{t^4} - \frac{4}{t^5} + \frac{3}{t^6} - \frac{2}{t^7} + \frac{1}{t^8}$	574565.48390627
10_{78}		no	$-t^9 + 2t^8 - 3t^7 + 4t^6 - 5t^5 + 5t^4$ $-4t^3 + 4t^2 - 2t + 2 - \frac{1}{t}$	6908070.79394631
10_{79}		yes	$-t^5 + 2t^4 - 3t^3 + 5t^2 - 6t + 7$ $-\frac{6}{t} + \frac{5}{t^2} - \frac{3}{t^3} + \frac{2}{t^4} - \frac{1}{t^5}$	0
10_{80}		no	$t^3 - 2t^2 + 3t - 4 + \frac{6}{t} - \frac{6}{t^2}$ $\frac{6}{t^3} - \frac{5}{t^4} + \frac{3}{t^5} - \frac{2}{t^6} + \frac{1}{t^7}$	2379943.94310624
10_{81}		no	$-t^6 + 3t^5 - 5t^4 + 7t^3 - 8t^2 + 8t$ $-7 + \frac{6}{t} - \frac{3}{t^2} + \frac{2}{t^3} - \frac{1}{t^4}$	119426.99072403
10_{82}		no	$-t^6 + 3t^5 - 6t^4 + 8t^3 - 9t^2 + 10t$ $-8 + \frac{7}{t} - \frac{4}{t^2} + \frac{2}{t^3} - \frac{1}{t^4}$	167585.82585146
10_{83}		no	$-t^7 + 2t^6 - 3t^5 + 5t^4 - 6t^3 + 7t^2$ $-7t + 6 - \frac{4}{t} + \frac{3}{t^2} - \frac{1}{t^3}$	4067.22332697
10_{84}		no	$-t^7 + 2t^6 - 4t^5 + 7t^4 - 8t^3 + 9t^2$ $-9t + 8 - \frac{5}{t} + \frac{3}{t^2} - \frac{1}{t^3}$	6952.61904181

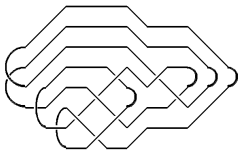
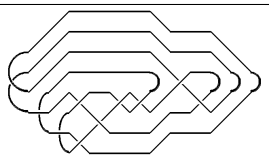
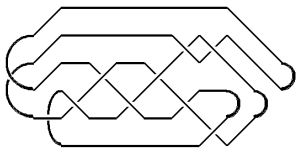
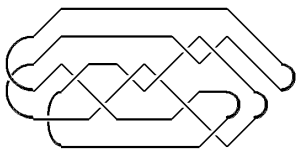
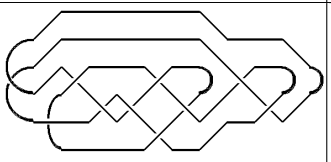
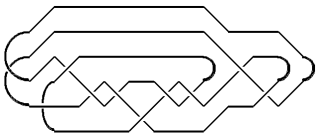
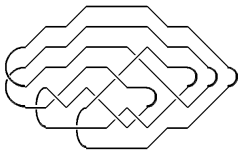
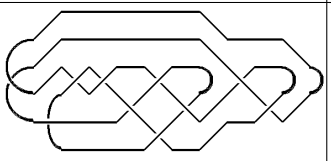
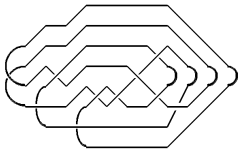
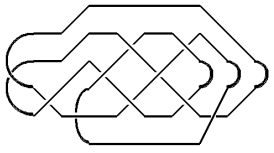
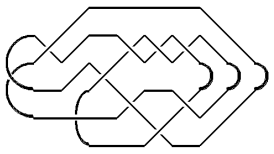
10_{85}		no	$t^{12} - 3t^{11} + 5t^{10} - 9t^9 + 11t^8 - 12t^7$ $12t^6 - 9t^5 + 7t^4 - 3t^3 + t^2$	247880.96685665
10_{86}		yes	$-t^5 + 3t^4 - 5t^3 + 8t^2 - 10t + 11$ $-\frac{10}{t} + \frac{8}{t^2} - \frac{5}{t^3} + \frac{3}{t^4} - \frac{1}{t^5}$	0
10_{87}		yes	$-t^5 + 4t^4 - 7t^3 + 11t^2 - 14t + 15$ $-\frac{14}{t} + \frac{11}{t^2} - \frac{7}{t^3} + \frac{4}{t^4} - \frac{1}{t^5}$	0
10_{88}		no	$t^6 - 3t^5 + 6t^4 - 10t^3 + 13t^2 - 14t$ $14 - \frac{11}{t} + \frac{8}{t^2} - \frac{4}{t^3} + \frac{1}{t^4}$	-12608.73813397
10_{89}		no	$t^6 - 3t^5 + 6t^4 - 10t^3 + 13t^2 - 14t$ $14 - \frac{11}{t} + \frac{8}{t^2} - \frac{4}{t^3} + \frac{1}{t^4}$	133.14372202
10_{90}		no	$t^{12} - 2t^{11} + 2t^{10} - 3t^9 + 2t^8 - 2t^7 + 2t^6 + t^3$	640860.36739280
10_{91}		no	$t^8 - 3t^7 + 4t^6 - 5t^5 + 5t^4$ $-4t^3 + 4t^2 - 2t + 1$	-10333.43066915
10_{92}		no	$-2t^6 + 4t^5 - 6t^4 + 8t^3 - 7t^2$ $7t - 5 + \frac{3}{t} - \frac{1}{t^2}$	40179.82674103
10_{93}		yes	$-t^5 + 3t^4 - 7t^3 + 11t^2 - 13t + 15$ $-\frac{13}{t} + \frac{11}{t^2} - \frac{7}{t^3} + \frac{3}{t^4} - \frac{1}{t^5}$	0
10_{94}		no	$-t^6 + 3t^5 - 5t^4 + 6t^3 - 6t^2$ $6t - 4 + \frac{3}{t} - \frac{1}{t^2}$	21110.55692058
10_{95}		no	$-t^6 + 3t^5 - 5t^4 + 6t^3 - 6t^2$ $6t - 4 + \frac{3}{t} - \frac{1}{t^2}$	25004.24552754
10_{96}		yes	$t^4 - 3t^3 + 5t^2 - 6t + 7$ $-\frac{6}{t} + \frac{5}{t^2} - \frac{3}{t^3} + \frac{1}{t^4}$	0

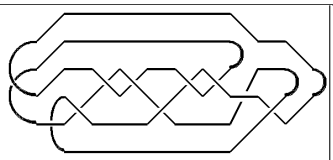
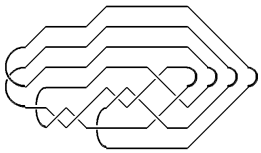
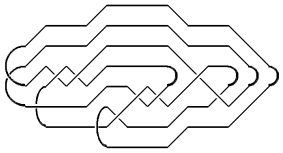
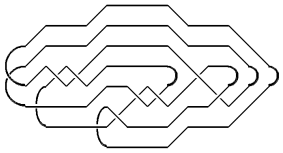
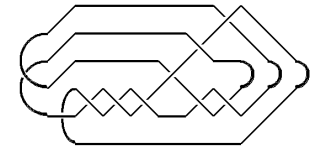
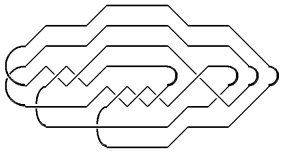
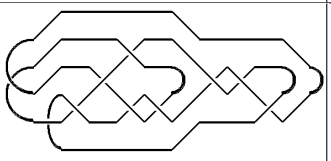
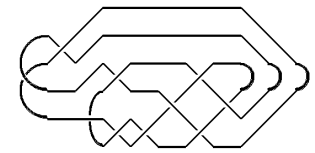
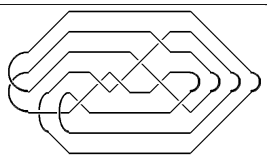
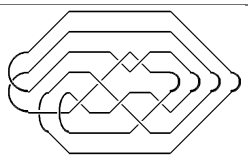
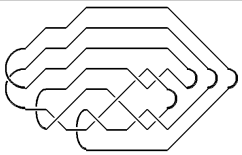
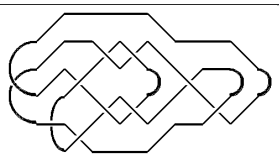
10_{97}		no	$-t^2+4t-7+\frac{11}{t}-\frac{14}{t^2}+\frac{15}{t^3}$ $-\frac{13}{t^4}+\frac{11}{t^5}-\frac{7}{t^6}+\frac{3}{t^7}-\frac{1}{t^8}$	-8196.09571876
10_{98}		no	$t^{12}-3t^{11}+4t^{10}-7t^9+9t^8-9t^7$ $9t^6-7t^5+5t^4-2t^3+t^2$	182397.88489512
10_{99}		no	$-t^7+t^6-t^5+2t^4-t^3+t^2-t+1$	311.75496511
10_{100}		no	$2t-3+\frac{5}{t}-\frac{7}{t^2}+\frac{7}{t^3}$ $-\frac{6}{t^4}+\frac{5}{t^5}-\frac{3}{t^6}+\frac{1}{t^7}$	7633.62317404
10_{101}		no	$t^{13}-4t^{12}+7t^{11}-10t^{10}+12t^9-13t^8$ $11t^7-8t^6+6t^5-2t^4+t^3$	4080836.04518546
10_{102}		no	$-t^8+2t^7-3t^6+4t^5-5t^4$ $5t^3-3t^2+3t-1$	67113.15662600
10_{103}		no	$t^2-2t+3-\frac{3}{t}+\frac{4}{t^2}$ $-\frac{3}{t^3}+\frac{2}{t^4}-\frac{2}{t^5}+\frac{1}{t^6}$	282.25305295
10_{104}		no	$t^{13}-3t^{12}+6t^{11}-10t^{10}+11t^9-12t^8$ $11t^7-8t^6+6t^5-2t^4+t^3$	3720706.07716963
10_{105}		no	$-t^8+2t^7-4t^6+5t^5-5t^4$ $6t^3-4t^2+3t-1$	86124.02686225
10_{106}		no	$t^{10}-3t^9+5t^8-7t^7+7t^6$ $-7t^5+6t^4-3t^3+2t^2$	-17525.52624503
10_{107}		no	$-t^5+t^4-t^3+t^2+1$ $\frac{1}{t}-\frac{1}{t^2}+\frac{1}{t^3}-\frac{1}{t^4}$	112928.58745497
10_{108}		no	$-2t^{10}+2t^9-2t^8+3t^7-2t^6+2t^5-t^4+t^3$	2882855.87398013

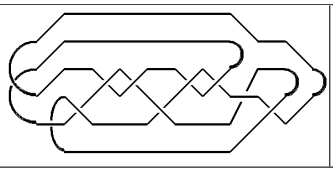
10_{109}		no	$-t^2+3t-5+\frac{8}{t}-\frac{10}{t^2}+\frac{11}{t^3}$ $-\frac{9}{t^4}+\frac{8}{t^5}-\frac{5}{t^6}+\frac{2}{t^7}-\frac{1}{t^8}$	-342907.90506704
10_{110}		no	$t^{13}-3t^{12}+5t^{11}-8t^{10}+9t^9-10t^8$ $9t^7-6t^6+5t^5-2t^4+t^3$	4578641.51257032
10_{111}		no	$t^{10}-2t^9+3t^8-5t^7+5t^6$ $-5t^5+4t^4-2t^3+2t^2$	-18811.24415598
10_{112}		no	$-t^4+t^3-t^2+2t-1$ $\frac{2}{t}-\frac{1}{t^2}+\frac{1}{t^3}-\frac{1}{t^4}$	-84417.17467416
10_{113}		no	$-1+\frac{2}{t}-\frac{2}{t^2}+\frac{4}{t^3}-\frac{3}{t^4}$ $\frac{3}{t^5}-\frac{2}{t^6}+\frac{1}{t^7}-\frac{1}{t^8}$	-134754.86501188
10_{114}		no	$-t^{10}+t^6+t^4$	47314554.82346105
10_{115}		yes	$-t^5+5t^4-10t^3+15t^2-19t+21$ $-\frac{19}{t}+\frac{15}{t^2}-\frac{10}{t^3}+\frac{5}{t^4}-\frac{1}{t^5}$	0
10_{116}		no	$t^7-4t^6+7t^5-11t^4+14t^3-14t^2$ $14t-10+\frac{7}{t}-\frac{4}{t^2}+\frac{1}{t^3}$	-127459.87994409
10_{117}		no	$-1+\frac{4}{t}-\frac{5}{t^2}+\frac{7}{t^3}-\frac{7}{t^4}$ $\frac{6}{t^5}-\frac{5}{t^6}+\frac{3}{t^7}-\frac{1}{t^8}$	-40086.09270555
10_{118}		no	$-t^8+5t^7-10t^6+14t^5-18t^4+19t^3$ $-17t^2+14t-8+\frac{4}{t}-\frac{1}{t^2}$	43305.33490236
10_{119}		no	$t^7-4t^6+8t^5-12t^4+15t^3-16t^2$ $15t-11+\frac{8}{t}-\frac{4}{t^2}+\frac{1}{t^3}$	-317243.90221776
10_{120}		yes	$-t^5+4t^4-8t^3+12t^2-15t+17$ $-\frac{15}{t}+\frac{12}{t^2}-\frac{8}{t^3}+\frac{4}{t^4}-\frac{1}{t^5}$	0

10_{121}		no	$-2t^6+5t^5-7t^4+9t^3-9t^2$ $8t-6+\frac{4}{t}-\frac{1}{t^2}$	12242.73290236
10_{122}		no	$t^4-4t^3+8t^2-12t+15-\frac{15}{t}$ $\frac{15}{t^2}-\frac{11}{t^3}+\frac{7}{t^4}-\frac{4}{t^5}+\frac{1}{t^6}$	1068.30102458
10_{123}		no	$-t^8+4t^7-9t^6+13t^5-16t^4+18t^3$ $-16t^2+13t-8+\frac{4}{t}-\frac{1}{t^2}$	107526.52229254
10_{124}		no	$-2t^3+5t^2-6t+8-\frac{8}{t}$ $\frac{7}{t^2}-\frac{5}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	-48.23433465
10_{125}		no	$1-\frac{2}{t}+\frac{3}{t^2}-\frac{3}{t^3}+\frac{4}{t^4}-\frac{3}{t^5}+\frac{3}{t^6}-\frac{2}{t^7}$	5539.77920802
10_{126}		no	$-t^{11}+t^{10}-t^9+t^8-t^7+t^6+t^3$	830589.83699493
10_{127}		yes	$-t^5+4t^4-9t^3+14t^2-17t+19$ $-\frac{17}{t}+\frac{14}{t^2}-\frac{9}{t^3}+\frac{4}{t^4}-\frac{1}{t^5}$	0
10_{128}		yes	$t^4-4t^3+6t^2-7t+9$ $-\frac{7}{t}+\frac{6}{t^2}-\frac{4}{t^3}+\frac{1}{t^4}$	0
10_{129}		no	$2t^5-4t^4+4t^3-5t^2+5t-3+\frac{3}{t}-\frac{1}{t^2}$	9904.81245175
10_{130}		no	$t^{12}-4t^{11}+8t^{10}-13t^9+16t^8-18t^7$ $17t^6-13t^5+10t^4-4t^3+t^2$	435874.62943978
10_{131}		no	$t^9-3t^8+4t^7-6t^6+7t^5$ $-6t^4+6t^3-4t^2+2t$	-1410.80172570
10_{132}		no	$t^4-4t^3+9t^2-13t+16-\frac{17}{t}$ $\frac{16}{t^2}-\frac{12}{t^3}+\frac{8}{t^4}-\frac{4}{t^5}+\frac{1}{t^6}$	13832.20861859

10_{133}		no	$-t^{12}+t^{11}-t^{10}+t^9-t^8+t^6+t^4$	49775906.30363440
10_{134}		no	$-t+2-\frac{3}{t}+\frac{6}{t^2}-\frac{6}{t^3}+\frac{7}{t^4}$ $-\frac{7}{t^5}+\frac{6}{t^6}-\frac{4}{t^7}+\frac{2}{t^8}-\frac{1}{t^9}$	-8315387.60231555
10_{135}		no	$-t^9+2t^8-4t^7+5t^6-6t^5+7t^4$ $-5t^3+5t^2-3t+2-\frac{1}{t}$	8927952.40847957
10_{136}		no	$t^3-2t^2+4t-6+\frac{8}{t}-\frac{8}{t^2}$ $\frac{8}{t^3}-\frac{7}{t^4}+\frac{4}{t^5}-\frac{2}{t^6}+\frac{1}{t^7}$	2974846.27813988
10_{137}		no	$-t^5+2t^4-4t^3+6t^2-7t+9$ $-\frac{7}{t}+\frac{6}{t^2}-\frac{4}{t^3}+\frac{2}{t^4}-\frac{1}{t^5}$	-88903.14489099
10_{138}		no	$t^{13}-2t^{12}+2t^{11}-3t^{10}+2t^9$ $-2t^8+t^7+t^6+t^4$	41715651.68254764
10_{139}		yes	$-t^5+2t^4-5t^3+8t^2-9t+11$ $-\frac{9}{t}+\frac{8}{t^2}-\frac{5}{t^3}+\frac{2}{t^4}-\frac{1}{t^5}$	0
10_{140}		no	$t^{11}-2t^{10}+3t^9-4t^8+4t^7-5t^6$ $4t^5-3t^4+3t^3-t^2+t$	-4999161.00125711
10_{141}		no	$t^8-2t^7+3t^6-4t^5+5t^4-5t^3$ $4t^2-4t+3-\frac{1}{t}+\frac{1}{t^2}$	-656179.17198256
10_{142}		no	$-t^8+4t^7-8t^6+11t^5-14t^4+15t^3$ $-13t^2+11t-6+\frac{3}{t}-\frac{1}{t^2}$	133217.96233487
10_{143}		no	$t^4-3t^3+6t^2-10t+13-\frac{13}{t}$ $\frac{13}{t^2}-\frac{10}{t^3}+\frac{7}{t^4}-\frac{4}{t^5}+\frac{1}{t^6}$	9990.89861928
10_{144}		no	$-t^8+3t^7-7t^6+12t^5-15t^4+17t^3$ $-16t^2+13t-9+\frac{5}{t}-\frac{1}{t^2}$	1043.65630300

10_{145}		yes	$-t^5+4t^4-8t^3+13t^2-16t+17$ $-\frac{16}{t}+\frac{13}{t^2}-\frac{8}{t^3}+\frac{4}{t^4}-\frac{1}{t^5}$	0
10_{146}		yes	$t^4-2t^3+4t^2-5t+5$ $-\frac{5}{t}+\frac{4}{t^2}-\frac{2}{t^3}+\frac{1}{t^4}$	0
10_{147}		no	$t^6-3t^5+7t^4-11t^3+14t^2-16t$ $15-\frac{12}{t}+\frac{9}{t^2}-\frac{4}{t^3}+\frac{1}{t^4}$	-854.81522016
10_{148}		no	$t^7-4t^6+8t^5-12t^4+15t^3-15t^2$ $14t-11+\frac{7}{t}-\frac{3}{t^2}+\frac{1}{t^3}$	3312.77867364
10_{149}		no	$-t^5+4t^4-8t^3+12t^2-15t+16$ $-\frac{14}{t}+\frac{12}{t^2}-\frac{7}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	3426.75630688
10_{150}		no	$-t^2+4t-8+\frac{12}{t}-\frac{14}{t^2}+\frac{16}{t^3}$ $-\frac{14}{t^4}+\frac{11}{t^5}-\frac{7}{t^6}+\frac{3}{t^7}-\frac{1}{t^8}$	-195738.37869393
10_{151}		no	$t^{10}-4t^9+8t^8-12t^7+14t^6-15t^5$ $14t^4-10t^3+7t^2-3t+1$	-31000.97863891
10_{152}		no	$t-3+\frac{7}{t}-\frac{11}{t^2}+\frac{14}{t^3}-\frac{14}{t^4}$ $\frac{14}{t^5}-\frac{11}{t^6}+\frac{7}{t^7}-\frac{4}{t^8}+\frac{1}{t^9}$	5233.85786013
10_{153}		no	$t^{12}-4t^{11}+7t^{10}-11t^9+13t^8-14t^7$ $14t^6-10t^5+7t^4-3t^3+t^2$	393249.33327092
10_{154}		no	$t^6-5t^5+9t^4-13t^3+17t^2-17t$ $17-\frac{13}{t}+\frac{8}{t^2}-\frac{4}{t^3}+\frac{1}{t^4}$	420.07456136
10_{155}		no	$-t^8+4t^7-9t^6+14t^5-18t^4+20t^3$ $-18t^2+15t-10+\frac{5}{t}-\frac{1}{t^2}$	58348.30072668
10_{156}		no	$t^7-3t^6+5t^5-8t^4+10t^3-10t^2$ $10t-7+\frac{5}{t}-\frac{3}{t^2}+\frac{1}{t^3}$	-844533.42467051

10_{157}		no	$t^{10} - 4t^9 + 6t^8 - 8t^7 + 9t^6$ $-8t^5 + 7t^4 - 4t^3 + 2t^2$	-36535.75617674
10_{158}		no	$t^6 - 3t^5 + 6t^4 - 9t^3 + 11t^2 - 12t$ $12 - \frac{9}{t} + \frac{6}{t^2} - \frac{3}{t^3} + \frac{1}{t^4}$	-9276.57719321
10_{159}		no	$2t - 3 + \frac{5}{t} - \frac{6}{t^2} + \frac{6}{t^3}$ $-\frac{6}{t^4} + \frac{4}{t^5} - \frac{2}{t^6} + \frac{1}{t^7}$	10500.62595380
10_{160}		no	$t^4 - 3t^3 + 6t^2 - 7t + 8$ $-\frac{8}{t} + \frac{6}{t^2} - \frac{4}{t^3} + \frac{2}{t^4}$	-1411.07755875
10_{161}		no	$-t^9 + 3t^8 - 5t^7 + 7t^6 - 9t^5 + 9t^4$ $-8t^3 + 7t^2 - 4t + 3 - \frac{1}{t}$	2795268.68653158
10_{162}		no	$-t^4 + 3t^3 - 5t^2 + 8t - 9 + \frac{10}{t}$ $-\frac{10}{t^2} + \frac{8}{t^3} - \frac{5}{t^4} + \frac{3}{t^5} - \frac{1}{t^6}$	-74314.70001592
10_{163}		no	$-t^9 + 3t^8 - 6t^7 + 8t^6 - 10t^5 + 11t^4$ $-9t^3 + 8t^2 - 5t + 3 - \frac{1}{t}$	4069344.11746252
10_{164}		no	$-t^4 + 3t^3 - 5t^2 + 8t - 10 + \frac{11}{t}$ $-\frac{10}{t^2} + \frac{9}{t^3} - \frac{6}{t^4} + \frac{3}{t^5} - \frac{1}{t^6}$	-94926.04260895
10_{165}		no	$t^7 - 3t^6 + 7t^5 - 11t^4 + 13t^3 - 14t^2$ $13t - 10 + \frac{7}{t} - \frac{3}{t^2} + \frac{1}{t^3}$	4724.36860718
10_{166}		no	$t^6 - 3t^5 + 6t^4 - 9t^3 + 12t^2 - 13t$ $12 - \frac{10}{t} + \frac{7}{t^2} - \frac{3}{t^3} + \frac{1}{t^4}$	-10188.05601842
10_{167}		no	$t^{10} - 3t^9 + 7t^8 - 11t^7 + 12t^6 - 14t^5$ $13t^4 - 9t^3 + 7t^2 - 3t + 1$	-108376.83747253
10_{168}		no	$t^{10} - 3t^9 + 6t^8 - 10t^7 + 12t^6 - 13t^5$ $12t^4 - 9t^3 + 7t^2 - 3t + 1$	-82197.13460272

10_{169}		yes	$-t^5+3t^4-7t^3+10t^2-12t+15$ $-\frac{12}{t}+\frac{10}{t^2}-\frac{7}{t^3}+\frac{3}{t^4}-\frac{1}{t^5}$	0

REFERENCES

[1] Charles Livingston and Allison H. Moore. Knotinfo: Table of knot invariants. https://knotinfo.math.indiana.edu.

[2] Paul Stäckel. Gauss als geometer. In *Carl Friedrich Gauss. Werke*, volume X, 2. Königlich Preußische Akademie der Wissenschaften, 1917.

[3] Peter Guthrie Tait. On knots i. *Transactions of the Royal Society of Edinburgh*, 28:145–190, 1877.

[4] Peter Guthrie Tait. On knots. ii. *Transactions of the Royal Society of Edinburgh*, 32:327–342, 1884. Presented 1883, published 1884.

[5] Peter Guthrie Tait. On knots. iii. *Transactions of the Royal Society of Edinburgh*, 32:493–506, 1885. Presented 1884, published 1885.

[6] Kenneth A. Perko Jr. On the classification of knots. *Proceedings of the American Mathematical Society*, 45:262–266, 1974.

[7] Kurt Reidemeister. Elementare begründung der knotentheorie. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 5, pages 24–32. Springer, 1927.

[8] R Jones VF and A Jones. A polynomial invariant for knots via von neumann algebras. *Bull. Am. Math. Soc., New Ser*, 12:103–111, 1985.

[9] Jim Hoste, Morwen Thistlethwaite, and Jeff Weeks. The first 1,701,936 knots. *The Mathematical Intelligencer*, 20(4):33–48, 1998.

[10] Benjamin A. Burton. The next 350 million knots. In Sergio Cabello and Danny Z. Chen, editors, *36th International Symposium on Computational Geometry (SoCG 2020)*, volume 164 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fÃijr Informatik.

[11] S Chmutov, S Duzhin, and J Mostovoy. Introduction to vassiliev knot invariants. *arXiv preprint arXiv:1103.5628*, 2011.

[12] José Ayala. Geometric constraints in link isotopy. *arXiv preprint arXiv:2506.04442*, 2025.

[13] WB Raymond Lickorish. *An introduction to knot theory*, volume 175. Springer Science & Business Media, 1997.

[14] William W. Menasco and Morwen B. Thistlethwaite. The classification of alternating links. *Annals of Mathematics*, 138(1):113–171, 1993.

[15] James W Alexander and Garland B Briggs. On types of knotted curves. *Annals of Mathematics*, 28(1/4):562–586, 1926.

[16] D Rolfsen. Knots and links. *Mathematics Lecture Series/Publish or Perish, Inc*, 1976.

[17] Louis H Kauffman. State models and the jones polynomial. *Topology*, 26(3):395–407, 1987.

[18] Dror Bar-Natan and Roland van der Veen. A very fast, very strong, topologically meaningful and fun knot invariant. Preprint, April 28, 2025. Available at https://drorbn.net/Theta, 2025.

[19] Roland van der Veen. The theta invariant. https://www.rolandvdv.nl/Theta/.

[20] Brendan D. McKay and Gunnar Brinkmann. Plantri – generator of certain types of planar graphs. https://users.cecs.anu.edu.au/~bdm/plantri/, 2025. Accessed: 2025-05-14.

[21] COS++. The combinatorial object server. http://combos.org, 2021. Accessed: 2025-05-14.

[22] Gunnar Brinkmann, Sam Greenberg, Catherine Greenhill, Brendan D McKay, Robin Thomas, and Paul Wollan. Generation of simple quadrangulations of the sphere. *Discrete mathematics*, 305(1-3):33–54, 2005.

[23] RB de Vries. Knot-table. https://github.com/RB-de-Vries/Knot-Table, 2025. Accessed: 2025-05-25.

[24] Python Software Foundation. Python Programming Language. https://www.python.org/, 2025. Accessed: 2025-05-14.

[25] Wikipedia contributors. Regular expression — wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Regular_expression, 2025. Accessed: 2025-05-14.