

THE VASSILIEV INVARIANT OF DEGREE 2 AS A TOOL TO STUDY PROTEIN STRUCTURE



Utrecht University

Master Thesis in Mathematical Sciences

Xareni Reyes Soto

Supervisors: Roland van der Veen and Gil Cavalcanti
Second reader: Lennart Meier

December 2018

Acknowledgements

The master's program has been the most challenging period in my life, and the presentation of this written work represents its successful culmination. This success is due in large part to the combination of the great love, friendship, and support of important people for me. I dedicate this thesis to my beloved partner Tim, who has been my main support in my new life in The Netherlands and who, besides everything else they have done for me, was my computer science and programming tutor during this process. I also dedicate this work to my family: Alicia, David, Citlalin, Diana, Peter, Frank and Laura who, from Mexico and The Netherlands, always showed me their unending love and support.

I want to thank my Mexican and American friends who always asked how I was doing and encouraged me to always try once more. Thanks also to the good friends I met in The Netherlands for supporting me and for making me trust that everything was going to be OK. Thanks to the Spanish Speaking Students and Staff Association of the Math Department at the UU for always making me laugh, especially during the hard times.

I want to thank Ana Ros Camacho for putting me in contact with my supervisor to work in this amazing research topic, and my supervisor Roland for always being enthusiastic and optimistic about the project.

I want to thank the Mexican National Council for Science and Technology (CONACYT) and the Utrecht Excellence Scholarship for providing me with the funding to study at Universiteit Utrecht.

Finally, I want to thank Almighty God for giving me the strength to be here today.

Contents

1	Knots and Tangles	5
1.1	Knots	5
1.2	Tangles	7
1.3	Knot invariants	9
1.3.1	Alexander polynomial	9
1.3.2	The Vassiliev invariant of degree 2, or simply v_2	11
1.3.3	Vassiliev skein relations	12
2	Generalization of the invariant v_2 of knots to the set of tangles	16
2.1	Meta-Monoids	16
2.1.1	The meta-monoid of tangles	17
2.1.2	The meta-monoid Γ -calculus	18
2.2	Generalization of Alexander polynomial to tangles	25
2.3	Some extra lemmas that will be useful for Chapter 4	26
2.4	Generalization of v_2 to tangles	29
2.4.1	Implementation of the stitching rule for v_2 in Python	30
3	PDB files into tangles	31
3.1	Proteins and Protein Data Bank files	31
3.1.1	Protein Data Bank files	32
3.2	Computing v_2 of a protein	33
3.2.1	PDB files into graphs	33
3.2.2	Graphs into tangles	35
3.2.3	A general method using a suitable spanning tree from the graph of atoms in a protein	35
3.2.4	Simplified graph of a protein	38
3.2.5	Spanning tree to tangle	40
3.2.6	Example, we put the approach of section 3.2.4 into practice to compute v_2 of a tangle arising from a graph	44
4	Computation of v_2 of an example protein and results	52
4.1	Computation of v_2 for the protein 1L2Y	52
4.2	Further work and improvements to be done	52
5	Code in Python	54
5.0.1	Useful Python data structures for this chapter	56
5.1	File 1: <code>graph.py</code>	57
5.2	File 2: <code>graph_position.py</code>	60
5.3	File 3: <code>lane_ordering.py</code>	61
5.4	File 4: <code>intersection_of_lanes.py</code>	63
5.5	File 5: <code>protein_structure.py</code>	68
5.6	File 6: <code>computation_v2_c2.py</code>	71
5.7	File 7: <code>compute_v2_protein.py</code>	76
5.8	File 8: <code>draw_protein_xy.py</code>	78
5.9	File 9: <code>draw_protein.py</code>	79

A Algorithms to find Spanning Trees	82
B Extra definitions	84

Introduction

Knot theory refers to the mathematical study of the objects with the same name we encounter in daily life. Ancient civilizations used knots as decorations, to attach materials or even as part of a number system, in the case of the Incas. In the 19th century physicists started to study knots. Their aim was to create a model for the atom using knots. Their conjecture was that chemical elements were represented by knots and links. However, this attempt at modeling atoms was unsuccessful. After this, mathematicians continued the study of knots, leading to beautiful and amazing results, which have been useful to other areas of knowledge, such as chemistry, fluid dynamics and molecular biology¹.

The main goal in knot theory is to determine when two knots are *equivalent*, i.e. knot theorists wonder about the existence of a continuous deformation from one knot into another. *Knot invariants* are tools that allow us to determine this. In this thesis, we focus on a certain invariant for knots, the Vassiliev invariant of degree 2 or simply v_2 . We generalize it to objects called *tangles* to study protein structure. In Chapter 1 we provide the main definitions from knot theory that we will be using.

The invariant v_2 can be defined in terms of another invariant of knots (Alexander polynomial). We devote section 1.3 the Alexander polynomial and v_2 knot invariants. In Chapter 2 we introduce a particular algebraic structure (meta-monoids) that allows us to present a generalization of the Alexander polynomial to tangles. We use this result as a base step to provide a generalization for v_2 to tangles.

At this point we are ready to work with proteins and programming. We explain how to read information from the Protein Data Bank, translate it into a mathematical object (*graph*) from which we will derive a tangle in Chapter 3. We present the results of the project in Chapter 4. Comments on further work and possible improvements of the code are also presented in Chapter 4. We end this thesis by presenting the code written in Python to compute v_2 of a protein in Chapter 5. Extra code to find the spanning tree of a graph appears in Appendix A. Relevant definitions for certain parts of this text are presented in the Appendix B.

¹One example of the applications to knot theory to molecular biology is the study of DNA recombination mechanisms by modelling the substrate and product of an enzymatic reaction using braids and tangles.

Chapter 1

Knots and Tangles

In order to pave the way for the coming chapters, we mention here basic definitions and results of knot theory. For this chapter we assume the reader has followed a basic course in knot theory, for example using the book [Ada94].

1.1 Knots

Let X and Y be topological spaces. We first recall that a (*topological*) *embedding* is a continuous and injective map $f : X \rightarrow Y$ such that f induces a homeomorphism between X and $f(X)$.

Definition 1. A *knot* is a simple (non-self-intersecting) closed curve embedded in a 3-dimensional space. \diamond

We will usually consider a knot as an embedding of the circle S^1 into \mathbb{R}^3 . The standard embedding is called the *trivial knot* or *unknot* and we will also denote it by K_0 .

Definition 2. [CF77]. A *polygonal knot* is a knot which is the union of a finite number of line segments called *edges* with endpoints called *vertices* of the knot. \diamond

Two knots K_1 and K_2 are said to be *equivalent* if we can “smoothly” deform one into another without any self intersections. When this occurs we write $K_1 \cong K_2$. More formally, K_1 and K_2 are equivalent if there is an *ambient isotopy* between them, as in the next definition.

Definition 3. Let M and N be (topological) manifolds, and $g, h : N \rightarrow M$ be embeddings. Let $F : M \times [0, 1] \rightarrow M$ be a continuous map such that $F(x, 0) = \text{Id}$, $F(x, t)$ is a homeomorphism of M to itself and $F(x, 1) \circ g = h$, then F is called an *ambient isotopy taking g to h* . \diamond

We can think of an ambient isotopy as a continuous deformation of an ambient space, for example, a deformation of a submanifold to another. A knot is called *tame* if it is equivalent to a polygonal knot, see figure 1.1. A knot which is not tame is called *wild*, see figure 1.2. All the theory in this thesis will refer to tame knots without mentioning the word “tame”.

A *knot diagram* is a representation of a knot in a plane. We can imagine this process as projecting the shadow of the knot onto a plane. Let $P : \mathbb{R}^3 \rightarrow \mathbb{R}^2$ be a projection map. Let K be a knot. A point $p \in P(K)$ is called a *multiple point* if $P^{-1}(p)$ contains more than one point of K . The *order* of $p \in P(K)$ is the cardinality of $P^{-1}(p) \cap K$. Then, a *double point* is a multiple point of order 2.

For a knot diagram, we allow a maximum of two points with the same projection. We will indicate which of these two points sits above the other by drawing a continuous line, and for the other we will draw a disrupted line. When we project *double points* (points with two preimages under the projection) we get a *crossing*, see figure 1.3. We assign a sign to a crossing as indicated in figure 1.4.

For two given knots K_1 and K_2 we see that the definition of equivalence we have, requires us to work with explicit isotopies in order to determine whether $K_1 \cong K_2$ holds. However, it is possible to work only with their diagrams, as the next theorem states.

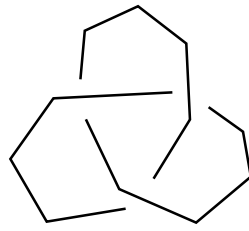


Figure 1.1: Diagram of a polygonal knot.

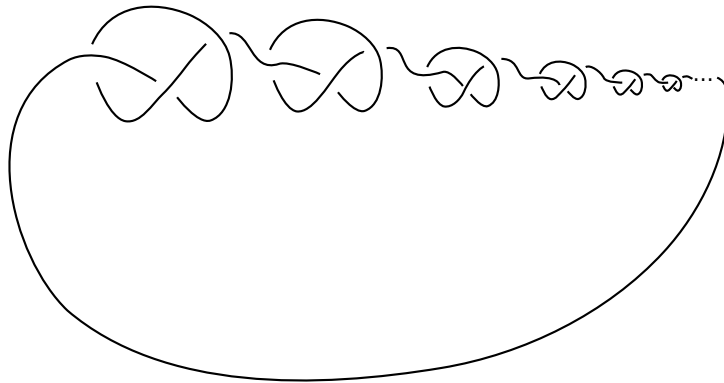


Figure 1.2: Example of a wild knot, from [Liv93].

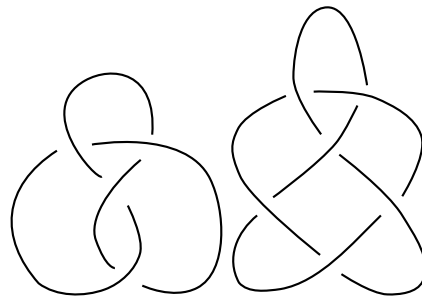


Figure 1.3: The diagram of a knot should indicate which are the over-crossings and which are the under-crossings. The first knot is known as *eight knot*, and the second is the knot 6_4 .

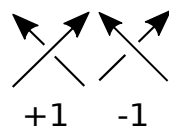


Figure 1.4: Signs for crossings.

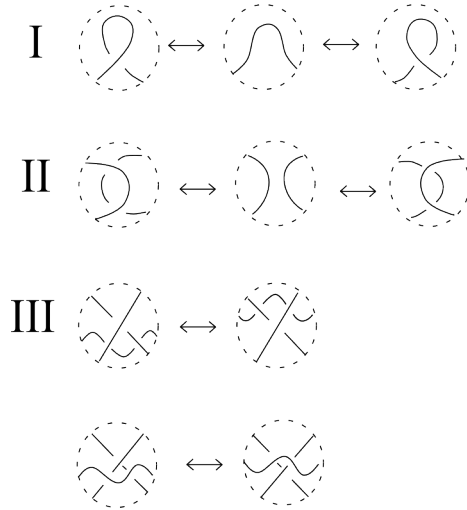


Figure 1.5: Reidemeister moves $R1$ (line I), $R2$ (line II) and $R3$ (line III).

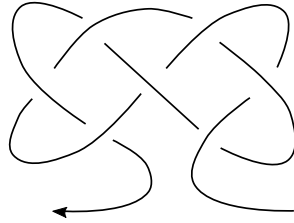


Figure 1.6: A diagram to represent the long knot 7_4 .

Theorem 4. (*Reidemeister*) 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 three types depicted in figure 1.5¹.

Since a knot K is a simple closed curve, we can assign an orientation to the curve, and this will be called the *orientation* of K . We will indicate it by an arrow on the curve. Any knot has two possible orientations.

Definition 5. A *link* is a finite collection of knots that do not intersect each other. Each of these knots is called a *component* of the link. \diamond

We will use the next definition, which comes from [GPV00], in chapter 2 :

Definition 6. A *long knot* is a smooth embedding of \mathbb{R} into \mathbb{R}^3 which coincides with the standard embedding, i.e. the embedding of \mathbb{R} into \mathbb{R}^3 as the x -axis, outside a compact set. Two long knots are equivalent if one can be transformed into the other using Reidemeister moves $R2$ and $R3$. \diamond

Intuitively a long knot consists of a curve of which a single interval is knotted and the rest stretches to infinity.

As we did with knots, we can also project a long knot into a plane to obtain a long knot diagram, by indicating the over and under crossings. A long knot diagram looks like figure 1.6.

1.2 Tangles

Let $X = \{c_1, c_2, \dots, c_n\}$ be a finite set of different labels and let D be an open disk around the origin in \mathbb{R}^2 .

¹Following the common practice in knot theory, the pictures in figure 1.5 represent knot diagrams that are identical except for the regions inside the circles.

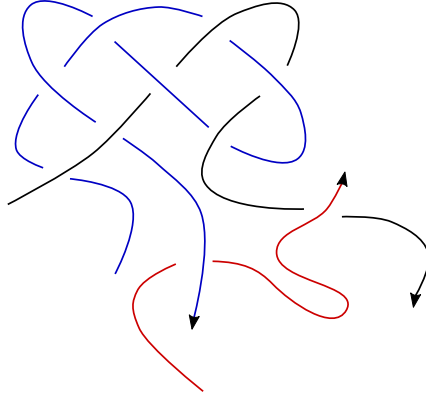


Figure 1.7: A tangle diagram with three components shown in black, red and blue.

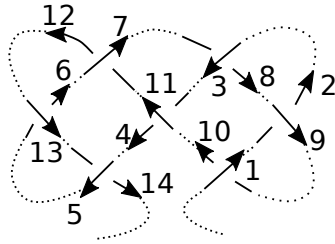


Figure 1.8: The long knot 7_4 seen as a tangle

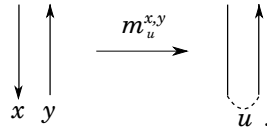
Definition 7. A *tangle diagram* labeled by X is a smooth immersion of n oriented intervals $\{I_1, I_2, \dots, I_n\}$ into D in which the only double points allowed are positive crossings and negative crossings. We call each component of the crossing a *strand*. We call *head* where the string starts and *tail* where it finishes. \diamond

Definition 8. A *tangle* is an equivalence class of tangle diagrams modulo labeling and modulo the equivalence generated by Reidemeister moves $R2$ and $R3$. We write \mathcal{T}_X for the set of tangles with labels in X . \diamond

We do not impose the first Reidemeister move $R1$, so a long knot is just a tangle with one component.

In simpler words, a (usual) tangle diagram is a collection of crossings with a certain number of loose ends, like in figure 1.7.

There are five operations defined in \mathcal{T} : two of them are *disjoint union* and *stitching*. The *disjoint union* is a binary operation $\sqcup : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{T}$ that gives a tangle consisting of the two tangles as separate components. *Stitching* is the map $m_u^{x,y} : \mathcal{T} \rightarrow \mathcal{T}$ that deletes strand x and y and creates a new strand u that starts where x starts and ends where y ends:



Stitching preserves the crossings of x and y .

We also define the operations of *identity* $e_x : \mathcal{T} \rightarrow \mathcal{T}$ which adds a string labeled x that does not cross any of the original strands, *deleting* $\eta_x : \mathcal{T} \rightarrow \mathcal{T}$ that removes the strand x and *renaming* $\sigma_b^a : \mathcal{T} \rightarrow \mathcal{T}$ which assigns to strand a the label b .

When we have a tangle diagram, we can construct it, starting with a disjoint union of crossings that we denote by $R_{i,j}^\pm$ for the crossing in which the strand i goes over and the strand j goes under, with sign as indicated. For example, we write $R_{a,b}^+$ for the crossing $\begin{array}{c} \nearrow \\ \searrow \end{array}$ and $R_{b,a}^-$ for the crossing $\begin{array}{c} \searrow \\ \nearrow \end{array}$. Then, we obtain the original tangle by considering a sequence of stitchings. For example, we can decompose the long knot from figure 1.7 in a set consisting of seven crossings, as shown in figure 1.8:

$$R_{1,10}^+ R_{9,2}^+ R_{3,8}^+ R_{11,4}^+ R_{5,14}^+ R_{13,6}^+ R_{7,12}^+.$$

If we want to obtain the original long knot, we need to stitch strand 1 to 2, strand 2 to 3 until we reach strand 13 and we stitch it to strand 14. Therefore:

$$R_{1,10}^+ R_{9,2}^+ R_{3,8}^+ R_{11,4}^+ R_{5,14}^+ R_{13,6}^+ R_{7,12}^+ // m_1^{1,2} // m_1^{1,3} // m_1^{1,4} // m_1^{1,5} // m_1^{1,6} // \dots // m_1^{1,14}.$$

This remark leads us to the following proposition.

Proposition 9. [Vo18]. *Every tangle can be obtained from a disjoint union of positive crossings and a sequence of stitching operations.*

1.3 Knot invariants

We already mentioned that the main goal of knot theory is to determine when two knots are equivalent. For this purpose, a lot of tools have been developed. An *invariant of knots* is a property of a knot that remains unchanged under ambient isotopy. This means that two equivalent knots will share this property.

Polynomial knot invariants take values in the ring of polynomials or also Laurent polynomials. They are of prime importance in knot theory [CDM12]. In the present chapter we will mention the two relevant knot invariants for this thesis. Namely, the Alexander polynomial invariant and the Vassiliev invariant of degree 2, or just v_2 .

1.3.1 Alexander polynomial

The Alexander polynomial of a knot K , denoted by $\Delta(K)$ was the first knot polynomial to be discovered. J.W. Alexander presented it in the paper *Topological invariants of knots and links* that was published in 1928. The approach presented in this paper is a combinatorial formulation of the Alexander polynomial. However, the calculation of this polynomial can also be done geometrically, using constructions called Seifert surfaces or algebraically, by considering the group of a knot and Fox derivatives [Lon05]. All these approaches are explained in the book *Knot Theory* by C. Livingston ([Liv93]). The derivation of the polynomial we present here is a skein relation discovered by J. H. Conway in 1969 and represents a very easy way to introduce the Alexander polynomial.

Conway showed that the Alexander polynomial of a knot can be computed using two rules: let L be a projection of a knot K and let K_0 denote the trivial knot. We take three projections of links L_+ , L_- , L_0 such that they are identical except in the region shown, as in figure 1.9. The Alexander polynomial is defined by the following rules:

1. $\Delta(K_0) = 1$.
2. $\Delta(L_+) - \Delta(L_-) + (t^{\frac{1}{2}} - t^{-\frac{1}{2}})\Delta(L_0) = 0$.

This is an example of a *skein relation*: an equation on the values of a function in terms of knot diagrams that only differ from each other near a crossing [CDM12].

Now we provide an example of the computation of the Alexander polynomial of a knot.

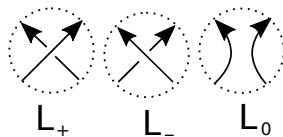


Figure 1.9: Knot projection regions for the Alexander polynomial.

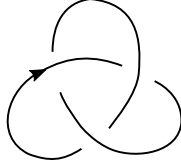
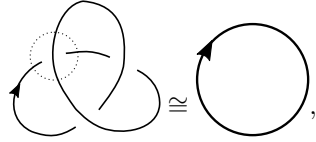


Figure 1.10: (Diagram of the) Right-handed trefoil knot.

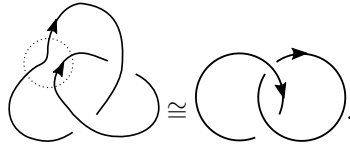
Example 10. Let K be a projection of a knot with orientation, we pick a crossing. This crossing is one of L_+ or L_- . Then we split the two strings of this crossing and we “glue” them together in such a way that the orientation is preserved. We do this for the right-handed trefoil knot, shown in figure 1.10:

$$\Delta \left(\text{trefoil} \right) - \Delta \left(\text{trefoil with one crossing split} \right) + (t^{\frac{1}{2}} - t^{-\frac{1}{2}}) \Delta \left(\text{trefoil with one crossing split and strands crossed} \right) = 0.$$

We note that



and



Therefore

$$\Delta \left(\text{trefoil} \right) = \Delta \left(\text{circle} \right) - (t^{\frac{1}{2}} - t^{-\frac{1}{2}}) \cdot \Delta \left(\text{two linked circles} \right).$$

It is straightforward to verify that

$$\Delta \left(\text{two linked circles} \right) = -(t^{\frac{1}{2}} - t^{-\frac{1}{2}}) \text{ since } \Delta \left(\text{two separate circles} \right) = 0.$$

Therefore

$$\Delta \left(\text{trefoil} \right) = 1 + (t^{\frac{1}{2}} - t^{-\frac{1}{2}})(t^{\frac{1}{2}} - t^{-\frac{1}{2}}) = t - 1 + t^{-1}. \tag{1.1}$$

◇

The Alexander polynomial of a diagram of a knot is unique up to multiplication by a factor of $\pm t^k$, for some integer k , as in theorem 11².

Theorem 11. ([Liv93]). *If the Alexander polynomial for a knot is computed using two different sets of choices for diagrams and labelings, the two polynomials will differ by a multiple of $\pm t^k$, for some integer k .*

²The proof of this result uses knowledge of the effects of performing Reidemeister moves on the *Alexander matrix* and the effects of a choice of orientation on the same matrix.

Given this fact, we use a *normalization* of this polynomial to make it unique. We take the *normalized* Alexander polynomial as the one containing no negative powers of t and a positive constant term [CF77]. For two polynomials p and q we write $p \equiv q$ to denote they are equal up to multiplication by $\pm t^n$, with $n \in \mathbb{Z}$.

Hence, if we multiply the polynomial in equation (1.1) by t we get the normalized polynomial for the trefoil knot:

$$t - 1 + t^{-1} \equiv t^2 - t + 1.$$

Very similarly to the Alexander polynomial, the *Conway polynomial* of a knot K , denoted by $C(z)$ is defined by:

1. $C(K_0) = 1$.
2. $C(L_+) - C(L_-) = zC(L_0)$.

Thus, by making the substitution $z \mapsto t^{-\frac{1}{2}} - t^{\frac{1}{2}}$ in the Conway polynomial, we get the Alexander polynomial. One important property of the Conway polynomial is the following: for every n , the coefficient a_n of z^n in the Conway polynomial is a numerical invariant of the knot K , see page 46 from [CDM12].

1.3.2 The Vassiliev invariant of degree 2, or simply v_2

The notion of *finite type knot invariants* was introduced by V. Vassiliev at the end of the 1980's in his paper [Vas90]. V.I. Arnold introduced the name "Vassiliev invariants" for them. As we mentioned in the introduction, v_2 is another name for the finite type knot invariant of degree 2. Vassiliev defined the finite type invariants from a very technical point of view: knots are embeddings of the circle S^1 into \mathbb{R}^3 . The space of these embeddings form a topological space where knot invariants can be regarded as locally constant functions. The space of knots A is an open subspace of the space X of all smooth maps from S^1 to \mathbb{R}^3 . The complement of A , $\Sigma = X \setminus A$, is called *discriminant* and consists of all maps that are not embeddings. Vassiliev constructed a spectral sequence for the homology of Σ . Then, he used the Alexander duality theorem to produce cohomology classes for A . The zeroth cohomology class is the set of *finite type knot invariants*.

Three years after Vassiliev published his work on finite type knot invariants, J. Birman and X.-S. Lin simplified this exposition in the paper [BL93]. In this paper they explain how the Jones polynomial (see Appendix) and finite type knot invariants are related, and they also study the importance of the algebra of *chord diagrams*. A chord diagram consists of an oriented circle with a finite number of chords marked on it. An example of this kind of diagram is a *Gauss diagram*.

The Gauss diagram of a knot consists of a circle with information about the preimages of double points. For the creation of a Gauss diagram we follow the next steps. First, we give an orientation to a knot and we enumerate the crossings. We choose a point p in the diagram that is not a double point. Second, we "travel around the knot" starting at the point p and we record the information about the crossings. We place the information of the embedding in a circle (with no crossings). More explicitly, we start at point p (denoted by a dot \bullet) in the knot diagram and we follow the orientation of the diagram. Each time we find a crossing, we indicate if we go over this crossing (we will denote it by \mathcal{O}_k , where k is the index of the crossing) or under (we will denote it by \mathcal{U}_k , where k is the index of the crossing). We do this until we return to the starting point p . As a result, we get a sequence of \mathcal{O}_i 's and \mathcal{U}_j 's. Next, we place each element of this sequence at points around the circle and we connect \mathcal{O}_i and \mathcal{U}_i with an arrow that goes from \mathcal{O}_i to \mathcal{U}_i . Sometimes, it is useful to write the sign of the crossing at each \mathcal{O}_i .

Example 12. Let us look at the trefoil knot from figure 1.11. We have the sequence:

$$\mathcal{U}_1 - \mathcal{O}_2 - \mathcal{U}_3 - \mathcal{O}_1 - \mathcal{U}_2 - \mathcal{O}_3,$$

so we obtain the first Gauss diagram in figure 1.11. Now, let us look at the eight knot from figure 1.11. We have the sequence:

$$\mathcal{O}_1 - \mathcal{U}_2 - \mathcal{O}_3 - \mathcal{U}_4 - \mathcal{O}_2 - \mathcal{U}_1 - \mathcal{O}_4 - \mathcal{U}_3,$$

so we obtain the second Gauss diagram in figure 1.11. ◇

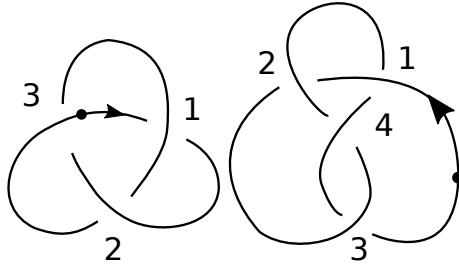


Figure 1.11: We number the crossings from a certain knot diagram to construct the corresponding Gauss diagram.

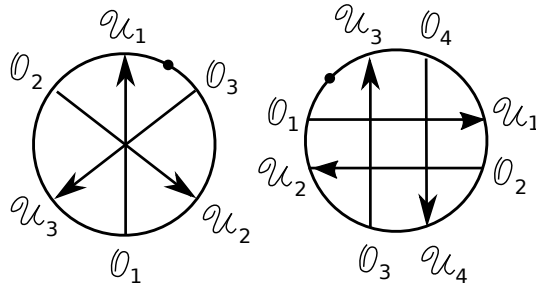


Figure 1.12: These are the Gauss diagrams for the trefoil knot and the eight knot from figure 1.12, respectively.

Coming back to the finite type knot invariants, the main theorem of the theory of Vassiliev knot invariants was proved by M. Kontsevich in [Kon93]. It states that real-valued Vassiliev invariants can be completely studied using only combinatorics of chord diagrams. The main tool to prove this was the Kontsevich integral.

The most important source about the fundamentals on Vassiliev knot and link invariants is the paper [Bar95] by D. Bar-Natan. Another interesting source is the book [CDM12] which contains an introductory explanation about these invariants, specially focusing on combinatorial results from them. In the paper [PV01] M.Polyak and O.Viro provide Gauss diagram formulae for v_2 .

1.3.3 Vassiliev skein relations

The first attempt we tried to generalize the invariant v_2 to tangles (the theoretical goal of this project) involved using the skein relation of v_2 , and that is why we include a section about it in this thesis.

A *singular knot* is a smooth map f from the circle S^1 to \mathbb{R}^3 that is not an embedding. The only singularity we will consider from now on is double points. Another way to describe them is the following: a point p is a double point if in a neighborhood of p the curve $f(S^1)$ has two branches with linearly independent tangent vectors, see Figure 1.13.

The next equation is known as the *Vassiliev skein relation* and is used to extend any knot invariant v to knots with double points:

$$v(\text{crossing}) = v(\text{smooth}) - v(\text{smooth}). \quad (1.2)$$

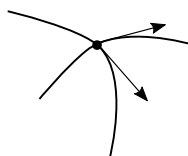


Figure 1.13: Double point.

A *Vassiliev invariant of order (of type) n* is defined as a knot invariant whose extension vanishes on knots with more than n double points [CDM12].

Example 13. The coefficient a_n of z^n in the Conway polynomial of a knot is a Vassiliev invariant of degree n . This example appears in [CDM12].

Let us first look at a knot with one double point. Combining the skein relation for the Conway polynomial:

$$C \left(\begin{array}{c} \text{---} \nearrow \text{---} \\ \text{---} \searrow \text{---} \end{array} \right) - C \left(\begin{array}{c} \text{---} \nearrow \text{---} \\ \text{---} \nearrow \text{---} \end{array} \right) - zC \left(\begin{array}{c} \text{---} \curvearrowright \text{---} \\ \text{---} \curvearrowright \text{---} \end{array} \right) = 0,$$

and taking C as our invariant v in equation (1.2) we get:

$$C \left(\begin{array}{c} \text{---} \nearrow \text{---} \\ \text{---} \searrow \text{---} \end{array} \right) = zC \left(\begin{array}{c} \text{---} \curvearrowright \text{---} \\ \text{---} \curvearrowright \text{---} \end{array} \right).$$

Now, let us look at a knot with two double points:

$$\begin{aligned} C \left(\begin{array}{c} \text{---} \nearrow \text{---} \quad \text{---} \nearrow \text{---} \\ \text{---} \searrow \text{---} \quad \text{---} \searrow \text{---} \end{array} \right) &= zC \left(\begin{array}{c} \text{---} \curvearrowright \text{---} \quad \text{---} \nearrow \text{---} \\ \text{---} \searrow \text{---} \quad \text{---} \searrow \text{---} \end{array} \right) \\ &= z^2 C \left(\begin{array}{c} \text{---} \curvearrowright \text{---} \quad \text{---} \curvearrowright \text{---} \\ \text{---} \curvearrowright \text{---} \quad \text{---} \curvearrowright \text{---} \end{array} \right). \end{aligned}$$

If we continue this process, we arrive to the equation:

$$C \left(\begin{array}{c} \text{---} \nearrow \text{---} \quad \text{---} \nearrow \text{---} \quad \cdots \quad \text{---} \nearrow \text{---} \\ \text{---} \searrow \text{---} \quad \text{---} \searrow \text{---} \quad \cdots \quad \text{---} \searrow \text{---} \end{array} \right) = z^m C \left(\begin{array}{c} \text{---} \curvearrowright \text{---} \quad \text{---} \curvearrowright \text{---} \quad \cdots \quad \text{---} \curvearrowright \text{---} \\ \text{---} \curvearrowright \text{---} \quad \text{---} \curvearrowright \text{---} \quad \cdots \quad \text{---} \curvearrowright \text{---} \end{array} \right),$$

where m is the number of double points of our knot. Hence, if $m \geq l + 1$ we get that the coefficient of t^l is zero.

As a particular case, the coefficient a_2 of z^2 in the Conway polynomial of a knot is a Vassiliev invariant of degree 2, since for a knot with three double points, the coefficient for z^2 is 0. \diamond

The invariant v_2

The invariant v_2 is the first non-trivial invariant of finite type. It is also called *Casson knot invariant*. Its skein relation is the following:

$$v_2 \left(\begin{array}{c} \text{---} \nearrow \text{---} \\ \text{---} \searrow \text{---} \end{array} \right) - v_2 \left(\begin{array}{c} \text{---} \searrow \text{---} \\ \text{---} \nearrow \text{---} \end{array} \right) = lk \left(\begin{array}{c} \text{---} \curvearrowright \text{---} \\ \text{---} \curvearrowright \text{---} \end{array} \right), \quad (1.3)$$

where $lk(L)$ denotes the linking number of link L (see Appendix B). This expression together with the condition $v_2(K_0) = 0$ defines a knot invariant that takes the value 1 in the trefoil knot.

It is shown in [CDM12] that the coefficient of z^2 in the Conway polynomial of a knot K coincides with the Casson invariant. It follows from this that the invariant v_2 can also be defined as

$$v_2(K) := \frac{1}{2} \Delta''(1),$$

where Δ denotes the normalized Alexander polynomial of the knot K . We now provide two examples of the computation of v_2 of a knot using the skein relation (1.3).

Example 14. We compute v_2 of the trefoil knot using the skein relation (1.3):

$$v_2 \left(\text{trefoil} \right) - v_2 \left(\text{trefoil} \right) = lk \left(\text{trefoil} \right).$$

This is the same as

$$\begin{aligned} v_2 \left(\text{trefoil} \right) &= v_2 \left(\text{circle} \right) + lk \left(\text{trefoil} \right) \\ &= 1, \end{aligned}$$

since

$$v_2 \left(\text{circle} \right) = 0 \text{ and } lk \left(\text{trefoil} \right) = \frac{1}{2}(1 + 1).$$

◇

Example 15. We now compute v_2 of the eight knot:


$$v_2 \left(\text{eight} \right) - v_2 \left(\text{eight} \right) = lk \left(\text{eight} \right).$$

This is the same as

$$\begin{aligned} v_2 \left(\text{eight} \right) &= 0 + lk \left(\text{eight} \right) \\ &= \frac{1}{2}(-1 - 1) = -1. \end{aligned}$$

◇

In [PV01] Polyak and Viro develop combinatorial formulas for v_2 , for example the one given in the following theorem. They claim these types of combinatorial formulae are the easiest ways to compute v_2 of a certain knot diagram.

Theorem 16. For a subdiagram isomorphic to  let c_1 and c_2 be the chords and set $\epsilon(c_i)$ its sign $\in \{-1, 1\}$. Then





$$v_2(K) = \sum \epsilon(c_1)\epsilon(c_2).$$

We also write $\sum \epsilon(c_1)\epsilon(c_2) = \langle \text{X}, G \rangle$.

We now compute v_2 of the trefoil and eight knots using theorem 16.

Example 17. Let K_1 be the trefoil knot from figure 1.11 and K_2 the eight knot from this same figure. For K_1 we have:




$$\begin{aligned} v_2(K_1) &= \langle \text{X}, K_1 \rangle \\ &= \langle \text{X}, \text{X}_{1,2} \rangle. \end{aligned}$$

The only subdiagram of  isomorphic to  is , since  has the dot at the wrong place. Then

$$\begin{aligned} v_2(K_1) &= \left\langle \text{diagram with dot at top-left}, K_1 \right\rangle \\ &= \epsilon(\text{chord 1}) \cdot \epsilon(\text{chord 2}) \\ &= (+1)(+1) = 1. \end{aligned}$$

We now do it for the eight knot K_2 :

$$\begin{aligned} v_2(K_2) &= \left\langle \text{diagram with dot at top-right}, K_2 \right\rangle \\ &= \left\langle \text{diagram with dot at top-right}, \text{diagram with four chords and dot at top-right} \right\rangle \end{aligned}$$

The only subdiagram of  isomorphic to  is . Then

$$\begin{aligned} v_2(K_2) &= \left\langle \text{diagram with dot at top-left}, K_2 \right\rangle \\ &= \epsilon(\text{chord 2}) \cdot \epsilon(\text{chord 3}) \\ &= (+1)(-1) = -1. \end{aligned}$$

◇

The attempt we used to generalize v_2 to tangles uses a skein relation and more advanced concepts of knot theory such as *braids* and *closure of braids*³. Since we did not achieve a well-defined generalization in this case we do not present these concepts in this thesis. We decided to look at an algebraic approach using a (relatively) new algebraic construction called *meta-monoid*. That will be the subject of the next chapter.

³We did not try generalizing the v_2 invariant of a knot using Gauss diagrams, but it could be an interesting exercise for the interested reader.

Chapter 2

Generalization of the invariant v_2 of knots to the set of tangles

As we mentioned in the introduction, one of the goals of this thesis is to present a generalization of some knot invariants to tangles, for example the Alexander polynomial and v_2 invariants. The first two sections in this chapter are based on the thesis [Vo18].

2.1 Meta-Monoids

The algebraic structure known as *meta-monoid* was introduced by D. Bar-Natan in the paper [Bar15], and he defined them as follows.

Definition 18. Let M be the collection of all finite subsets X of a (fixed) set Y . A *meta-monoid indexed by M* is an indexed collection of sets $\{\mathcal{G}_X\}_{X \in M}$ together with the following maps between them.

- *stitching or multiplication:*

$$m_z^{x,y} : \mathcal{G}_{\{x,y\} \cup X} \rightarrow \mathcal{G}_{\{z\} \cup X}, \text{ if } \{x,y,z\} \cap X = \emptyset \text{ and } x \neq y.$$

- *disjoint union:*

$$\sqcup : \mathcal{G}_X \times \mathcal{G}_Y \rightarrow \mathcal{G}_{X \cup Y}, \text{ if } X \cap Y = \emptyset.$$

- *identity:*

$$e_x : \mathcal{G}_X \rightarrow \mathcal{G}_{\{x\} \cup X}, \text{ if } x \notin X.$$

- *deletion:*

$$\eta_x : \mathcal{G}_{X \cup \{x\}} \rightarrow \mathcal{G}_X, \text{ if } x \notin X.$$

- *renaming:*

$$\sigma_z^x : \mathcal{G}_{X \cup \{x\}} \rightarrow \mathcal{G}_{X \cup \{z\}}, \text{ if } \{x,z\} \cap X = \emptyset.$$

These maps are also called *operations*. Following the notation in [Vo18], we denote composition of the operations defined above by $f // g := g \circ f$. The maps just described satisfy the axioms:

- Monoid axioms $m_u^{x,y} // m_v^{u,z} = m_u^{y,z} // m_v^{x,u}$ (meta-associativity)
 $e_a // m_c^{a,b} = \sigma_c^b$ (left-identity)
 $e_b // m_c^{a,b} = \sigma_c^a$ (right-identity)

- Miscellaneous axioms $e_a // \sigma_b^a = e_b$, $\sigma_b^a // \sigma_c^b = \sigma_c^a$, $\sigma_b^a // \sigma_a^b = \text{Id}$,
 $\sigma_b^a // \eta_b = \eta_a$, $e_a // \eta_a = \text{Id}$, $m_c^{a,b} // \eta_c = \eta_b // \eta_a$,
 $m_c^{a,b} // \sigma_d^c = m_d^{a,b}$, $\sigma_b^a // m_d^{b,c} = m_d^{a,c}$.

It is required that operations with different labels commute, for example:

$$m_c^{a,b} // m_f^{d,e} = m_f^{d,e} // m_c^{a,b}.$$

We also require that the disjoint union operation \sqcup commutes with all other operations. We will write \mathcal{G} for a meta-monoid $(\{\mathcal{G}_X\}_{X \in M}, m_c^{a,b}, e_a, \eta_a, \sigma_c^a, \sqcup)$. \diamond

Definition 19. For two meta-monoids \mathcal{G} and \mathcal{H} , a *meta-monoid homomorphism* is a collection of maps $\{f_X : \mathcal{G}_X \rightarrow \mathcal{H}_X\}_{X \in M}$, that forms a natural transformation for each operation. This means that we have a commutative diagram:

$$\begin{array}{ccc} \mathcal{G}_X & \xrightarrow{\mu_{\mathcal{G}}} & \mathcal{G}_Z \\ f_X \downarrow & & \downarrow f_Z \\ \mathcal{H}_X & \xrightarrow{\mu_{\mathcal{H}}} & \mathcal{H}_Z \end{array},$$

where $\mu_{\mathcal{G}}$ and $\mu_{\mathcal{H}}$ denote operations in the meta-monoids \mathcal{G} and \mathcal{H} respectively. \diamond

For instance, in the case of stitching and finite sets $X = \{x, y\}$ and $Z = \{z\}$ we get the diagram:

$$\begin{array}{ccc} \mathcal{G}_{\{x,y\}} & \xrightarrow{m_z^{x,y}} & \mathcal{G}_{\{z\}} \\ f_X \downarrow & & \downarrow f_Z \\ \mathcal{H}_{\{x,y\}} & \xrightarrow{m_z^{x,y}} & \mathcal{H}_{\{z\}} \end{array}.$$

In the next two sections we present two meta-monoids that are very important to us, since they will allow us to come up with a generalization of the Alexander polynomial to tangles. In Chapter 1 we already saw the definition of tangles, the first of our important meta-monoids.

2.1.1 The meta-monoid of tangles

Lemma 20. *The set of tangles \mathcal{T} is a meta-monoid.*

Proof. We will prove this visually (as in [Vo18]). We will only prove the meta-associativity, left and right identity axioms since the proofs for the miscellaneous axioms are very similar.

For meta-associativity, we will show that $m_u^{x,y} // m_v^{u,z} = m_u^{y,z} // m_v^{x,u}$. On the one hand we have:

$$\begin{array}{ccc} \begin{array}{c} \downarrow \\ x \end{array} & \begin{array}{c} \uparrow \\ y \end{array} & \begin{array}{c} \downarrow \\ z \end{array} & \xrightarrow{m_u^{x,y}} & \begin{array}{c} \downarrow \\ u \end{array} & \begin{array}{c} \uparrow \\ z \end{array} & \xrightarrow{m_v^{u,z}} & \begin{array}{c} \downarrow \\ v \end{array} \end{array}$$

On the other hand:

$$\begin{array}{ccc} \begin{array}{c} \downarrow \\ x \end{array} & \begin{array}{c} \uparrow \\ y \end{array} & \begin{array}{c} \downarrow \\ z \end{array} & \xrightarrow{m_u^{y,z}} & \begin{array}{c} \downarrow \\ x \end{array} & \begin{array}{c} \uparrow \\ u \end{array} & \xrightarrow{m_v^{x,u}} & \begin{array}{c} \downarrow \\ v \end{array} \end{array}$$

thus establishing the meta-associativity property.

For left-identity we have:

$$\begin{array}{c}
\downarrow \\
b
\end{array}
\begin{array}{c}
\longrightarrow \\
e_a
\end{array}
\begin{array}{c}
\uparrow \\
a
\end{array}
\begin{array}{c}
\downarrow \\
b
\end{array}
\begin{array}{c}
\longrightarrow \\
m_c^{a,b}
\end{array}
\begin{array}{c}
\downarrow \\
c
\end{array}
=
\begin{array}{c}
\downarrow \\
c
\end{array}$$

i.e. $e_a // m_c^{a,b} = \sigma_c^b$.

For right-identity we have:

$$\begin{array}{c}
\uparrow \\
a
\end{array}
\begin{array}{c}
\longrightarrow \\
e_b
\end{array}
\begin{array}{c}
\uparrow \\
a
\end{array}
\begin{array}{c}
\downarrow \\
b
\end{array}
\begin{array}{c}
\longrightarrow \\
m_c^{a,b}
\end{array}
\begin{array}{c}
\downarrow \\
c
\end{array}
=
\begin{array}{c}
\uparrow \\
c
\end{array}$$

i.e. $e_b // m_c^{a,b} = \sigma_c^a$.

□

Since we only consider positive and negative crossings, in order to prove a function $f : \mathcal{T} \rightarrow \mathcal{T}$ is an invariant of tangles, we only need to prove it remains unchanged under Reidemeister moves $R2$ and $R3$. Moreover, suppose we want to define a meta-monoid homomorphism f of tangles, then we only need to establish the image of $\begin{array}{c} \nearrow \\ \searrow \end{array}$ and $\begin{array}{c} \searrow \\ \nearrow \end{array}$ under f and verify we satisfy the relations $R2$ and $R3$.

2.1.2 The meta-monoid Γ -calculus

The next important meta-monoid for this thesis is called Γ -calculus. Its relevance lies in the fact that this object will be the target space of an algebraic invariant of tangles. We define it as follows: let X be a finite set of labels, let T_X be $\mathbb{Q}(t)$, the field of rational functions in t with coefficients in \mathbb{Q} and let $M_{X \times X}(T_X)$ be the collection of $|X| \times |X|$ labeled matrices with rational functions in t as entries and elements in X as labels for rows and columns.

An element $\xi \in T_X \times M_{X \times X}(T_X)$ consists of an element ω in T_X , referred to as *scalar part* and an element A in $M_{X \times X}(T_X)$, called the *matrix part*. For example, if $X = \{x, y, z\} \cup S$, with $S \cap \{x, y, z\} = \emptyset$ then ξ can be written as:

$$\xi = \left(\begin{array}{c|cccc} \omega & x & y & z & S \\ \hline x & \alpha & \beta & \zeta & \theta \\ y & \gamma & \delta & \eta & \epsilon \\ z & \iota & \kappa & \lambda & \mu \\ S & \phi & \psi & \nu & \Xi \end{array} \right).$$

On ξ we can define substitution of t by 1. We will denote it by $t \rightarrow 1$.

Let us consider the subset Γ_X of $T_X \times M_{X \times X}(T_X)$ such that for every $x \in \Gamma_X$ with scalar part ω and matrix part M (recall that X is a finite set of labels) we have the following:

$$\left(\begin{array}{c|X} \omega & X \\ \hline X & M \end{array} \right)_{t \rightarrow 1} = \left(\begin{array}{c|X} 1 & X \\ \hline X & \text{Id} \end{array} \right).$$

In the previous line Id denotes the identity matrix.

In Γ_X we define the following operations:

- identity:

$$e_c : \Gamma_S \rightarrow \Gamma_{S \cup \{c\}}, \left(\begin{array}{c|S} \omega & S \\ \hline S & M \end{array} \right) // e_c = \left(\begin{array}{c|cS} \omega & c & S \\ \hline c & 1 & \mathbf{0} \\ S & \mathbf{0} & M \end{array} \right), \text{ where } c \notin S.$$

- disjoint union:

$$\sqcup : \Gamma_{X_1} \times \Gamma_{X_2} \rightarrow \Gamma_{X_1 \cup X_2},$$

$$\left(\frac{\omega_1}{X_1} \middle| \frac{X_1}{M_1} \right) \sqcup \left(\frac{\omega_2}{X_2} \middle| \frac{X_2}{M_2} \right) = \left(\frac{\omega_1 \omega_2}{X_1} \middle| \begin{array}{c|c} X_1 & X_2 \\ \hline M_1 & \mathbf{0} \\ X_2 & \mathbf{0} & M_2 \end{array} \right), \text{ where } X_1 \cap X_2 = \emptyset.$$

- renaming:

$$\sigma_b^a : \Gamma_{S \cup \{a\}} \rightarrow \Gamma_{S \cup \{b\}}, \left(\frac{\omega}{S} \middle| \begin{array}{c|c} a & S \\ \hline a & \alpha & \theta \\ S & \phi & \Xi \end{array} \right) // \sigma_b^a = \left(\frac{\omega}{S} \middle| \begin{array}{c|c} b & S \\ \hline b & \alpha & \theta \\ S & \phi & \Xi \end{array} \right), \text{ where } \{a, b\} \cap S = \emptyset.$$

- deletion:

$$\eta_a : \Gamma_{S \cup \{a\}} \rightarrow \Gamma_S, \left(\frac{\omega}{S} \middle| \begin{array}{c|c} a & S \\ \hline a & \alpha & \theta \\ S & \phi & \Xi \end{array} \right) // \eta_a = \left(\frac{\omega}{S} \middle| \frac{S}{\Xi} \right)_{t \rightarrow 1}, \text{ where } a \notin S.$$

- stitching:

$$m_c^{a,b} : \Gamma_{\{a,b\} \cup S} \rightarrow \Gamma_{\{c\} \cup S},$$

$$\left(\frac{\omega}{S} \middle| \begin{array}{c|c|c} a & b & S \\ \hline a & \alpha & \beta & \theta \\ b & \gamma & \delta & \epsilon \\ S & \phi & \psi & \Xi \end{array} \right) // m_c^{a,b} = \left(\frac{(1-\beta)\omega}{S} \middle| \begin{array}{c|c} c & S \\ \hline \gamma + \frac{\alpha\delta}{1-\beta} & \epsilon + \frac{\delta\theta}{1-\beta} \\ \phi + \frac{\alpha\psi}{1-\beta} & \Xi + \frac{\psi\theta}{1-\beta} \end{array} \right), \quad (2.1)$$

where $\{a, b, c\} \cap S = \emptyset$ and $a \neq b$.

In the previous lines $\mathbf{0}$ denotes a matrix consisting of zeros whose size depends on the context.

Lemma 21. *The operation of stitching is well-defined, i.e. β is never equal to 1 and the image is contained in $\Gamma_{\{c\} \cup S}$.*

Proof. Assume for the sake of contradiction that $\beta = 1$. Then $1 = \beta = R(t)_{t \rightarrow 1} = \frac{P(t)}{Q(t)}_{t \rightarrow 1} = 0$ since β sits outside the diagonal, so we find a contradiction.

We also need to show that the right-hand side of (2.1) is an element of $\Gamma_{\{c\} \cup S}$. Since $\xi \in \Gamma_{\{a,b\} \cup S}$, we have after substituting $t \rightarrow 1$ that $\alpha_{t \rightarrow 1} = 1$, $\beta_{t \rightarrow 1} = 0$, $\gamma_{t \rightarrow 1} = 0$, $\delta_{t \rightarrow 1} = 1$, $\theta_{t \rightarrow 1} = \mathbf{0}$, $\epsilon_{t \rightarrow 1} = \mathbf{0}$, $\phi_{t \rightarrow 1} = \mathbf{0}$, $\psi_{t \rightarrow 1} = \mathbf{0}$ and $\Xi_{t \rightarrow 1} = \text{Id}$. Hence $\gamma + \frac{\alpha\delta}{1-\beta} = 1$, $\epsilon + \frac{\delta\theta}{1-\beta} = \mathbf{0}$, $\phi + \frac{\alpha\psi}{1-\beta} = \mathbf{0}$ and $\Xi + \frac{\psi\theta}{1-\beta} = \text{Id}$.

Since $\omega_{t \rightarrow 1} = 1$ and $\beta_{t \rightarrow 1} = 0$, it follows $((1-\beta)\omega)_{t \rightarrow 1} = 1$. \square

Now we introduce some notation and conditions for a sequence of stitching operations from [Vo18]. Let $\xi \in \Gamma_X$,

$$\xi = \left(\frac{\omega}{X} \middle| \frac{X}{A} \right).$$

Suppose that we want to perform the sequence of stitchings

$$m_{c_1}^{a_1, b_1} // m_{c_2}^{a_2, b_2} // \dots // m_{c_k}^{a_k, b_k}, \text{ for } a_i, b_j, c_k \in X. \quad (2.2)$$

Setting $\mathbf{a} = (a_1, a_2, \dots, a_k)$, $\mathbf{b} = (b_1, b_2, \dots, b_k)$ and $\mathbf{c} = (c_1, c_2, \dots, c_k)$ we denote expression (2.2) more compactly as

$$m_{\mathbf{c}}^{\mathbf{a}, \mathbf{b}}.$$

The vectors \mathbf{a} , \mathbf{b} and \mathbf{c} should satisfy some conditions that come from conditions on stitching. First, $a_i \neq a_j$, $b_i \neq b_j$ and $a_i \neq b_i$ for $i, j \in \{1, 2, \dots, k\}$. Second, for a subset $I \subset \{1, 2, \dots, k\}$, we cannot have (b_i, \dots, b_j) is a permutation of (a_i, \dots, a_j) where $i, \dots, j \in I$. The second condition prevents us from stitching a component to itself.

Having said this, we can wonder: will the order in which we do a composition of stitchings matter? Fortunately for us, the answer is “no”. The following result appears in [Vo18] page 20. It becomes very relevant when we work with examples, since by using this result we can stitch elements in any order convenient for us.

Proposition 22. *Assume that we will perform a sequence of stitching operations*

$$m_{\mathbf{c}}^{\mathbf{a}, \mathbf{b}} = m_{c_1}^{a_1, b_1} // m_{c_2}^{a_2, b_2} // \dots // m_{c_k}^{a_k, b_k}.$$

Then a permutation of the stitching operations does not change the result, in other words, the order in which we perform the stitchings does not matter.

The proof of this result, which appears in [Vo18], uses a formula called *Stitching in Bulk* introduced in Proposition 3.1 from the same source.

The following result shows that Γ -calculus is indeed a meta-monoid.

Lemma 23. *For a set X of labels, Γ_X is a meta-monoid.*

Proof. In order to see that Γ_X is a meta-monoid, we need to check it satisfies the monoid axioms (meta-associativity, left-identity, right-identity and miscellaneous axioms). We will only show the meta-associativity and left-identity properties since the others follow similarly.

For meta-associativity, let

$$\xi = \left(\begin{array}{c|cccc} \omega & 1 & 2 & 3 & S \\ \hline 1 & \alpha_{11} & \alpha_{12} & \alpha_{13} & \theta_1 \\ 2 & \alpha_{21} & \alpha_{22} & \alpha_{23} & \theta_2 \\ 3 & \alpha_{31} & \alpha_{32} & \alpha_{33} & \theta_3 \\ S & \psi_1 & \psi_2 & \psi_3 & \Xi \end{array} \right).$$

We are going to show that

$$\xi // m_1^{1,2} // m_1^{1,3} = \xi // m_2^{2,3} // m_1^{1,2}. \quad (2.3)$$

We use the program in Mathematica written by H. Vo that appears in page 23 of [Vo18] to compute the result after these stitchings.

For the left-hand side of (2.3) we have that:

$$\begin{aligned} \xi // m_1^{1,2} // m_1^{1,3} &= \left(\begin{array}{c|ccc} \omega(1 - \alpha_{12}) & 1 & 3 & S \\ \hline 1 & \alpha_{21} + \frac{\alpha_{11}\alpha_{22}}{1 - \alpha_{12}} & \alpha_{23} + \frac{\alpha_{22}\alpha_{13}}{1 - \alpha_{12}} & \frac{\alpha_{22}}{1 - \alpha_{12}}\theta_1 + \theta_2 \\ 3 & \alpha_{31} + \frac{\alpha_{11}\alpha_{32}}{1 - \alpha_{12}} & \alpha_{33} + \frac{\alpha_{32}\alpha_{13}}{1 - \alpha_{12}} & \theta_3 + \frac{\alpha_{32}}{1 - \alpha_{12}}\theta_1 \\ S & \psi_1 + \frac{\alpha_{11}}{1 - \alpha_{12}}\psi_2 & \psi_3 + \frac{\alpha_{13}}{1 - \alpha_{12}}\psi_2 & \Xi + \frac{\psi_2\theta_1}{1 - \alpha_{12}} \end{array} \right) // m_1^{1,3} \\ &= \left(\begin{array}{c|cc} \omega(1 - \alpha_{12})(1 - \alpha_{23} - \frac{\alpha_{13}\alpha_{22}}{1 - \alpha_{12}}) & 1 & S \\ \hline 1 & \mu_{11} & \mu_{1S} \\ S & \mu_{S1} & \mu_{SS} \end{array} \right), \end{aligned}$$

where

$$\begin{aligned} \mu_{11} &= \frac{\alpha_{31} - \alpha_{12}\alpha_{31} - \alpha_{13}\alpha_{22}\alpha_{31} - \alpha_{23}\alpha_{31} + \alpha_{12}\alpha_{23}\alpha_{31} + \alpha_{11}\alpha_{32} + \alpha_{13}\alpha_{21}\alpha_{32} - \alpha_{11}\alpha_{23}\alpha_{32} + \alpha_{21}\alpha_{33}}{1 - \alpha_{12} - \alpha_{13}\alpha_{22} - \alpha_{23} + \alpha_{12}\alpha_{23}} \\ &+ \frac{-\alpha_{12}\alpha_{21}\alpha_{33} + \alpha_{11}\alpha_{22}\alpha_{33}}{1 - \alpha_{12} - \alpha_{13}\alpha_{22} - \alpha_{23} + \alpha_{12}\alpha_{23}}, \\ \mu_{1S} &= \frac{\alpha_{32}\theta_1 - \alpha_{23}\alpha_{32}\theta_1 + \alpha_{22}\alpha_{33}\theta_1 + \alpha_{13}\alpha_{32}\theta_2 + \alpha_{33}\theta_2 - \alpha_{12}\alpha_{33}\theta_2 + \theta_3 - \alpha_{12}\theta_3 - \alpha_{13}\alpha_{22}\theta_3 - \alpha_{23}\theta_3}{1 - \alpha_{12} - \alpha_{13}\alpha_{22} - \alpha_{23} + \alpha_{12}\alpha_{23}} \\ &+ \frac{\alpha_{12}\alpha_{23}\theta_3}{1 - \alpha_{12} - \alpha_{13}\alpha_{22} - \alpha_{23} + \alpha_{12}\alpha_{23}}, \end{aligned}$$

$$\mu_{S1} = \frac{\psi_1 - \alpha_{12}\psi_1 - \alpha_{13}\alpha_{22}\psi_1 - \alpha_{23}\psi_1 + \alpha_{12}\alpha_{23}\psi_1 + \alpha_{11}\psi_2 + \alpha_{13}\alpha_{21}\psi_2 - \alpha_{11}\alpha_{23}\psi_2 + \alpha_{21}\psi_3 - \alpha_{12}\alpha_{21}\psi_3}{1 - \alpha_{12} - \alpha_{13}\alpha_{22} - \alpha_{23} + \alpha_{12}\alpha_{23}} + \frac{\alpha_{11}\alpha_{22}\psi_3}{1 - \alpha_{12} - \alpha_{13}\alpha_{22} - \alpha_{23} + \alpha_{12}\alpha_{23}},$$

and finally

$$\mu_{SS} = \frac{\Xi - \Xi\alpha_{12} - \Xi\alpha_{13}\alpha_{22} - \Xi\alpha_{23} + \Xi\alpha_{12}\alpha_{23} + \theta_1\psi_2 - \alpha_{23}\theta_1\psi_2 + \alpha_{13}\theta_2\psi_2 + \alpha_{22}\theta_1\psi_3 + \theta_2\psi_3 - \alpha_{12}\theta_2\psi_3}{1 - \alpha_{12} - \alpha_{13}\alpha_{22} - \alpha_{23} + \alpha_{12}\alpha_{23}}.$$

For the right-hand side of (2.3) we have that

$$\xi // m_2^{2,3} // m_1^{1,2} = \left(\begin{array}{c|ccc} \omega(1 - \alpha_{23}) & 1 & 2 & S \\ \hline 1 & \alpha_{11} + \frac{\alpha_{13}\alpha_{12}}{1 - \alpha_{23}} & \alpha_{12} + \frac{\alpha_{22}\alpha_{13}}{1 - \alpha_{23}} & \frac{\alpha_{13}}{1 - \alpha_{23}}\theta_2 + \theta_1 \\ 2 & \alpha_{31} + \frac{\alpha_{33}\alpha_{21}}{1 - \alpha_{23}} & \alpha_{32} + \frac{\alpha_{22}\alpha_{33}}{1 - \alpha_{23}} & \theta_3 + \frac{\alpha_{33}}{1 - \alpha_{23}}\theta_2 \\ S & \psi_1 + \frac{\alpha_{21}}{1 - \alpha_{23}}\psi_3 & \psi_3 + \frac{\alpha_{23}}{1 - \alpha_{23}}\psi_3 & \Xi + \frac{\psi_3\theta_2}{1 - \alpha_{23}} \end{array} \right) // m_1^{1,2} \\ = \left(\begin{array}{c|cc} \omega(1 - \alpha_{12})(1 - \alpha_{23} - \frac{\alpha_{13}\alpha_{22}}{1 - \alpha_{12}}) & 1 & S \\ \hline 1 & \mu'_{11} & \mu'_{1S} \\ S & \mu'_{S1} & \mu'_{SS} \end{array} \right).$$

Using the program in Mathematica mentioned above we see that $\mu'_{11}, \mu'_{1S}, \mu'_{S1}$ and μ'_{SS} coincide with $\mu_{11}, \mu_{1S}, \mu_{S1}$ and μ_{SS} respectively.

We show now the left-identity property. Let

$$\xi = \left(\begin{array}{c|ccc} \omega & \bar{a} & \bar{b} & S \\ \hline \bar{a} & \alpha & \beta & \theta \\ \bar{b} & \gamma & \delta & \epsilon \\ S & \phi & \psi & \Xi \end{array} \right).$$

We want to show $\xi // e_a // m_c^{a,\bar{b}} = \xi // \sigma_c^{\bar{b}}$.

For the left-hand side we have:

$$\xi // e_a // m_c^{a,\bar{b}} = \left(\begin{array}{c|cccc} \omega & a & \bar{a} & \bar{b} & S \\ \hline a & 1 & 0 & 0 & 0 \\ \bar{a} & 0 & \alpha & \beta & \theta \\ \bar{b} & 0 & \gamma & \delta & \epsilon \\ S & 0 & \phi & \psi & \Xi \end{array} \right) // m_c^{a,\bar{b}} = \left(\begin{array}{c|ccc} \omega & \bar{a} & c & S \\ \hline \bar{a} & \alpha & \beta & \theta \\ c & \gamma & \delta & \epsilon \\ S & \phi & \psi & \Xi \end{array} \right).$$

For the right-hand side we have:

$$\xi // \sigma_c^{\bar{b}} = \left(\begin{array}{c|ccc} \omega & \bar{a} & c & S \\ \hline \bar{a} & \alpha & \beta & \theta \\ c & \gamma & \delta & \epsilon \\ S & \phi & \psi & \Xi \end{array} \right).$$

The other properties follow similarly. \square

Now, there exists a meta-monoid homomorphism that will help us to represent a tangle in terms of an element of Γ -calculus, and so we will be able to write the operations of stitching, disjoint union, etc. in term of matrices.

Proposition 24. [Vo18]. *There exists a meta-monoid homomorphism φ (i.e. for each finite set X there exists a collection of maps $\varphi_X : \mathcal{T}_X \rightarrow \Gamma_X$ that commutes with the operations) from the meta-monoid \mathcal{T} of tangles to Γ -calculus. This map is generated by*

$$\varphi(R_{a,b}^{\pm}) = \left(\begin{array}{c|cc} 1 & a & b \\ \hline a & 1 & 1 - t^{\pm 1} \\ b & 0 & t^{\pm 1} \end{array} \right), \quad (2.4)$$

depending on the sign of the crossing.

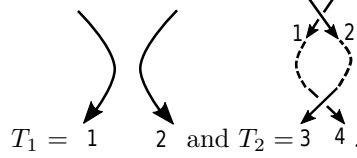


Figure 2.1: The tangles T_1 and T_2 are used to prove invariance under the Reidemeister move 3.

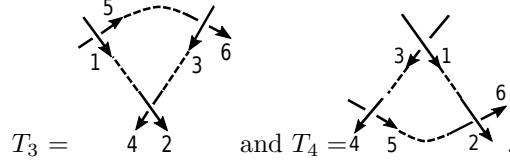


Figure 2.2: The tangles T_3 and T_4 are used to prove invariance under the Reidemeister move 3.

Proof. As we mentioned in section 2.1.1, when we want to define a meta-monoid homomorphism $\mathcal{T} \rightarrow \Gamma$ we only need to indicate the image of positive and negative crossings under this map and verify it is invariant under Reidemeister moves $R2$ and $R3$, so it remains to verify the definition under $R2$ and $R3$.

For the move $R2$, let us consider the tangles T_1 and T_2 from figure 2.1. Then

$$\varphi(T_1) = \left(\begin{array}{c|cc} 1 & 1 & 2 \\ \hline 1 & 1 & 0 \\ 2 & 0 & 1 \end{array} \right).$$

On the other hand, we know $T_2 = R_{2,1}^- R_{3,4}^+ // m_2^{2,3} // m_1^{1,4}$ and

$$\varphi(R_{2,1}^- R_{3,4}^+) = \left(\begin{array}{c|cccc} 1 & 1 & 2 & 3 & 4 \\ \hline 1 & t^{-1} & 0 & 0 & 0 \\ 2 & 1 - t^{-1} & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 1 - t \\ 4 & 0 & 0 & 0 & t \end{array} \right).$$

After stitching $m_2^{2,3}$ we have:

$$\varphi(R_{2,1}^- R_{3,4}^+ // m_2^{2,3}) = \left(\begin{array}{c|cccc} 1 & 1 & 2 & 4 \\ \hline 1 & t^{-1} & 0 & 0 \\ 2 & 1 - t^{-1} & 1 & 1 - t \\ 4 & 0 & 0 & t \end{array} \right).$$

The resulting matrix after $m_1^{1,4}$ is

$$\varphi(T_2) = \left(\begin{array}{c|cc} 1 & 1 & 2 \\ \hline 1 & 1 & 0 \\ 2 & 0 & 1 \end{array} \right).$$

Hence, φ is invariant under $R2$.

Now we want to show φ is invariant under $R3$. Consider the tangles T_3 and T_4 as in figure 2.2. Then

$$T_3 = R_{1,5}^+ R_{3,6}^+ R_{2,4}^- // m_1^{1,2} // m_3^{3,4} // m_5^{5,6},$$

so

$$\varphi(R_{1,5}^+ R_{3,6}^+ R_{2,4}^-) = \left(\begin{array}{c|cccccc} 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 0 & 0 & 0 & 1 - t & 0 \\ 2 & 0 & 1 & 0 & 1 - t^{-1} & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 & 0 & 1 - t \\ 4 & 0 & 0 & 0 & t^{-1} & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & t & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & t \end{array} \right).$$

After stitchings $m_1^{1,2} // m_3^{3,4} // m_5^{5,6}$ we get:

$$\varphi(T_3) = \left(\begin{array}{c|ccc} \omega = 1 & 1 & 3 & 5 \\ \hline 1 & 1 & 1-t^{-1} & t(1-t) \\ 3 & 0 & t^{-1} & 1-t \\ 5 & 0 & 0 & t^2 \end{array} \right).$$

Similarly, we can write

$$T_4 = R_{1,3}^- R_{4,5}^+ R_{2,6}^+ // m_1^{1,2} // m_3^{3,4} // m_5^{5,6},$$

so

$$\varphi(R_{1,3}^- R_{4,5}^+ R_{2,6}^+) = \left(\begin{array}{c|cccccc} 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 0 & 1-t^{-1} & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 & 0 & 1-t \\ 3 & 0 & 0 & t^{-1} & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 1 & 1-t & 0 \\ 5 & 0 & 0 & 0 & 0 & t & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & t \end{array} \right).$$

After stitchings $m_1^{1,2} // m_3^{3,4} // m_5^{5,6}$ we get:

$$\varphi(T_4) = \left(\begin{array}{c|ccc} \omega = 1 & 1 & 3 & 5 \\ \hline 1 & 1 & 1-t^{-1} & t(1-t) \\ 3 & 0 & t^{-1} & 1-t \\ 5 & 0 & 0 & t^2 \end{array} \right),$$

thus proving φ is invariant under Reidemeister move 3. \square

Now, let us refer to the map φ in (2.4) as the Γ -calculus map for tangles, or simply the Γ -calculus map. Let T be a tangle and assume $\varphi(T)$ has scalar part ω and matrix part A :

$$\varphi(T) = \left(\begin{array}{c|c} \omega & X \\ \hline X & A \end{array} \right).$$

We now present the following result which will be useful when we do the implementation of the Γ -calculus map in a computer, because at a certain point in the computations we will consider the Taylor expansion around 1 of degree 2 of the rational functions involved.

Lemma 25. *Let T be a tangle and $\varphi(T) = \left(\begin{array}{c|c} \omega & X \\ \hline X & A \end{array} \right)$. Let us consider the Taylor expansion around 1 of every entry of the matrix A and let us call this new matrix B . Let $s = 1 - t$. Then the off-diagonal entries of B expressed in terms of the variable s have no constant term. In other words, the off-diagonal entries are always divisible by s .*

Proof. We will prove the result by induction on the number of stitchings in the tangle T . For the base case, we need to consider a tangle consisting of a single positive or negative crossing. For a positive crossing $\begin{array}{c} \nearrow \\ \searrow \\ \text{a} \quad \text{b} \end{array}$ we know:

$$\varphi\left(\begin{array}{c} \nearrow \\ \searrow \\ \text{a} \quad \text{b} \end{array}\right) = \left(\begin{array}{c|cc} 1 & a & b \\ \hline a & 1 & 1-t \\ b & 0 & t \end{array} \right) = \left(\begin{array}{c|cc} 1 & a & b \\ \hline a & 1 & s \\ b & 0 & 1-s \end{array} \right).$$

For a negative crossing $\begin{array}{c} \nwarrow \\ \nearrow \\ \text{b} \quad \text{a} \end{array}$ we have the following:

$$\varphi\left(\begin{array}{c} \nwarrow \\ \nearrow \\ \text{b} \quad \text{a} \end{array}\right) = \left(\begin{array}{c|cc} 1 & a & b \\ \hline a & 1 & 1-t^{-1} \\ b & 0 & t^{-1} \end{array} \right) = \left(\begin{array}{c|cc} 1 & a & b \\ \hline a & 1 & -s(1+s+s^2+s^3+\dots) \\ b & 0 & \frac{1}{1-s} \end{array} \right),$$

since $s = 1 - t$. We see that in the matrix parts of $\varphi(\begin{smallmatrix} \gamma \\ a \end{smallmatrix} \begin{smallmatrix} \delta \\ b \end{smallmatrix})$ and $\varphi(\begin{smallmatrix} \gamma \\ b \end{smallmatrix} \begin{smallmatrix} \delta \\ a \end{smallmatrix})$ the off-diagonal entries are divisible by s . This is the basis step for induction.

Assume that the result holds for n stitchings, $n \in \mathbb{N}$. We want to show the result holds for $n + 1$ stitchings. For this, we will look at the components $\gamma + \frac{\alpha\delta}{1-\beta}$, $\phi + \frac{\alpha\psi}{1-\beta}$, $\epsilon + \frac{\delta\theta}{1-\beta}$ and $\Xi + \frac{\psi\theta}{1-\beta}$ (see expression (2.1)) of

$$(\varphi(T) // m_{\mathbf{c}}^{\mathbf{a}, \mathbf{b}}) // m_{c_{n+1}}^{a_{n+1}, b_{n+1}},$$

where $\mathbf{a} = (a_1, a_2, \dots, a_n)$, $\mathbf{b} = (b_1, b_2, \dots, b_n)$ and $\mathbf{c} = (c_1, c_2, \dots, c_n)$.

First of all, for $\gamma + \frac{\alpha\delta}{1-\beta}$ we note that γ , α and δ all sit in the diagonal. Hence, we are not concerned about them. Next, for $\phi + \frac{\alpha\psi}{1-\beta}$: ϕ is a column with only off-diagonal entries and by the induction hypothesis, each one of these entries is of the form sP_i for some polynomial P_i . The same holds for ψ , so each one of its entries is of the form sQ_j for some polynomial Q_j . Since β is an off-diagonal entry, $\beta = sR$ for some polynomial R . Hence $\frac{1}{1-\beta} = 1 + \beta + \beta^2 + \beta^3 + \dots = 1 + sR + s^2R^2 + s^3R^3 + \dots$, so

$$\begin{aligned} \phi_i + \frac{\alpha\psi_i}{1-\beta} &= sP_i + \frac{\alpha sQ_i}{1-\beta} \\ &= sP_i + \alpha(1 + sR + s^2R^2 + \dots)sQ_i \\ &= s(P_i + \alpha(1 + sR + s^2R^2 + \dots)Q_i). \end{aligned}$$

This shows the result is valid for $n + 1$ stitchings in the case of $\phi + \frac{\alpha\psi}{1-\beta}$. A similar reasoning shows that the result is also valid for $n + 1$ stitchings in the case $\epsilon + \frac{\delta\theta}{1-\beta}$.

We now look at $\Xi + \frac{\psi\theta}{1-\beta}$. Note that we are only concerned about the (i, j) entries of this matrix with $i \neq j$. Each one of these entries is a multiple of s , so we can write $\Xi_{i,j} = sT_{i,j}$ for a polynomial $T_{i,j}$ and thus

$$\Xi = \begin{pmatrix} M_{1,1} & \cdots & sT_{1,k} \\ \vdots & \ddots & \vdots \\ sT_{k,1} & \cdots & M_{k,k} \end{pmatrix},$$

here k is the size of Ξ and $M_{i,i}$ denotes an arbitrary polynomial, not necessarily divisible by s . On the other hand

$$\begin{aligned} \psi\theta &= \begin{pmatrix} \psi_1 \\ \psi_2 \\ \vdots \\ \psi_k \end{pmatrix} (\theta_1, \theta_2, \dots, \theta_k) \\ &= \begin{pmatrix} sQ_1 \\ sQ_2 \\ \vdots \\ sQ_k \end{pmatrix} (s\tilde{Q}_1, s\tilde{Q}_2, \dots, s\tilde{Q}_k) \\ &= s^2 \begin{pmatrix} Q_1\tilde{Q}_1 & \cdots & Q_1\tilde{Q}_k \\ \vdots & \ddots & \vdots \\ Q_k\tilde{Q}_1 & \cdots & Q_k\tilde{Q}_k \end{pmatrix}. \end{aligned}$$

Therefore

$$\frac{\psi\theta}{1-\beta} = (1 + sR + s^2R^2 + s^3R^3 + \dots)s^2 \begin{pmatrix} Q_1\tilde{Q}_1 & \cdots & Q_1\tilde{Q}_k \\ \vdots & \ddots & \vdots \\ Q_k\tilde{Q}_1 & \cdots & Q_k\tilde{Q}_k \end{pmatrix}.$$

Since $\left(\Xi + \frac{\psi\theta}{1-\beta}\right)_{i,j} = sT_{i,j} + (1 + sR + s^2R^2 + \dots)s^2Q_i\tilde{Q}_j = s(T_{i,j} + (1 + sR + s^2R^2 + \dots)sQ_i\tilde{Q}_j)$, the entry $\left(\Xi + \frac{\psi\theta}{1-\beta}\right)_{i,j}$ for $(i \neq j) \in \{1, 2, \dots, k\}$ is also divisible by s . This concludes the inductive step. The result follows by induction. \square

2.2 Generalization of Alexander polynomial to tangles

In this section we finish the generalization of the Alexander polynomial of knots to tangles. For this purpose we use the following property of the image of a tangle T under φ :

Proposition 26. [Vo18]. *Let T be a tangle whose components are labeled by the set X and*

$$\varphi(T) = \left(\begin{array}{c|c} \omega & X \\ \hline X & M \end{array} \right).$$

Then the sum of the entries in each column of M is 1.

From this result we can deduce that if K is a long knot then its matrix part is 1, so in this case we only care about the scalar part ω_K of $\varphi(K)$:

$$\varphi(K) = \left(\begin{array}{c|c} \omega_K & 1 \\ \hline 1 & 1 \end{array} \right).$$

Example 27. Let us consider the right-handed long trefoil knot, shown in figure 2.3. The tangle notation for this knot is:

$$R_{1,2}^+ R_{4,3}^+ R_{6,5}^+ // m_1^{1,3} // m_1^{1,6} // m_1^{1,2} // m_1^{1,4} // m_1^{1,5}.$$

We will show that the polynomial we obtain by using the methods in this chapter coincides with (a multiple of) the Alexander polynomial we computed in section 1.3.1.

$$\varphi(R_{1,2}^+ R_{4,3}^+ R_{6,5}^+) = \left(\begin{array}{c|cccccc} 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 1-t & 0 & 0 & 0 & 0 \\ 2 & 0 & t & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & t & 0 & 0 & 0 \\ 4 & 0 & 0 & 1-t & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & t & 0 \\ 6 & 0 & 0 & 0 & 0 & 1-t & 1 \end{array} \right).$$

Then

$$\varphi(R_{1,2}^+ R_{4,3}^+ R_{6,5}^+) // m_1^{1,3} // m_1^{1,6} // m_1^{1,2} // m_1^{1,4} // m_1^{1,5} = (*),$$

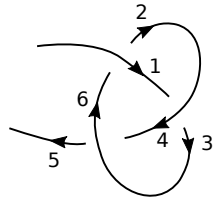


Figure 2.3: Right-handed long trefoil knot.

where

$$\begin{aligned}
(*) &= \left(\begin{array}{c|cccc} 1 & 1 & 2 & 4 & 5 & 6 \\ \hline 1 & t & t(1-t) & 0 & 0 & 0 \\ 2 & 0 & t & 0 & 0 & 0 \\ 4 & 1-t & (1-t)^2 & 1 & 0 & 0 \\ 5 & 0 & 0 & 0 & t & 0 \\ 6 & 0 & 0 & 0 & 1-t & 1 \end{array} \right) // m_1^{1,6} // m_1^{1,2} // m_1^{1,4} // m_1^{1,5} \\
&= \left(\begin{array}{c|cccc} 1 & 1 & 2 & 4 & 5 \\ \hline 1 & t & t(1-t) & 0 & 1-t \\ 2 & 0 & t & 0 & 0 \\ 4 & 1-t & (1-t)^2 & 1 & 0 \\ 5 & 0 & 0 & 0 & t \end{array} \right) // m_1^{1,2} // m_1^{1,4} // m_1^{1,5} \\
&= \left(\begin{array}{c|ccc} 1-t(1-t) & 1 & 4 & 5 \\ \hline 1 & \frac{t^2}{1-t(1-t)} & 0 & \frac{t(1-t)}{1-t(1-t)} \\ 4 & 1-t + \frac{t(1-t)^2}{1-t(1-t)} & 1 & \frac{(1-t)^3}{1-t(1-t)} \\ 5 & 0 & 0 & t \end{array} \right) // m_1^{1,4} // m_1^{1,5} \\
&= \left(\begin{array}{c|ccc} 1-t(1-t) & 1 & 5 \\ \hline 1 & 1-t + \frac{t(1-t)^2}{1-t(1-t)} + \frac{t^2}{1-t(1-t)} & \frac{t(1-t)}{1-t(1-t)} + \frac{(1-t)^3}{1-t(1-t)} \\ 5 & 0 & t \end{array} \right) // m_1^{1,5} \\
&= \left(\begin{array}{c|c} t(t^2-t+1) & 1 \\ \hline 1 & 1 \end{array} \right).
\end{aligned}$$

Hence $\omega = t(t^2 - t + 1)$. Note that $\frac{\omega}{t^2} = t - 1 + t^{-1}$, which coincides with our computations from section 1.3.1. \diamond

The example we just saw is not a coincidence, this happens for every long knot, as the following proposition states:

Proposition 28. [Vo18]. *Let K be a long knot and assume that*

$$\varphi(K) = \left(\frac{\omega}{1} \mid \frac{1}{1} \right).$$

Then $\omega \doteq \Delta_{\tilde{K}}(t)$, where $\Delta_{\tilde{K}}(t)$ denotes the Alexander polynomial of \tilde{K} and \tilde{K} is the closed knot obtained by closing the open component of K trivially and \doteq means equality up to multiplication by $\pm t^n$, $n \in \mathbb{Z}$.

The proof for this proposition can be found in [Vo18], page 32. The importance of this proposition is remarkable, since it tells us that we have now created an invariant of tangles that for a specific subset of the set of tangles (namely: long knots, which can be viewed as tangles with only one component) it coincides with the Alexander polynomial. Thus we have an extension of the Alexander polynomial of knots to the set of tangles.

We define the *Alexander polynomial of a tangle T* as the pair $(\omega, A) \in \Gamma_X \subset T_X \times M_{X \times X}(T_X)$ that results from taking $\varphi(T)$, where T is expressed as a collection of crossings and stitchings.

Note that $T_1 \cong T_2$ implies $(\omega_{T_1}, A_1) = (\omega_{T_2}, A_2)$ since φ is well-defined.

2.3 Some extra lemmas that will be useful for Chapter 4

The following lemmas give us the image under φ of some special tangles (we call them *multi-crossings*) that will appear as part of the computation of v_2 of a protein. Using these results as long as they are identified by the code can help to reduce the computation time.

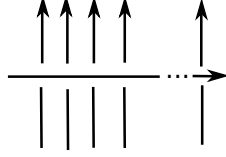


Figure 2.4: Multi-crossing for Lemma 29.

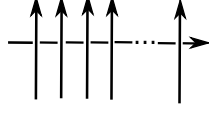


Figure 2.5: Multi-crossing for Lemma 30.

Lemma 29. *Let n be an odd natural number, $n \geq 3$. Let $T_n^+ = R_{1,2}^+ R_{3,4}^+ R_{5,6}^+ \cdots R_{n,n+1}^+ // m_1^{1,3} // m_1^{1,5} \cdots // m_1^{1,n}$. Then*

$$\varphi(T_n^+) = \left(\begin{array}{c|cccccc} 1 & 1 & 2 & 4 & 6 & \cdots & n+1 \\ \hline 1 & 1 & 1-t & 1-t & 1-t & \cdots & 1-t \\ 2 & 0 & t & 0 & 0 & \cdots & 0 \\ 4 & 0 & 0 & t & 0 & \cdots & 0 \\ 6 & 0 & 0 & 0 & t & \cdots & 0 \\ \vdots & \vdots & & & & \ddots & \\ n+1 & 0 & 0 & 0 & 0 & \cdots & t \end{array} \right). \quad (2.5)$$

The tangle diagram for T_n^+ is (equivalent to) that of figure 2.4.

Proof. We will prove this result by induction on n . The base case for the induction is for $n = 3$. Note that:

$$\varphi(R_{1,2}^+ R_{3,4}^+) = \left(\begin{array}{c|cccc} 1 & 1 & 2 & 3 & 4 \\ \hline 1 & 1 & 1-t & 0 & 0 \\ 2 & 0 & t & 0 & 0 \\ 3 & 0 & 0 & 1 & 1-t \\ 4 & 0 & 0 & 0 & t \end{array} \right).$$

After stitching $m_1^{1,3}$ we have

$$\varphi(T_3^+) = \varphi(R_{1,2}^+ R_{3,4}^+ // m_1^{1,3}) = \left(\begin{array}{c|ccc} 1 & 1 & 2 & 4 \\ \hline 1 & 1 & 1-t & 1-t \\ 2 & 0 & t & 0 \\ 4 & 0 & 0 & t \end{array} \right).$$

Now, we assume $\varphi(T_k^+)$ has the form (2.5). We want to show $\varphi(T_{k+2}^+)$ also does. Note the following:

$$\varphi(T_k^+ \sqcup R_{k+2,k+3}^+) = \left(\begin{array}{c|ccccccc} 1 & 1 & 2 & 4 & \cdots & k+1 & k+2 & k+3 \\ \hline 1 & 1 & 1-t & 1-t & \cdots & 1-t & 0 & 0 \\ 2 & 0 & t & 0 & \cdots & 0 & 0 & 0 \\ 4 & 0 & 0 & t & \cdots & 0 & 0 & 0 \\ \vdots & & & & \ddots & & & \\ k+1 & 0 & 0 & 0 & \cdots & t & 0 & 0 \\ k+2 & 0 & 0 & 0 & \cdots & t & 1 & 1-t \\ k+3 & 0 & 0 & 0 & \cdots & t & 0 & t \end{array} \right).$$

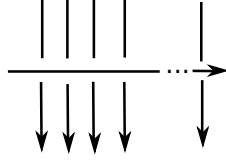


Figure 2.6: Multi-crossing.

For the stitching $m_1^{1,k+2}$ we have that $\alpha = 1$, $\beta = 0$, $\gamma = 0$, $\delta = 1$, $\theta = (1-t, 1-t, \dots, 1-t, 0)$, $\epsilon = (0, 0, \dots, 0, 1-t)$, $\phi = (0, 0, \dots, 0)^t$, $\psi = (0, 0, \dots, 0)^t$ and

$$\Xi = \begin{pmatrix} t & 0 & 0 & 0 \\ 0 & t & 0 & 0 \\ & & \ddots & \\ 0 & 0 & 0 & t \end{pmatrix}.$$

Hence, after $m_1^{1,k+2}$ we have:

$$\varphi(T_{k+2}^+) = \left(\begin{array}{c|cccccc} 1 & 1 & 2 & 4 & 6 & \cdots & k+1 & k+3 \\ \hline 1 & 1 & 1-t & 1-t & 1-t & \cdots & 1-t & 1-t \\ 2 & 0 & t & 0 & 0 & \cdots & 0 & 0 \\ 4 & 0 & 0 & t & 0 & \cdots & 0 & 0 \\ 6 & 0 & 0 & 0 & t & \cdots & 0 & 0 \\ \vdots & & & & & \ddots & & \\ k+1 & 0 & 0 & 0 & 0 & \cdots & t & 0 \\ k+3 & 0 & 0 & 0 & 0 & \cdots & 0 & t \end{array} \right),$$

which concludes the proof. \square

Similarly, we have the following result for negative crossings. The way to prove it is exactly the same as we did in the previous lemma.

Lemma 30. *Let n be an odd natural number, $n \geq 3$. Let $T_n^- = R_{2,1}^- R_{4,3}^- R_{6,5}^- \cdots R_{n+1,n}^- // m_1^{1,3} // m_1^{1,5} \cdots // m_1^{1,n}$. Then*

$$\varphi(T_n^-) = \left(\begin{array}{c|cccccc} 1 & 1 & 2 & 4 & 6 & \cdots & n+1 \\ \hline 1 & t & 0 & 0 & 0 & \cdots & 0 \\ 2 & 1-t^{-1} & 1 & 0 & 0 & \cdots & 0 \\ 4 & (1-t^{-1})t^{-1} & 0 & 1 & 0 & \cdots & 0 \\ 6 & (1-t^{-1})t^{-2} & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & & & & \ddots & \\ n+1 & (1-t^{-1})t^{-(\frac{n+1}{2}-1)} & 0 & 0 & 0 & \cdots & 1 \end{array} \right). \quad (2.6)$$

The tangle diagram for T_n^- is (equivalent to) that of figure 2.5.

We can get similar results for the next multi-crossings:

- $R_{1,2}^- R_{3,4}^- R_{5,6}^- \cdots R_{n,n+1}^- // m_1^{1,3} // m_1^{1,5} \cdots // m_1^{1,n}$. Its tangle diagram is (equivalent to) that of figure 2.6.
- $R_{2,1}^+ R_{4,3}^+ R_{6,5}^+ \cdots R_{n+1,n}^+ // m_1^{1,3} // m_1^{1,5} \cdots // m_1^{1,n}$. Its tangle diagram is (equivalent to) that of figure 2.7.

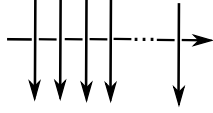


Figure 2.7: Multi-crossing.

2.4 Generalization of v_2 to tangles

We use the extension of the Alexander polynomial to tangles from the previous sections as a model to define an extension of the invariant v_2 of knots to tangles, for which we use the invariant Γ -calculus as basis step.

We have seen before that in the case of a knot K , v_2 can be defined as $\frac{1}{2}\Delta''(1)$, where $\Delta(t)$ is the Alexander polynomial of K . We now present a stitching rule for v_2 inspired by this fact. Let T be a tangle, and A the matrix part of $\varphi(T)$. We consider again the Taylor expansion around 1 of every entry of A . Since we only care about the second derivative and we will evaluate at 1, all the terms of degree three or bigger in the Taylor expansion of the polynomial around 1 will vanish. We recall that since β is outside the diagonal, it is of the form $\beta = sR$, for a polynomial R . Then,

$$\frac{1}{1-\beta} = 1 + \beta + \beta^2 + \beta^3 + \dots = 1 + sR + s^2R^2 + s^3R^3 + \dots$$

Now, we are only interested in the terms of degree two or lower. Hence, we can write

$$\frac{1}{1-\beta} = 1 + \beta + \beta^2.$$

Therefore we define the stitching rule for v_2 by:

$$:= \left(\begin{array}{c|ccc} \omega & a & b & S \\ \hline a & \alpha & \beta & \theta \\ b & \gamma & \delta & \epsilon \\ S & \phi & \psi & \Xi \end{array} \right) // m_c^{a,b} \\ := \left(\begin{array}{c|ccc} (1-\beta)\omega \pmod{s^3} & c & S & \\ \hline c & \gamma + \alpha\delta(1 + \beta + \beta^2) \pmod{s^3} & \epsilon + \delta\theta(1 + \beta + \beta^2) \pmod{s^3} & \\ S & \phi + \alpha\psi(1 + \beta + \beta^2) \pmod{s^3} & \Xi + \psi\theta(1 + \beta + \beta^2) \pmod{s^3} & \end{array} \right). \quad (2.7)$$

We readily see that this operation is well-defined (since this time we have $P(s) = 1$ as denominator in our rational fraction). We have seen in lemma 25 that this is in $\Gamma_{\{c\} \cup S}$.

Definition 31. Let T be a tangle, expressed as a collection of crossings and stitchings. Let $(\omega, A) \in \Gamma_X \subset T_X \times M_{X \times X}(T_X)$ be the image of T under φ and the stitching rule (2.7). Define $v_2(T) := (k, A)$ where $k := \frac{1}{2}\omega''(1)$, or equivalently, k is the coefficient of the quadratic term of ω . We call k the *scalar part* and A the *matrix part*. \diamond

Lemma 32. The function $v_2 : \mathcal{T}_X \rightarrow \Gamma_X$ is an invariant of tangles.

Proof. All we need to show is v_2 remains unchanged under Reidemeister moves $R2$ and $R3$. For $R2$, let us consider the tangles T_1 and T_2 from figure 2.1. Note that we are now using the stitching rule for v_2 . Then

$$v_2(T_1) = \left(\begin{array}{c|cc} 0 & 1 & 2 \\ \hline 1 & 1 & 0 \\ 2 & 0 & 1 \end{array} \right).$$

On the other hand, we know that $\varphi(T_2) = (\omega_{T_2}, A)$. Then

$$\begin{aligned} v_2(T_2) &= v_2(R_{2,1}^- R_{3,4}^+ // m_2^{2,3} // m_1^{1,4}) \\ &= \left(\frac{1}{2}\omega_{T_2}''(1), A \right). \end{aligned}$$

We know $T_2 = R_{2,1}^- R_{3,4}^+ // m_2^{2,3} // m_1^{1,4}$ and

$$\varphi(R_{2,1}^- R_{3,4}^+) = \left(\begin{array}{c|cccc} 1 & 1 & 2 & 3 & 4 \\ \hline 1 & t^{-1} & 0 & 0 & 0 \\ 2 & 1-t^{-1} & 1 & 0 & 0 \\ 3 & 0 & 0 & 1 & 1-t \\ 4 & 0 & 0 & 0 & t \end{array} \right).$$

After stitchings $m_2^{2,3} // m_1^{1,4}$ we have:

$$v_2(T_2) = \left(\begin{array}{c|cc} 0 & 1 & 2 \\ \hline 1 & 1 & 0 \\ 2 & 0 & 1 \end{array} \right).$$

Hence, v_2 is invariant under $R2$. Next, we show v_2 is invariant under $R3$. For this, consider the tangles T_3 and T_4 as in figure 2.2. Then we can write

$$T_3 = R_{1,5}^+ R_{3,6}^+ R_{2,4}^- // m_1^{1,2} // m_3^{3,4} // m_5^{5,6},$$

so

$$\varphi(R_{1,5}^+ R_{3,6}^+ R_{2,4}^-) = \left(\begin{array}{c|cccccc} 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 0 & 0 & 0 & 1-t & 0 \\ 2 & 0 & 1 & 0 & 1-t^{-1} & 0 & 0 \\ 3 & 0 & 0 & 1 & 0 & 0 & 1-t \\ 4 & 0 & 0 & 0 & t^{-1} & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & t & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & t \end{array} \right).$$

After stitchings $m_1^{1,2} // m_3^{3,4} // m_5^{5,6}$ we get:

$$v_2(T_3) = \left(\begin{array}{c|ccc} \frac{1}{2}\omega''_{T_3}(1) = 0 & 1 & 3 & 5 \\ \hline 1 & 1 & t - (t-1)^2 - 1 & -t - (t-1)^2 + 1 \\ 3 & 0 & -t + (t-1)^2 + 2 & 1-t \\ 5 & 0 & 0 & 2t + (t-1)^2 - 1 \end{array} \right).$$

Similarly, we can write

$$T_4 = R_{1,3}^- R_{4,5}^+ R_{2,6}^+ // m_1^{1,2} // m_3^{3,4} // m_5^{5,6},$$

so

$$\varphi(R_{1,3}^- R_{4,5}^+ R_{2,6}^+) = \left(\begin{array}{c|cccccc} 1 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & 0 & 1-t^{-1} & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 & 0 & 1-t \\ 3 & 0 & 0 & t^{-1} & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 1 & 1-t & 0 \\ 5 & 0 & 0 & 0 & 0 & t & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & t \end{array} \right).$$

After stitchings $m_1^{1,2} // m_3^{3,4} // m_5^{5,6}$ we get:

$$v_2(T_4) = \left(\begin{array}{c|ccc} \frac{1}{2}\omega''_{T_4}(1) = 0 & 1 & 3 & 5 \\ \hline 1 & 1 & t - (t-1)^2 - 1 & -t - (t-1)^2 + 1 \\ 3 & 0 & -t + (t-1)^2 + 2 & 1-t \\ 5 & 0 & 0 & 2t + (t-1)^2 - 1 \end{array} \right),$$

thus proving v_2 is invariant under Reidemeister move 3. \square

2.4.1 Implementation of the stitching rule for v_2 in Python

The code that performs these calculations in Python can be found in Chapter 5, under file 6. This code can also be used to check the computations that appear in this chapter.

Chapter 3

PDB files into tangles

3.1 Proteins and Protein Data Bank files

Proteins are large biomolecules, responsible for the transport and synthesis of nutrients, the creation of rigid structures in organisms such as skeletons, nails and hair, among other functions. The study of how different proteins interact with each other is extremely important to understand the complex processes inside the organisms that allow them to live, grow and reproduce. They are studied in areas such as organic chemistry and molecular biology but in recent years tools coming from computer science, physics and mathematics are proving to be more and more useful to understand how proteins work.

Proteins are made of substructures called *amino acids*. There are 20 amino acids which are listed in table 3-2 from page 127 of [Alb+07]. An amino acid is formed by a *central chain* (which is common to all amino acids) and a *side chain* (which is unique to each amino acid). The central chain is formed by the sequence -N-C-C-. See figure 3.1. Amino acids are classified according to their side chain into *basic*, *acidic*, *uncharged* and *non-polar*. Two amino acids are held together by a specific type of bond called *peptide bond*, see figure 3.2. This is the reason why proteins are also called *polypeptides*. Proteins usually have between 50 and 2000 amino acids [Alb+07].

The atoms forming amino acids are held together by a certain type of bond called a *covalent bond*, in which atoms share one pair or more of electrons. Another type of covalent bonds present in proteins is a *disulfide bond*, which occurs only between two cysteine amino acids. A *disulfide* is a chemical compound that contains the bond $S - S$ between two sulfur atoms.

There are four levels of the structure of proteins: *primary structure*, *secondary structure*, *tertiary structure* and *quaternary structure*. For the primary structure we are concerned about the sequence of amino acids in a chain. In the secondary structure we can find additional structural elements: α -helices and β -sheets. This level describes how different segments in the primary structure fold. This folding is produced by the presence of numerous *hydrogen bonds* that hold together different parts in the chain of a protein [Bru08]. For more specific information about them, see section 3.2.1.

The three-dimensional location of all the atoms in a protein is its tertiary structure. Some of the interactions responsible for the three-dimensional conformation of a protein are the disulfide bonds and some weaker attractions such as Van der Waals forces (hydrophobic attractions), electrostatic forces (between opposite charges) and hydrogen bonds. When the protein has two or more chains, we also consider the quaternary structure, in which we describe how the different chains interact with each other. An example of a protein with two chains is the human insulin. One consists of 21 amino acids and the

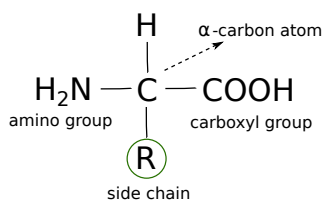


Figure 3.1: Diagram of an amino acid. The letter R represents the side chain.

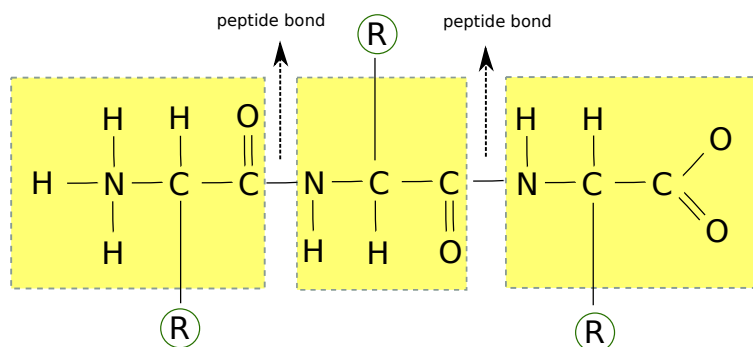


Figure 3.2: A protein is formed by a backbone (inside yellow boxes) and side chains attached to it.

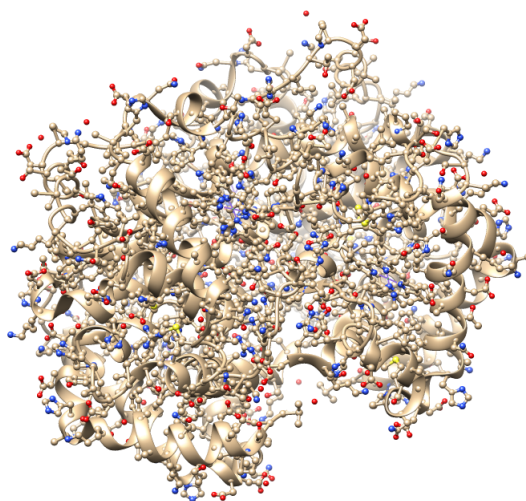


Figure 3.3: This is human hemoglobin-A, with PDB entry 1FN3. In this image we can appreciate several α -helices.

other contains 30 amino acids. Another example of a protein with more chains is human hemoglobin-A, which consists of four chains. Its quaternary structure is depicted in figure 3.3.

3.1.1 Protein Data Bank files

The Protein Data Bank (PDB) is an online repository that stores thousands of files containing the coordinates in three dimensional space (measured in Ångstroms) of the atoms forming a certain protein. Researchers from all over the world decode the structure of proteins and release this data to be publicly used. A unique code consisting of four characters (such as 1L2Y, 5JQ3, 1FN3) is assigned to each protein and can be used to retrieve the files from the PDB sever.

A typical PDB file looks as in figure 3.4. It is a text file that contains the 3-dimensional coordinates for the atoms of proteins and small molecules, such as water. A line of information starting with a keyword (ATOM or HETATM) is assigned to each atom. The ATOM keyword is used to identify protein atoms and HETATM is used for atoms of small molecules. One example is the protein with PDB entry 1ZIK, which consists of two identical chains of 30 amino acids each. The PDB file for this protein indicates at the end the location of 30 molecules of water. After these keywords there is a list of information including the name of the atom, its index in the file, the name and index of its residue, one letter to indicate the chain and the x , y and z coordinates.

There are other keywords included in the PDB file, for example AUTHOR and MODEL, that provide information about the experimental techniques used to determine the structure of a certain protein and

Atomic Coordinates: PDB Format

	Amino Acid	Element	Chain name	Sequence Number	-----Coordinates-----			(etc.)
					X	Y	Z	
ATOM	1	N	ASP L	1	4.060	7.307	5.186	...
ATOM	2	CA	ASP L	1	4.042	7.776	6.553	...
ATOM	3	C	ASP L	1	2.668	8.426	6.644	...
ATOM	4	O	ASP L	1	1.987	8.438	5.606	...
ATOM	5	CB	ASP L	1	5.090	8.827	6.797	...
ATOM	6	CG	ASP L	1	6.338	8.761	5.929	...
ATOM	7	OD1	ASP L	1	6.576	9.758	5.241	...
ATOM	8	OD2	ASP L	1	7.065	7.759	5.948	...

Element position within amino acid

Figure 3.4: This is how a PDB file looks. Image taken from [Hod+17].

the team of scientists responsible for the experiment.

3.2 Computing v_2 of a protein

We can mathematically represent a protein by a spatial graph where the vertices are the atoms and the edges are the covalent, hydrogen and sulfur bonds. The Alexander polynomial of proteins was studied in [Mun17], by using the algebraic definition of this invariant via the group of a knot and Fox derivatives. We use this research as a model for the present work. In [Mun17] it is mentioned that the existence of a vertex of at least degree 3 implies the corresponding Alexander polynomial is 0. Hence, as is done in this thesis, we look at certain cycles in the graph, and we project them onto a suitable plane to obtain a tangle.

Note that the tangle arising from projecting a long knot depends on the angle of projection, and this will also be the case when we compute the tangle coming from a protein. This issue does not prevent the use of tangles for studying protein structure, since there is a finite number of cycles in a protein graph and there is also a finite number of projections that give us a different tangle. By computing v_2 of all of them we get a unique set of values of v_2 of a protein.

In our model we assume the backbone (which is piecewise linear) is parallel to the xy -plane, as in figure 3.5.

3.2.1 PDB files into graphs

To process information about the protein, we wrote a program in the programming language Python. To read PDB files we use a package called *ProDy*, which provides classes representing the data, such as:

- *Chain*. A Chain object points to atoms in the same chain. The different chains that exist in a protein are indicated with the letters A, B, C , etc.
- *Residue*. A Residue object points to atoms with the same residue number. In biology, “residue” is another term for *amino acid*.

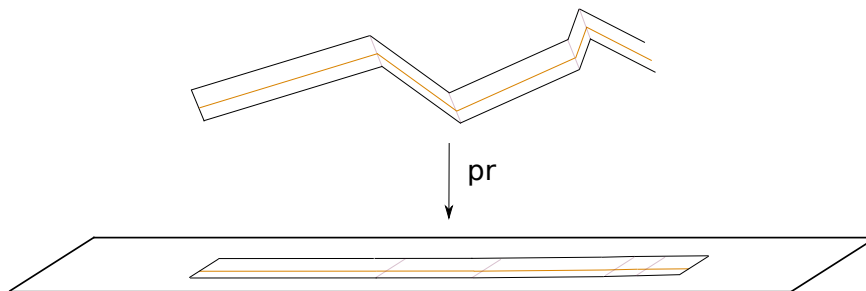


Figure 3.5: In our model we always project to the xy -plane.

- *Atom*. It points to only one atom. This is the lowest stage in the hierarchy.

ProDy reads a PDB file with the function `prody.parsePDB`. Some headers that are not parsed are: `KEYWDS`, `SSBOND` and `LINK`.

Example 33. Consider the protein with PDB entry 1L2Y. This is a mini protein consisting of 20 amino acids. In order to start working with it using ProDy we can write:

```
import prody
protein_1l2y=prody.parsePDB('1l2y')
```

Once the PDB file has been loaded, we can get information about its atoms by writing for example `protein_1l2y[0].getCoords()` to get the coordinates in the 3-dimensional space of this particular atom. We can also obtain the number of atoms in this protein (which is 305), the number of chains, number of residues by using several ProDy functions, see [BMB11]. \diamond

We recall that there exist different types of bonds that determine the three-dimensional conformation of a protein, and we would like to consider as much of these bonds as possible to create a realistic model for the protein. We will consider three types of bonds: covalent, hydrogen and disulfide bonds. Following [Hod+17] we say that a covalent bond exist between two atoms if they are at a distance $d \leq 1.9 \text{ \AA}$.

One of the representations of a graph in a computer we will use is by means of a dictionary, where the *keys* (see section 5.0.1) are vertices whose *value* (see section 5.0.1) are the neighbors of that key. The code in Python that finds covalent bonds based on a distance $d \leq 1.9 \text{ \AA}$ is the following:

```
edges = collections.defaultdict(list)
for atom_1 in protein_1l2y:
    for atom_2 in protein_1l2y:
        c1 = atom_1.getCoords()
        c2 = atom_2.getCoords()
        D=np.sqrt(numpy.dot(c1-c2, c1-c2))
        if D<=1.9 and D>0:
            edges[atom_1.getIndex()] .append(atom_2)
```

When we compare the graph generated by the above code (we create a picture of this graph using PyGame, see section 3.2.2) and the image generated with a molecular graphics program such as UCSF Chimera, we note that our program detects a lot of edges that are not really present in the protein. Thus, we need to refine the search for covalent bonds. We do this by using an important property of chemical elements: its covalent radius, as in [Mun17]. We consider the covalent radius of the chemical elements C, H, O, N and S plus 10 percent, to give a margin of error (for measurement). A covalent bond is declared to exist if the distance between the two atoms involved is less than the sum of their covalent radii plus the margin of error. The full code to find the bonds inside a protein using the covalent radii is part of the `ProteinStructure` class, included in Chapter 5.

As noted above, ProDy does not read information about disulfide bonds (indicated with the header `SSBOND`) so we have to do this manually. We illustrate this with an example, with the code:

```
with open('5jq3.pdb') as file_5jq3:
    for line in file_5jq3:
        if line.startswith('SSBOND'):
            print(line)
```

we get as answer:

```
SSBOND  1 CYS A  53    CYS B  609                1555  1555  2.05
SSBOND  2 CYS A 108    CYS A  135                1555  1555  2.06
SSBOND  3 CYS A 121    CYS A  147                1555  1555  2.05
SSBOND  4 CYS B 511    CYS B  556                1555  1555  2.07
SSBOND  5 CYS B 601    CYS B  608                1555  1555  2.05 .
```

We can now use the previous lines to create the edges of our graph corresponding to disulfide bonds. For this particular protein, with PDB entry 5JQ3 (its name is ebola glycoprotein, a viral protein affecting human beings), we see that there exist 5 disulfide bonds. The first one occurs between amino acids with indexes 53 and 609, the second between indexes 108 and 135, etcetera. With this information, now we only need to find the indexes of the sulfur atoms between which the disulfide bond appears. Another example is the human keratin-5 (with PDB entry 1FNU, present in human hair) in which there exists one disulfide bond.

We find the graph corresponding to the atoms in a protein (using the improved method to find covalent bonds and the method to find disulfide bonds) in the class `ProteinStructure`, included in Chapter 5.

Hydrogen bonds added to the graph

Determining hydrogen bonds between molecules is a very difficult task. There exist several criteria to determine when a hydrogen bond exist. The criterion we use appears in [McR99] page 266. We will consider a hydrogen bond between two atoms exists if one of them is a hydrogen atom H and the second one is an oxygen O, nitrogen N or sulfur S atom, there does not exist a covalent bond between them and they are located at a distance of at most 3.0 \AA . We will not take the angle of the bond into consideration for our purposes since we are not interested in the strength of hydrogen bonds in our model.

PyGame aids to the visualization of a graph

The code in file 5 in Chapter 5 finds the graph for a certain protein. However, we also want to visualize the structure of this graph, to have a concrete idea of what type of object we are dealing with. PyGame is a set of modules for Python that is mainly used for the creation of video games in two dimensions. However, it is an excellent tool to provide us with a visualization of the graph of a protein. For example, in figure 3.6 we include a picture generated with the visualization software called USCF Chimera (shown in green) and the one generated by our code in PyGame (shown in black) for the protein 1L2Y, which additionally includes the hydrogen bonds of this protein.

3.2.2 Graphs into tangles

So far we have generated a graph from a protein. Our next step is to convert this into a tangle by considering certain cycles in the graph and projecting them into a suitable plane. These cycles are defined in terms of the *spanning tree* of the graph, and will give the components of the cycle.

Definition 34. A *tree* is a contractible graph. Given a graph $X = (V, E)$, a subgraph of X that contains all of the vertices and is a tree is called a *spanning tree*. \diamond

Note that for a graph with n vertices, the number of edges in any spanning tree is equal to $n - 1$.

3.2.3 A general method using a suitable spanning tree from the graph of atoms in a protein

We first tried to convert the graph of atoms in a protein into a tangle, using spanning trees. In computer science, the following two algorithms are popular to find a spanning tree of a graph: Prim's algorithm and Kruskal's algorithm [Cor+09]. We worked with the former.

Prim's algorithm

This algorithm picks a random vertex and starts building the spanning tree out of this chosen vertex, checking at each stage of the process two things. First, the possible edges we can add to the spanning tree and second, the vertices that are already part of the spanning tree [Cor+09]. The code in Python is the following:

```
# Prim's algorithm
```

```
A = dict()
```

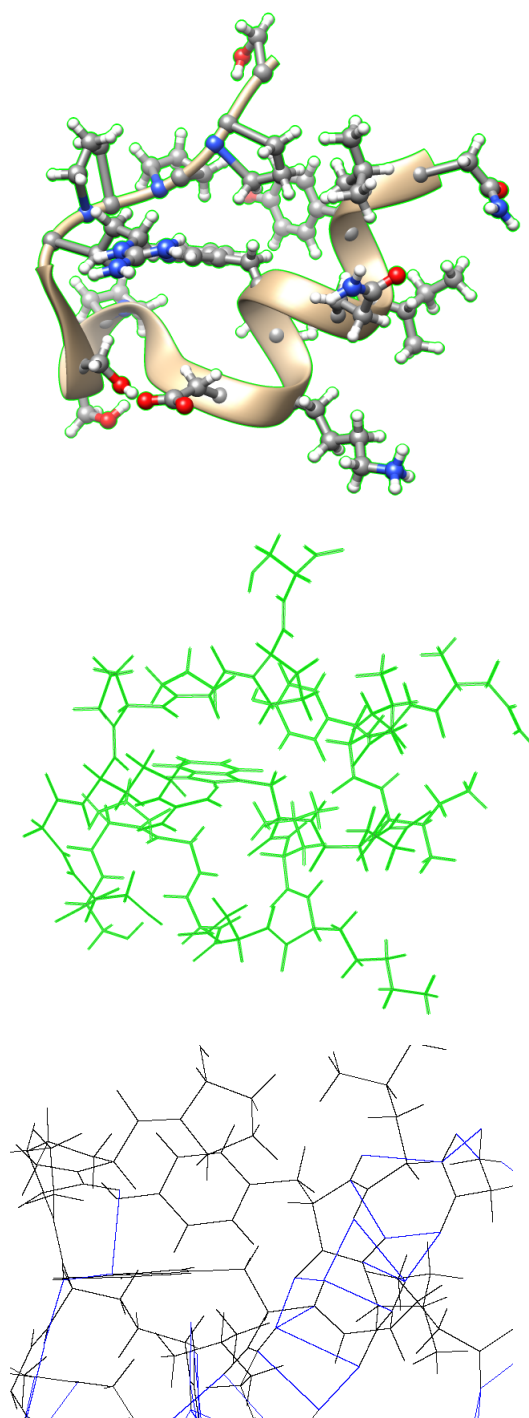


Figure 3.6: Protein with PDB entry 1L2Y. In the first image we show the atoms and the α -helix present in this protein. In the second picture the atoms and bonds for this protein are presented in the model *wire*. The first two images were created using UCSF Chimera. The second figure was created using the code in PyGame. The blue edges represent the hydrogen bonds. It is important to note these bonds visually coincide with the location of the α -helix.

```

B = set() # The set of nodes we have visited.
x=next(iter(graph.keys()))
tree = set() # a set of edges (v, w)
while True:
    # visit x
    B.add(x)
    for neighbor in graph[x]:
        if neighbor not in B:
            A[neighbor] = x, neighbor
    # go to the next x
    try:
        x, edge = next(iter(A.items())) #give an arbitrary element or raise StopIteration
    except StopIteration:
        # There is no next x, so we are finished with the tree.
        break

del A[x] #A has all the vertices neighbors to the tree but not in the tree
tree.add(edge)

```

It is important to mention that for a graph with several connected components, the version of Prim's algorithm we present here gives us only one spanning tree, corresponding to the connected component of the first vertex the algorithm chooses arbitrarily. A slightly modified code that finds one spanning tree for each connected component is presented in Appendix A.

The next example illustrates how Prim's algorithm works.

Example 35. Consider the following graph:

```

graph = { "a" : ["b"],
          "b" : ["a","e","c","g"],
          "c" : ["b","f","m","d"],
          "d" : ["c","j","i"],
          "e" : ["b","l"],
          "f" : ["c","g"],
          "g" : ["b","f","h","p"],
          "h" : ["g"],
          "i" : ["d","k","m"],
          "j" : ["l","d"],
          "k" : ["i"],
          "l" : ["e","j","n"],
          "m" : ["i","c","p","n"],
          "n" : ["l","m","p"],
          "p" : ["n","m","g"],
        }

```

Let

```
A = dict()    B = set()    tree = set().
```

The dictionary A will store the edges neighboring the tree and B will store the vertices in the spanning tree. The set tree will contain the vertices in our spanning tree. Assume we pick 'b' as the first vertex of our spanning tree, i.e. $x = 'b'$. Then in the first iteration:

```
A = dict()    B = {'b'}
neighbors = {'e','a','c','g'}
```

Since $\text{neighbors} - B$ is equal to neighbors, we add all the neighbors to 'b' and the respective edge connecting the neighbors to 'b' to the dictionary A:

```
A = {'e':('b','e'), 'a':('b','a'), 'c': ('b','c'), 'g': ('b','g')}
```

We pick an arbitrary key (and value) in A , say $x = 'e'$. The edge connecting 'b' and 'e' is $edge = ('b', 'e')$. We delete this key and value from A , i.e.

$A = \{'a':('b', 'a'), 'c':('b', 'c'), 'g': ('b', 'g')\}$

and we add $edge$ to $tree$, so

$tree = \{('b', 'e')\}$

and

$B = \{'b', 'e'\}$

Now the edges neighboring the tree are:

$A = \{'a':('b', 'a'), 'c':('b', 'c'), 'g': ('b', 'g'), 'l':('e', 'l')\}$

We keep picking an arbitrary key (and value) in A until there are no elements in A , i.e. there are no edges neighboring the tree, which means we are finished, since all the vertices belong to the tree already. \diamond

The spanning tree of a graph is not unique in general. We can define a unique spanning tree in certain cases: if we assign a *weight* to each edge, in our protein case this could be the length of a bond, then we can construct a *minimum spanning tree*. In the case of Prim's algorithm, (recall that we choose a random vertex and we start adding at every step one edge from the neighboring edges to the tree) we choose to add the edge of smallest weight, and using the next result we get a unique spanning tree.

Lemma 36. *If all the edges of a graph have different weight, then the minimum spanning tree is unique.*

Hence, we can slightly modify the graph representing a protein in a way such that all the edges have different length. Then, Prim's algorithm will give us a unique minimum spanning tree from which we can obtain a unique tangle.

3.2.4 Simplified graph of a protein

In general, a protein has very many atoms and bonds, so we decided to use a simpler graph. In very general terms, we will only look at the backbone of the protein and the amino acids involved in a hydrogen or sulfur bond. More specifically, we consider the following simplifications in our protein model:

1. The backbone of the protein will be formed by the α -carbons, i.e. we dismiss the N and second C atom in the central chain. These edges are drawn in yellow in figure 3.8. This simplification is annotated as `SIMP1` in the Python code.
2. If in an amino acid there exist hydrogen bonds with itself, we ignore them. This simplification is annotated as `SIMP2` in the Python code.
3. Let A and B be two amino acids, and c_A, c_B their respective α -carbons. Assume that there exist a (hydrogen or sulfur) bond between atom $a_1 \in A$ and atom $a_2 \in B$. Then we ignore the rest of the atoms in the side chain of A and B and we only consider the edge between a_1 and c_A and the edge between a_2 and c_B . These edges are drawn in black in figure 3.8. This simplification is annotated as `SIMP3` in the Python code.
4. If A and B are adjacent (i.e. assuming the residue number of A is i then the residue number for B is $i + 1$) then we ignore the bonds that might occur between them. For every pair of atoms, we only allow one connection between them. This simplification is annotated as `SIMP4` in the Python code.

We create the simplified version of a protein graph in the function `simplified_protein` inside file 5 from Chapter 5. We will refer to the edges between two α -carbons as *backbone bonds* (written B and depicted in yellow in the PyGame picture) and to the edges between an α -carbon and another atom from the side chain as *branch bonds* (written br and depicted in white in the PyGame picture). We will write hydrogen or sulfur bonds as bo , they are depicted in blue and red, respectively. We note the following important feature of this method: the union of the B edges and the br edges form a spanning tree of the protein graph. Most importantly, this spanning tree is unique! In the file 5 from Chapter 5 we also compute this canonical spanning tree for a protein as part of the output of `simplified_protein`.

In the next sections we take a suitable projection of a graph as initial point to get a tangle out of it, seen as a collection of crossings and instructions to stitch them.

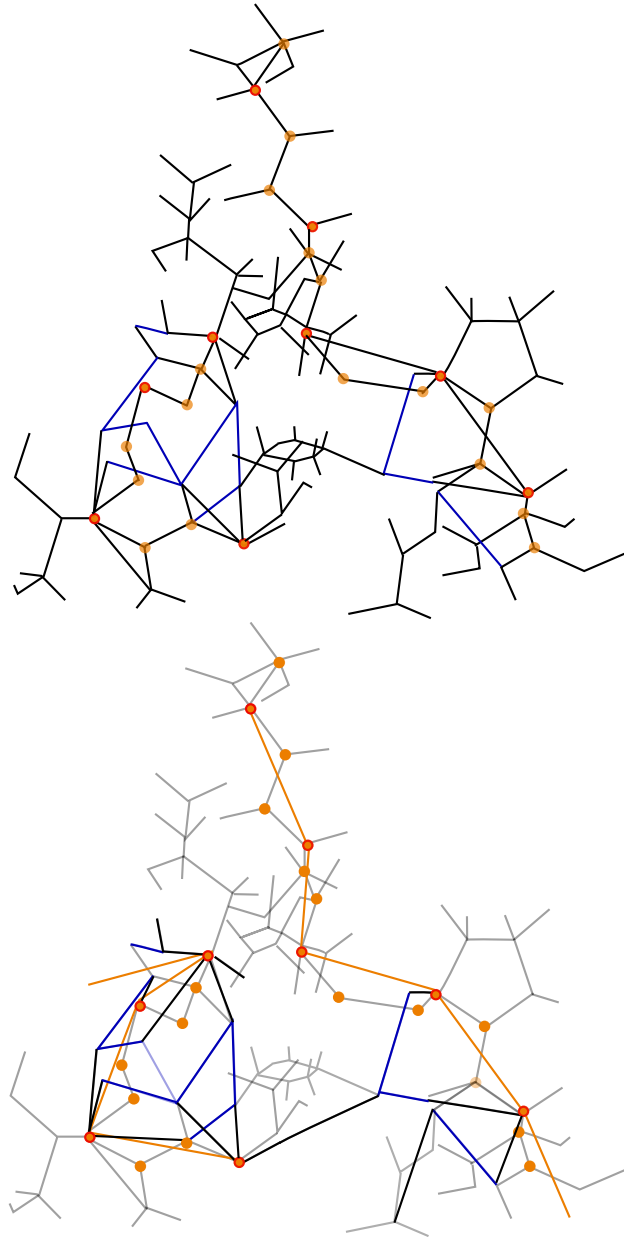


Figure 3.7: Protein graph towards simplification. The first figure shows the graph of atoms in a protein including hydrogen bonds (in blue). The α -carbons are represented by dark orange dots. The second figure shows the edges that we will consider in the simplified structure.

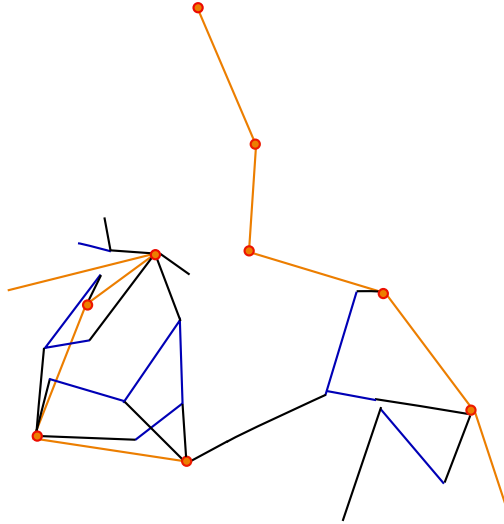


Figure 3.8: Protein graph simplified. The backbone of the protein is formed by the α -carbons. These edges are drawn in yellow. We ignore the atoms that are not involved in a hydrogen or sulfur bond. We connect the relevant atoms to their respective α -carbon. These edges are drawn in black. Hydrogen bonds are shown in blue.

3.2.5 Spanning tree to tangle

We take as initial step a graph with spanning tree, for example the simplified graph from section 3.2.4. We will find the “certain cycles” mentioned in the first paragraph of section 3.2 here. We fix a point in the spanning tree, called *root* (for simplified graphs, this is the first α -carbon) and we look at the edges which are not part of the spanning tree. For each one of this, we assign a cycle using the following result:

Theorem 37. [Die17]. *Let $X = (E, V)$ be a graph and S a tree. Then, for any two vertices v_1 and v_2 of X there exists a unique path through the tree from v_1 to v_2 .*

Let ρ be the root and suppose the edge $e = (x, y)$ (seen as a directed edge from x to y) is not part of the spanning tree. Then we assign the path $[\rho, x, y, \rho]$ to the edge e . We do this for every edge not in the spanning tree. Here we introduce the following name:

Definition 38. To an edge $e = (x, y)$ and a cycle e_j passing through either (x, y) or (y, x) , we associate a *lane*. We denote them by a triplet $(e_j, k, \text{True/False})$ to indicate the cycle e_j they belong to, the index k of the segment we are looking at in the cycle and if the direction agrees or not with the direction of e . In order to keep Python’s convention for indexes, k starts from 0. \diamond

Example 39. Cycle $C = [a, b, c, d, a]$ gives, among other, lane $(C, 0, T)$ through (a, b) and lane $(C, 2, F)$ through (d, c) . \diamond

We will build a tangle by adding strands, crossings for these strands, and stitching them. Each lane will correspond to a strand. The crossings of the strand are given by the intersections of the edge that the lane passes through, with other edges of the graph. The stitchings are given by iterating over a cycle and stitching all lanes in this cycle into one component in the corresponding order.

In the following subsections we explain in more detail the process to find a tangle from a graph.

We sort the lanes

Suppose we have found a cycle for each edge not in the spanning tree. Now we need to order the lanes for each edge in a consistent way. We should not create any additional intersections than the ones coming from the projection, and so that intersections between edges with multiple lanes have correctly ordered crossings.

Imagine that we are diving on a highway with many lanes ordered from left to right as l_1, l_2, \dots, l_n . Imagine a car in lane l_3 wants to leave this highway to the right exit. It will need to cross lanes l_4, l_5, \dots, l_n , so when merging onto the highway it should ensure it is already in lane l_n .

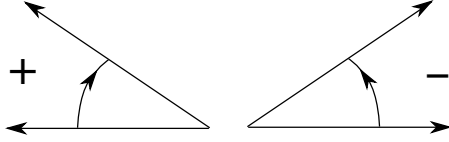


Figure 3.9: Convention for angles.

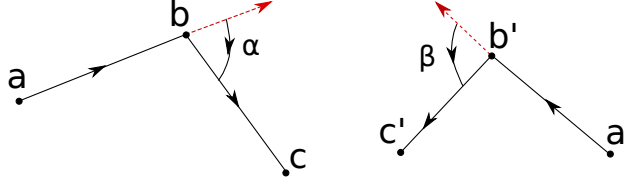


Figure 3.10: Angles for cycles.

A way in which we can order the lanes and avoid this issue is the following: we consider a sequence of *turning angles* for each one of the cycles not in our spanning tree, because this will tell us how the lanes fit with respect to a “bigger” setting.

My convention for the sign of the angle between two vectors is that of figure 3.9.

Sequence of turning angles for cycles

We iterate over triplets of adjacent vertices in the cycle (recall that a cycle is given by a sequence of vertices). We will consider all the angles except for the one between the first and last vertices of the cycle.

For a triplet (a, b, c) we consider the vectors $b - a$ and $c - b$ respectively. We consider the ray r leaving from a and passing through b , showed in red in figure 3.10. We compute the signed angle between the unitary vector u leaving from b in the same direction as the ray r (i.e. the vector u has the same direction as the vector in red from 3.10 for both cases) and $v_1 - v_2$. Thus, using the notation from figure 3.10, we get that the angle α between a, b and c is positive and the angle β between a', b' and c' is negative. Then, for a given cycle we will retrieve a list of angles as computed here. For example, for the cycle in picture 3.11 we get a sequence of angles $[\alpha, \beta, \gamma, \delta, \epsilon]$.

We will implement a signed angle function in Python as follows: let v_1 and v_2 be two vectors in \mathbb{R}^2 . We would like to compute the signed angle between these vectors. We can get the angle α between these vectors by computing $\alpha = \cos^{-1} \left(\frac{v \cdot w}{\|v\| \|w\|} \right)$, where $v \cdot w$ denotes the dot product of v and w . However, this angle is always positive. We can fix this as follows: consider the vector orthogonal to w_2 , w_2^\perp (if $w = (a, b)$, then $w^\perp = (-b, a)$), so that the signed angle α' is given by:

$$\alpha' = \text{sign}(v_2^\perp \cdot v_1) \alpha,$$

where

$$\alpha = \cos^{-1} \left(\frac{v_1 \cdot v_2}{\|v_1\| \|v_2\|} \right).$$

Note that for a cycle c with list of angles $[\alpha_1, \alpha_2, \dots, \alpha_n]$, its inverse cycle c^{-1} , i.e. the cycle going in the opposite direction, has the list of angles $[-\alpha_n, \dots, -\alpha_2, -\alpha_1]$.

Intersection of edges

We use the information in [Sun12] as reference for this section. In this section we will denote vectors in \mathbb{R}^2 by bold characters, to distinguish them from numbers.

Let l be a line in \mathbb{R}^n , $n \in \mathbb{N}$ and assume l goes through the points P_0 and P_1 . Then its parametric equation is given by

$$P(s) = P_0 + s(P_1 - P_0), \text{ where } s \in \mathbb{R}. \quad (3.1)$$

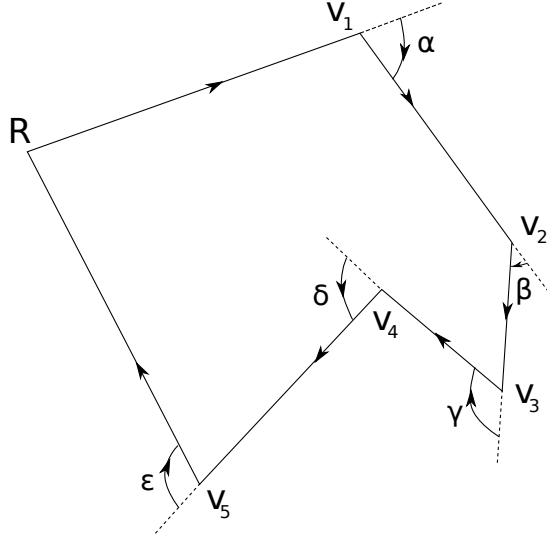


Figure 3.11: We assign to the cycle $[R, v_1, v_2, v_3, v_4, v_5, R]$ the list of *turning angles* $[\alpha, \beta, \gamma, \delta, \epsilon]$.

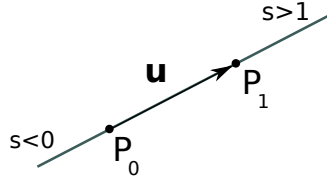


Figure 3.12: A linear segment.

If $\mathbf{u} = P_1 - P_0$, then we can write equation (3.1) as $P(s) = P_0 + s\mathbf{u}$. From this we know that $P(0) = P_0$, $P(1) = P_1$ and $s \in [0, 1]$ implies $P(s)$ is a point of the bounded segment P_0P_1 , see figure 3.7.

Now, we use this representation of a line to compute the intersection of two non-parallel lines in \mathbb{R}^2 . Consider two lines $P(s) = P_0 + s(P_1 - P_0)$ and $Q(t) = Q_0 + t(Q_1 - Q_0)$, assume they are non-parallel. Writing $\mathbf{u} = P_1 - P_0$ and $\mathbf{v} = Q_1 - Q_0$ we get the equations $P(s) = P_0 + s\mathbf{u}$ and $Q(t) = Q_0 + t\mathbf{v}$. Let us look at the diagram in figure 3.8. Let M be the point of intersection of these lines, to determine its coordinates we have the following vector equality:

$$P(s) - Q_0 = \mathbf{w} + s\mathbf{u}, \text{ where } \mathbf{w} = P_0 - Q_0.$$

At the intersection point M , the vector $P(s_M) - Q_0$ is perpendicular to \mathbf{v}^\perp . This condition is equivalent to $\mathbf{v}^\perp \cdot (\mathbf{w} + s_M\mathbf{u}) = 0$. From this, we get that

$$\begin{aligned} \mathbf{v}^\perp \cdot \mathbf{w} + \mathbf{v}^\perp \cdot (s_M\mathbf{u}) &= 0, \\ \mathbf{v}^\perp \cdot \mathbf{w} + s_M(\mathbf{v}^\perp \cdot \mathbf{u}) &= 0, \end{aligned}$$

so

$$s_M = \frac{-\mathbf{v}^\perp \cdot \mathbf{w}}{\mathbf{v}^\perp \cdot \mathbf{u}}.$$

Note that since the lines are not parallel, $\mathbf{v}^\perp \cdot \mathbf{u} \neq 0$. We determine t_M similarly. We look at the equality $Q(t) - P_0 = -\mathbf{w} + t\mathbf{v}$. At M we get $\mathbf{u}^\perp \cdot (-\mathbf{w} + t_M\mathbf{v}) = 0$, i.e. $\mathbf{u}^\perp \cdot (-\mathbf{w}) + \mathbf{u}^\perp \cdot (t_M\mathbf{v}) = 0$ or equivalently, $\mathbf{u}^\perp \cdot (-\mathbf{w}) + t_M(\mathbf{u}^\perp \cdot \mathbf{v}) = 0$. Hence,

$$t_M = \frac{\mathbf{u}^\perp \cdot \mathbf{w}}{\mathbf{u}^\perp \cdot \mathbf{v}}.$$

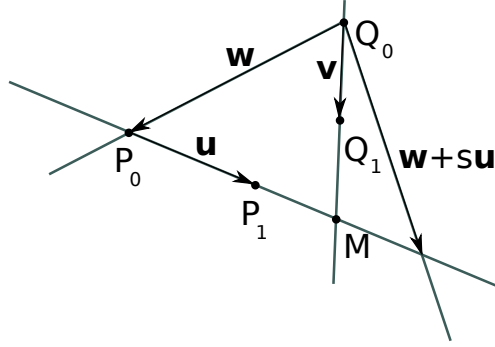


Figure 3.13: Intersection of segments.

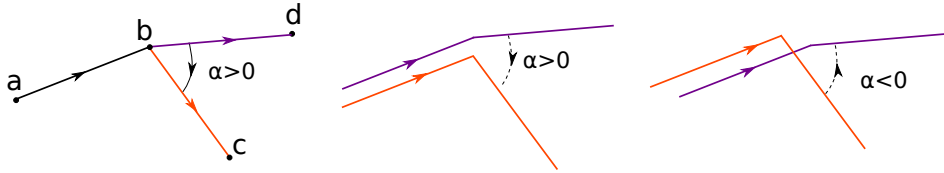


Figure 3.14: This is how we avoid creating extra intersections than the ones coming from the projection. When we create the lanes (in purple and orange) we need to preserve the sign of the angle between the edges.

Since the lines are not parallel, $\mathbf{u}^\perp \cdot \mathbf{v} \neq 0$.

The intersection point is given by $M = P(s_M) = Q(t_M)$. If both lines are segments, they intersect if and only if $s_M, t_M \in [0, 1]$. If only one of them is a segment, say Q_0Q_1 , then M is in this segment if and only if $t_M \in [0, 1]$.

Once we have the list of turning angles for each cycle and we have a list of all intersections, we determine which edges have multiple lanes, because these will need to be ordered.

Multiple lanes in an edge and their ordering

We can get the multiple lanes passing through an edge $e = (x, y)$ by searching this pair in the list of vertices of a cycle. We do this for every edge and every cycle.

Assume the edge $e = (x, y)$ has several lanes. Let l be a cycle that passes through e . Assume the index of this segment (inside the cycle) is i . Assume the list of angles for l is $[\alpha_0, \alpha_1, \dots, \alpha_i, \dots, \alpha_n]$.

If this segment goes in the same direction, we will split the list of angles for l as:

$$[\alpha_{i-1}, \alpha_i, \alpha_{i+1}, \dots, \alpha_n][\alpha_{i-2}, \alpha_{i-3}, \dots, \alpha_0].$$

If this segment goes in the opposite direction, we need to look at the list of angles of the inverse cycle: $[-\alpha_n, -\alpha_{n-1}, \dots, -\alpha_2, -\alpha_1, -\alpha_0]$. We split this list as:

$$[-\alpha_n, -\alpha_{n-1}, \dots, -\alpha_{i+1}][-\alpha_0, -\alpha_1, \dots, -\alpha_i].$$

After this we just use lexicographic order with the resulting list of angles. We order the lanes for an edge $e = (x, y)$ from left to right according to the orientation of e , see figure 3.14.

Collection of crossings and stitching instructions

We “travel” along each one of the cycles to detect for each lane how many strands we encounter, and we stitch them in this order.

All the steps mentioned in these sections will become clear when we deal with a concrete example. That will be the topic of our following section.

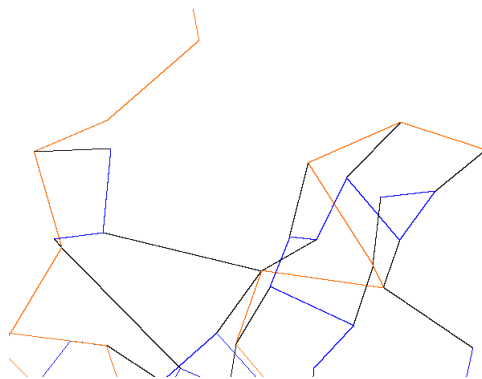


Figure 3.15: Portion of the simplified graph for protein with PDB entry 1L2Y drawn in PyGame. We show the B edges in orange, the br edges in black and hydrogen bonds bo in blue.

3.2.6 Example, we put the approach of section 3.2.4 into practice to compute v_2 of a tangle arising from a graph

Consider the graph in figure 3.17. This graph is based on a projection of a portion of the protein 1L2Y, shown in figure 3.15. We will try to simulate the real example as much as possible. For this, we give coordinates to our vertices that preserve the “shape” of the projection:

$$\begin{aligned} v_1 &= (10, 14, 1), v_2 = (7, 18, 4), v_3 = (1, 15, 0), v_4 = (4, 6, -10), v_5 = (11, 1, 5), \\ v_6 &= (-4, -1, 5), v_7 = (-5, -7, -3), v_{11} = (9, 11, 0), v_{21} = (4, 12, 0), v_{31} = (-4, 6, 1), \\ v_{41} &= (7, -4, -5), v_{21} = (4, 12, 10), v_{51} = (10, 9, 4), v_{61} = (-2, 5, 10), v_{71} = (1, -3, -2). \end{aligned}$$

The atoms in the PDB file for the protein 1L2Y are numbered according to the residues. The residues in the PDB file for 1L2Y are in the following order: Asn (1), Leu (2), Tyr (3), Ile (4), Gln (5), Trp (6), Leu (7), Lys (8), Asp (9), Gly (10), Gly (11), Pro(12), Ser (13), Ser (14), Gly (15), Arg (16), Pro (17), Pro (18), Pro (19) and Ser (20).

We created the backbone in the simplified structure by connecting only the α -carbons. These have indices: 2, 18, 37, 58, 77, 94, 118, 137, 159, 171, 178, 185, 199, 210, 221, 228, 252, 266, 280 and 294.

In our example we only show the first seven α -carbons. We illustrate how we number the atoms using the graph coming from the simplified graph for 1L2Y. Here, the atom connected to the first α -C (index 2) is an O atom with index 4. The atom connected to the second α -carbon is an O atom with index 20, the atom connected to the third α -carbon is an O atom with index 39, the atoms connected to the fourth α -carbon are an O atom with index 60 and a H atom with index 65, etcetera. We name the vertices in the backbone starting from 1. We name the atoms connected to backbone vertices by v_{i1}, v_{i2}, \dots . In our example 2 is the maximum number of br vertices. A br vertex is one of the vertices of a br bond (the other vertex in this type of bond is an α -C).

In the graph in figure 3.14 we have colored the bonds according to their type. Following the notation introduced in the previous paragraph, the backbone consists of the vertices $v_1, v_2, v_3, v_4, v_5, v_6$ and v_7 , and the br vertices are named $v_{11}, v_{21}, v_{31}, v_{41}, v_{42}, v_{51}, v_{61}$ and v_{71} . We will orient the hydrogen bonds leaving from the smaller index, as in the figure. In this picture we also represent the coloring scheme we use in PyGame: the backbone edges are yellow, the branch edges are gray, hydrogen bonds are blue and sulfur bonds are red.

Cycles

As we mentioned in section 3.2.4, the union of the yellow and gray edges creates a spanning tree. Hence, we will compute the cycles for the hydrogen (and sulfur) bonds since these are the edges outside the spanning tree taking the vertex v_3 as root. We will call the edge $(v_{11}, v_{51}) = e_1$, the edge $(v_{11}, v_{42}) = e_2$, the edge $(v_{21}, v_{51}) = e_3$, the edge $(v_{21}, v_{61}) = e_4$, the edge $(v_{31}, v_{61}) = e_5$, the edge $(v_{31}, v_{71}) = e_6$ and the edge $(v_{41}, v_{71}) = e_7$.

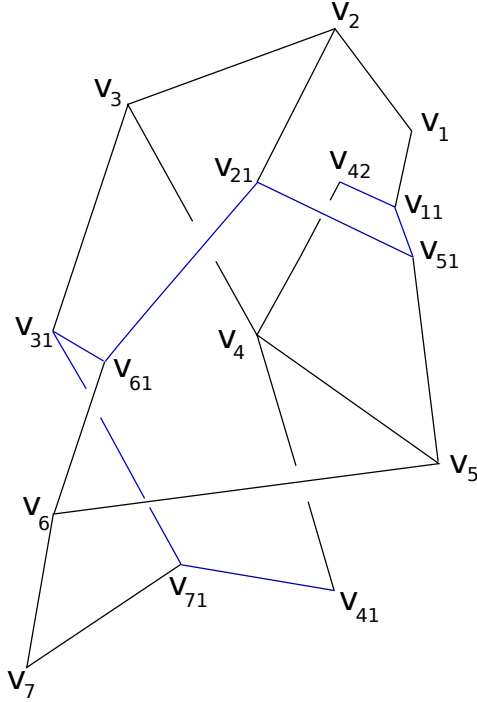


Figure 3.16: Protein graph simplified. Blue is *bo* and black are *B* and *br* edges (see section 3.2.2 for notation).

Then the cycles for these edges (denoted also by e_i) are:

$$\begin{aligned}
 e_1 &= [v_3, v_2, v_1, v_{11}, v_{51}, v_5, v_4, v_3], \\
 e_2 &= [v_3, v_2, v_1, v_{11}, v_{42}, v_4, v_3], \\
 e_3 &= [v_3, v_2, v_{21}, v_{51}, v_5, v_4, v_3], \\
 e_4 &= [v_3, v_2, v_{21}, v_{61}, v_6, v_5, v_4, v_3], \\
 e_5 &= [v_3, v_{31}, v_{61}, v_6, v_5, v_4, v_3], \\
 e_6 &= [v_3, v_{31}, v_{71}, v_7, v_6, v_5, v_4, v_3] \text{ and} \\
 e_7 &= [v_3, v_4, v_{41}, v_{71}, v_7, v_6, v_5, v_4, v_3].
 \end{aligned}$$

Note: In the actual Python code we took the first α -carbon of the protein to be the root, see section 3.2.5 for this convention.

Angles for cycles

The lists of angles (truncated after two decimals) for the cycles e_i are:

$$\begin{aligned}
 le_1 &= [73.86, 55.30, -45, 19.44, 132.66, 25.40], \\
 le_2 &= [73.68, 55.30, 98.13, -90, 12.38], \\
 le_3 &= [137.21, -90, 56.30, 132.66, 25.40], \\
 le_4 &= [137.21, 14.03, -22.16, -116.02, -136.86, 25.40], \\
 le_5 &= [-81.86, 81.86, -116.02, -136.86, 25.40], \\
 le_6 &= [-47.48, 85.36, 133.15, 72.94, -136.86, 25.40] \text{ and} \\
 le_7 &= [12.35, 116.16, -43.15, 133.15, 72.94, -136.86, 25.40]
 \end{aligned}$$

Recall that we do not compute the last angle (for the triplet v_{i-2}, v_{i-1} and the root).

Ordering of lanes

In this section we will order the lanes for two edges, the lanes for the remaining edges are ordered in a similar way.

First we look at an edge for which all the lanes passing through it go in the same direction: vertex (v_6, v_5) . By looking at the lists of vertices for the cycles e_1, \dots, e_7 we note that for the ordered edge (v_6, v_5) , there are four lanes going in the same direction. These lanes are: $(e_4, 4, \text{True})$, $(e_5, 3, \text{True})$, $(e_6, 4, \text{True})$ and $(e_7, 5, \text{True})$. Recall that we start to count the segments of a cycle from 0.

We perform the analysis from the subsection “Multiple lanes in an edge and their ordering” in the following table, using the lists of angles for the cycles e_4, \dots, e_7 from the previous subsection:

lane	index	same direction?	forward angles	backward angles
$(e_4, 4, \text{True})$	4	Yes	$[-116.02, -136.86, 25.40]$	$[-22.16, 14.03, 137.21]$
$(e_5, 3, \text{True})$	3	Yes	$[-116.02, -136.86, 25.40]$	$[81.86, -81.86]$
$(e_6, 4, \text{True})$	4	Yes	$[72.94, -136.86, 25.40]$	$[133.15, 85.36, -47.48]$
$(e_7, 4, \text{True})$	5	Yes	$[72.94, -136.86, 25.40]$	$[133.15, -43.15, 116.16, 12.35]$

For each lane we concatenate the lists of forward angles and backward angles and we order them lexicographically. This gives us the order:

$$(e_4, 4, \text{True}), (e_5, 3, \text{True}), (e_7, 5, \text{True}), (e_6, 4, \text{True}).$$

If we look at the picture in figure 3.16 we note the following:

1. The cycles e_4 and e_5 merge at v_{61} , e_4 comes from the left and e_5 comes from the right.
2. The cycles e_7 and e_6 merge at v_{71} , e_7 comes from the left and e_6 comes from the right.
3. These four cycles merge at v_6 but we need to care about their previous history, i.e. where they come from and how they fit with respect to other cycles previously. Putting all this information together we have that the correct order must be:

$$(e_4, 4, \text{True}), (e_5, 3, \text{True}), (e_7, 5, \text{True}), (e_6, 4, \text{True}).$$

Now we look at an edge for which there are lanes passing through it in both directions. For vertex (v_4, v_3) there are eight lanes: $(e_1, 6, \text{True})$, $(e_2, 5, \text{True})$, $(e_3, 5, \text{True})$, $(e_4, 6, \text{True})$, $(e_5, 5, \text{True})$, $(e_6, 6, \text{True})$ and also $(e_7, 0, \text{False})$, $(e_7, 7, \text{True})$.

We perform the analysis from the subsection “Multiple lanes in an edge and their ordering” in the following tables.

Lanes in the same direction:

lane	index	same direction?	forward angles	backward angles
$(e_1, 6, \text{True})$	6	Yes	$[25.40]$	$[132.66, 19.44, -45, 55.30, 73.86]$
$(e_2, 5, \text{True})$	5	Yes	$[124.38]$	$[-90, 98.13, 55.30, 73.68]$
$(e_3, 5, \text{True})$	5	Yes	$[25.40]$	$[132.66, 56.30, -90, 137.21]$
$(e_4, 6, \text{True})$	6	Yes	$[25.40]$	$[-136.86, -116.02, -22.16, 14.03, 137.21]$
$(e_5, 6, \text{True})$	5	Yes	$[25.40]$	$[-136.86, -116.02, 81.86, -81.86]$
$(e_6, 6, \text{True})$	6	Yes	$[25.40]$	$[-136.86, 72.94, 133.15, 85.36, -47.48]$
$(e_7, 7, \text{True})$	7	Yes	$[25.40]$	$[-136.86, 72.94, 133.15, -43.15, 116.16, 12.35]$

The lane $(e_7, 0, \text{False})$ goes in the opposite direction, so we need to consider the list of angles for the inverse cycle for e_7 :

$$le_7^{-1} = [-25.40, 136.86, -72.94, -133.15, 43.15, -116.16, -12.35]$$

Hence, we get the following table:

lane	index	same dir?	forward angles	backward angles
$(e_7, 0, \text{False})$	0	No	$[-25.40, 136.86, -72.94, -133.15, 43.15, -116.16]$	$[-12.35]$

The lexicographic order of the previous concatenated lists is:

$(e_7, 0, \text{False}), (e_4, 6, \text{True}), (e_5, 5, \text{True}), (e_7, 7, \text{True}), (e_6, 6, \text{True}), (e_1, 6, \text{True}), (e_3, 5, \text{True}), (e_2, 5, \text{True})$.

When we order the lanes for all the edges in the graph from figure 3.17 we get the picture of all the lanes of figure 3.18.

Strands

We first introduce the following convention for the naming of the strands of a crossing: the over strand is given an even number n and the under strand is given an odd number $n + 1$. We enumerate the strands of a crossing according to the following algorithm: we start with the first cycle e_1 and number all the multi-crossings we encounter while traveling this cycle. We do it respecting the ordering of lanes and using the above convention (the over strand is given an even number n and the under strand is given an odd number $n + 1$). We continue with the next cycles e_2, e_3 , etcetera, skipping the strands we already numbered.

We explain this from figure 3.19. The first cycle e_1 belongs to the multi-crossing a . Hence, we start the numbering here: the gray over strands in multi-crossing a will have the indices 0, 2, 4, 6, 8, 10, 12 and 14 and the under strands will have the indexes 1, 3, 5, 7, 9, 11, 13 and 15. We are done with this multi-crossing and the cycle e_1 is finished. We now look at e_2 . Its first strand appears in multi-crossing e , and it is an under strand, so we give the index 16 to the over strand of this crossing and the index 17 to the under strand. The next strands (following the cycles) for e_2 and e_3 already have an index so we continue with the next cycle e_4 . It first appears in multi-crossing a but we already assigned indexes to it, so we skip it. The next multi-crossings we encounter are b (we give the indexes 18 and 20 for over strands and 19 and 21 for under strands), c (with indexes 22, 24, 26 and 28 for over strands and 23, 25, 27 and 29 for under strands) and finally d (with indexes 30, 32, 34 and 36 for over strands and 31, 33, 35 and 37 for under strands). At this point all the strands have an index and we are ready to stitch them.

Stitching

We perform the stitchings by following the cycles. For now, we will not indicate the new name after stitching. In the Python implementation we will perform the stitching $m_{\min(a,b)}^{a,b}$.

The optimized way to stitch all strands together is by first doing the necessary stitchings to recover the multi-crossings and then stitch across multi-crossings. Hence, we will stitch strand 0 to 2, 2 to 4, 4 to 6, \dots , 12 to 14 to get multi-crossing a . Then we stitch strand 21 to 19 to get multi-crossing b , strand 23 to 25, 25 to 27 and 27 to 29 to get multi-crossing c and finally we stitch strand 31 to 33, 33 to 35 and 35 to 37 to get multi-crossing d . This minimizes the size of intermediate results.

We then follow the cycles to stitch across multi-crossings: cycle e_1 is done (there is only one crossing for this cycle). For cycle e_2 we stitch strands 17 to 1, and for cycle e_3 we stitch strands 16 to 3. For e_4 we stitch strand 14 to 18, 18 to 22, 22 to 30 and 30 to 13. For e_5 we stitch strand 20 to 24, 24 to 32 and 32 to 11. For e_6 we stitch 19 to 23, 29 to 28, 28 to 36 and 36 to 7. Finally, for e_7 we stitch strand 15 to 30, 37 to 26, 26 to 34 and 34 to 9. At this point, we have reached all the stitchings indicated by dotted lines in figure 3.19. Now we are ready to compute the image of this tangle under the map φ from chapter 2 (this will be an element of $\Gamma_{\{0,1,2,3,4,5\}}$).

Result

When we run the program on the example graph, we can compute that the image under the map φ is given as in the matrix of Figure 3.20, where $\omega_1 = -t + (t - 1)^2 + 2$. This implies the scalar part of v_2 is 1.

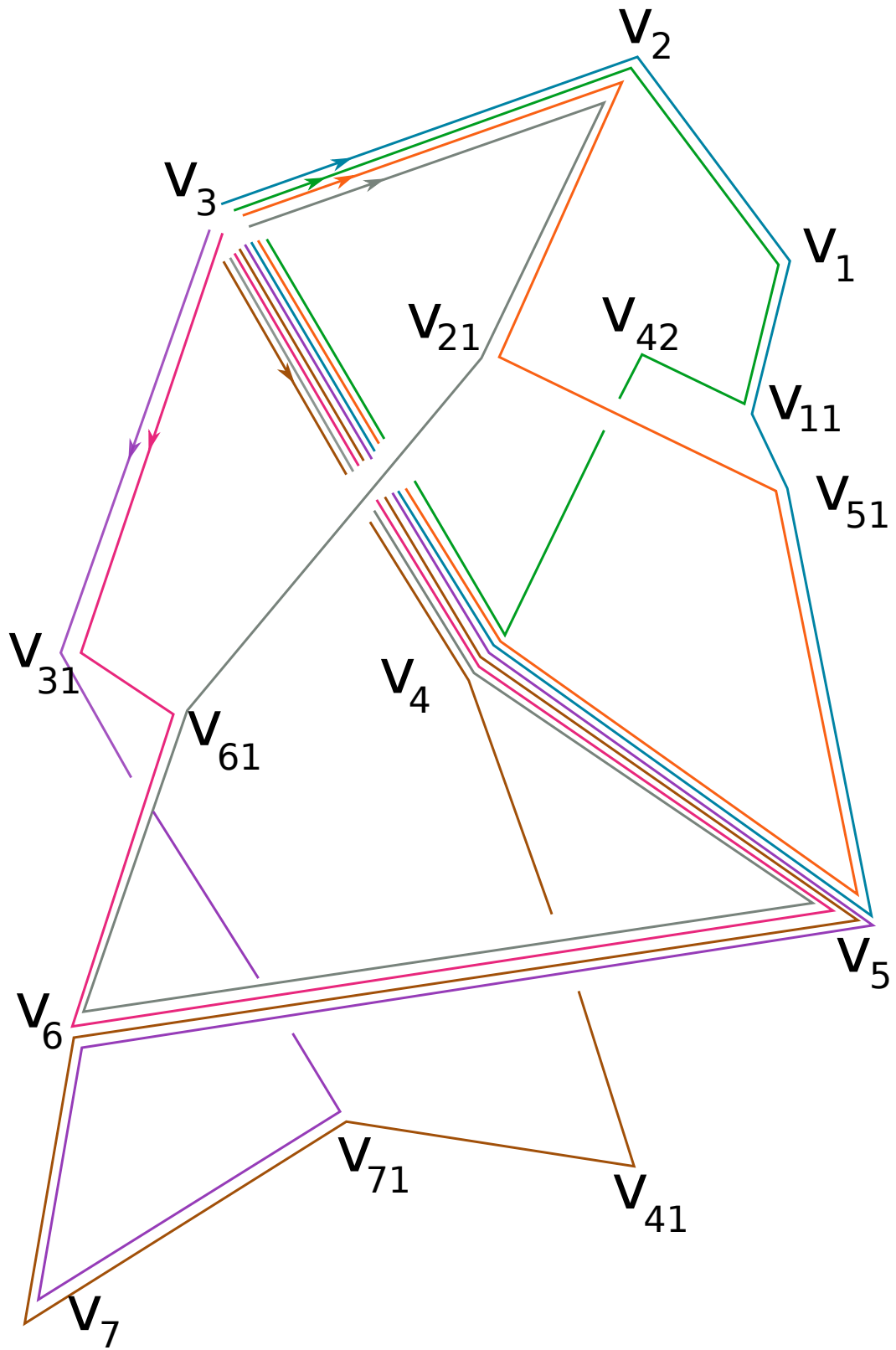


Figure 3.18: Lanes for the graph in figure 3.14.

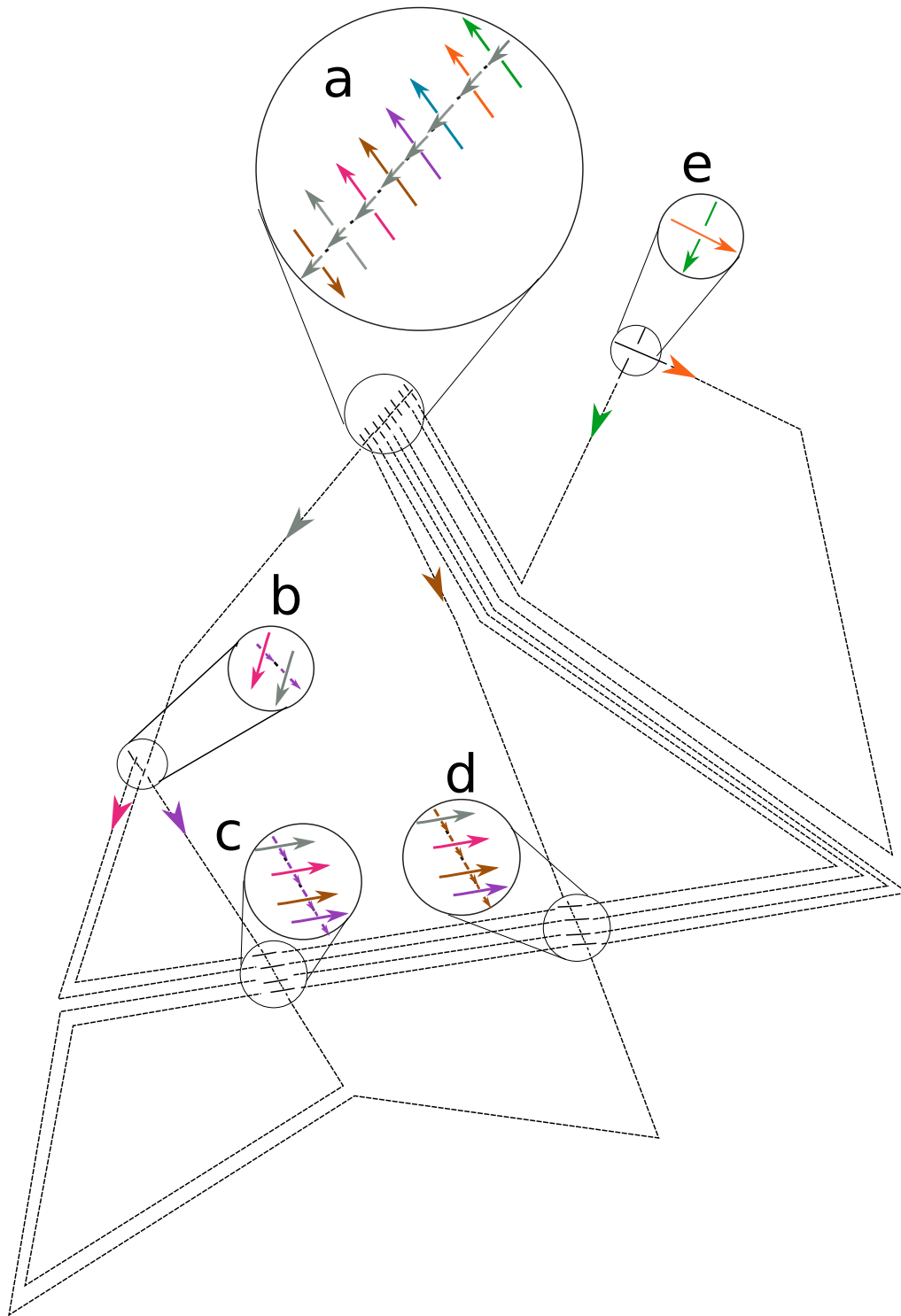


Figure 3.19: Tangle for the graph in figure 3.14.

$$\begin{pmatrix}
\omega_1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
0 & -t+(t-1)^2+2 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & -2t-3(t-1)^2+3 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & -t-2(t-1)^2-1 & -t+(t-1)^2+2 & 0 & 0 & 0 & 0 \\
3 & t-(t-1)^2-1 & t-(t-1)^2-1 & t-(t-1)^2-1 & t-(t-1)^2-1 & t-(t-1)^2-1 & t-(t-1)^2-1 & t-(t-1)^2-1 \\
4 & 0 & 0 & 0 & 0 & -t+(t-1)^2+2 & 0 & t-2(t-1)^2-1 \\
5 & 0 & 0 & 0 & 0 & 0 & -2t+3(t-1)^2+3 & t-4(t-1)^2-1 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 & -3t+7(t-1)^2+4
\end{pmatrix},$$

Figure 3.20: The matrix for the graph of Figure 3.14.

Chapter 4

Computation of v_2 of an example protein and results

In the previous chapter we explained how to get a tangle from a protein. We get it as a collection of crossings and stitching instructions. Therefore, at this point we “only” need to create the corresponding matrix and stitch rows and lanes accordingly. However, this is a challenging task since the smallest proteins (containing 20 amino acids) have a bit more than 300 atoms and between 17 and 22 bonds. Since we get a cycle from every hydrogen and sulfur bond, which translates into components of the tangles, we get at the end a matrix consisting on for example 22×22 entries.

We have six types of crossings in a protein tangle:

$$\begin{aligned} & B - B, br - br, \\ & B - br, br - bo, \\ & B - bo \text{ and } bo - bo. \end{aligned}$$

The type of crossing where more crossings of the type *grid* (as the one in figure 4.1) appear are in the $B - B$ and $B - br$ crossings, and also possibly for $br - br$. We have crossings of the types of figures 2.4 and 2.5 for crossings $br - bo$. The crossing $bo - bo$ is a *simple* crossing, that look like figure 1.2. This is the case because bo edges always have a single lane, br only one lane per connected bo edge, and a B edge has (in the extreme case) a lane for each bo edge in the graph.

4.1 Computation of v_2 for the protein 1L2Y

When we analyze the simplified structure for this protein, we see that there are 22 hydrogen bonds in this protein and no sulfur bonds are present. This means we will consider 22 cycles that will give rise to a tangle with 22 components. This means that the matrix part of the protein tangle is a matrix of size 22×22 .

When we ran the Python code from file 7 for this protein, we see that the value of ω is

$$\omega(t) = -2t + 3(1 - t)^2 + 3.$$

Hence, $k = \frac{1}{2}\omega''(1) = 3$. The code took 3 and a half minutes to complete the calculations for the protein 1L2Y. We do not include the matrix in this thesis for size reasons but the reader can retrieve it by running the code from file 7 from section 5.7 with the entry 1L2Y.

4.2 Further work and improvements to be done

The further work we want to develop is to compute v_2 of a large sample of proteins and discuss with experts in the areas of biology and chemistry to see if we can get insight about the three-dimensional structure of proteins by looking at some properties of their matrices and values of ω .

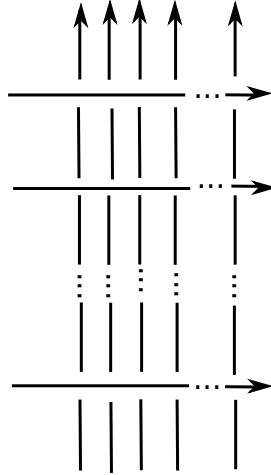


Figure 4.1: A tangle to which we will refer to as *grid crossing*.

In Chapter 2, we only showed lemmas for tangles that look like figure 2.4. However, it would be useful to obtain the expression for v_2 for a grid crossing, as the one in figure 4.1.

Also, there are some optimizations that can be taken into account for the programs in Python. For example, we can optimize the stitching process if we compute v_2 of the blocks coming from the five types mentioned above except for the crossing $bo - bo$, using the results from section 2.3.

We also need an optimization for the code that computes the n -th Taylor polynomial of a function around a point. During benchmarking of the code, we discovered that computing the Taylor polynomial was relatively slow.

An important note: the program in file 4 `intersection_of_lanes` picks an arbitrary cycle and it starts to number the strands it encounters from this. Hence, at this moment the labels of the matrix are not unique. We can give them a unique name like the one mentioned in section 3.2.6 if we need to ¹.

At the moment, the program accepts that the information in the PDB file contains precisely the atoms of the protein/molecule. We need to refine the code of Section 5.7 so it can deal with unusual notation and missing data. For example, hydrogen atoms appear only in the PDB files for proteins whose structure was determined using a specific type of experiment: NMR. A vast number of proteins were analyzed by X-Rays but these do not detect hydrogen atoms, hence we are not able to compute hydrogen bonds and thus we are not able to create a tangle from it (if there are not sulfur bonds either).

¹ The effect of not having this implementation yet is that the matrix we get as a result has the entries permuted with respect to the matrix we would get following the naming algorithm in section 3.2.6.

Chapter 5

Code in Python

In the present chapter we include the code written in Python that performs the analysis of a PDB file and computes at the end an $n \times n$ matrix and a polynomial ω .

The file `graph.py` from section 5.1 provides a class to work with “abstract” graphs. We include methods to add vertices and edges and to switch between representations of a graph: as a `Graph` object, as a dictionary (see definition below) and as a set consisting of pairs of edges. We also include the algorithm to compute a spanning tree of a given graph using Prim’s algorithm. We also have methods to compute a path between any two vertices of a graph and a method to compute the unique path along a spanning tree between two vertices.

The file `graph_position.py` in section 5.2 consists of a class called `GraphPosition`, which is a subclass of the `Graph` class. Here, in addition to the methods for a graph to add vertices and edges, we also create a dictionary where we store the coordinates of the vertices. These positions are necessary for when we create the graph of a certain protein.

In the file `lane_ordering.py` from section 5.3 we perform the analysis and ordering of lanes of a certain edge. For this, we include functions to compute the angle between two vectors. Here we also compute the list of angles of a certain cycle. This will allow us to compare the *behavior* of lanes before and after travelling along a certain edge.

In the file `intersection_of_lanes.py` from section 5.4 we have functions to create a list of intersection of edges. We determine if two edges intersect using parametric equations. For two intersecting segments (a, b) and (c, d) we get the values of two parameters s and t that will help us to order the crossings in the case the edge (a, b) intersects several segments $(c_1, d_1), \dots, (c_n, d_n)$. Also, given a certain pair of intersecting edges, we decide which strand goes over and under by looking at the third coordinate. For this, we look at the parameters s and t and we substitute them in the parametric equation of a line in \mathbb{R}^3 . We sort lanes for every edge by calling the function `sorted_lanes` from the file `lane_ordering.py` from section 5.3.

Given a cycle, we travel along it. For each edge, we are going to name both strands of each crossing we encounter as follows: the over strand is given an even number n and the under strand is given an odd number $n + 1$. In the fourth file we also find the function `order_crossing_labels` that sorts all the strands we find along an edge. The function `stitch_lanes_together_for_crossings` receives the data of a graph, cycles and a list of crossings and assigns an index to each strand we encounter using the rule we just described (over is an even number, under is an odd number). Then we travel along a cycle and we “give the instruction” to attach the strands we find along the way (given by the function `order_crossing_labels`). In this code we also get the sign of a crossing in the function `get_crossing_directions` by looking at the angle between a vector in the same direction as the over strand and a vector in the same direction as the under strand. If the angle is negative we have a positive crossing and if the angle is positive we get a negative crossing.

In the file `protein_structure.py` from section 5.5 we create the graph of a protein, using the methods to find the covalent, hydrogen and sulfur bonds from the chapter 3 and we refine this graph in the method `simplified_protein` also according to chapter 3. In this code we also include two extra functions (`get_center` and `recenter`) that will ensure the rendering of the protein is in the middle of the screen when we draw a protein using PyGame.

The file `computation_v2_c2.py` from section 5.6 performs the calculation of v_2 of a tangle. The

way to implement the stitching rule in Python is the following. Suppose that we want to perform the stitching of strand a with strand b . Then we compute the product of column b by the row a , we then divide every entry by $1 - \beta$, or in our simplified stitching rule for v_2 , we multiply by $1 + \beta + \beta^2$. Then we delete column b and row a and after this step we need to rearrange the columns and/or rows in our result so that the entry $\gamma + \alpha\delta(1 + \beta + \beta^2) \pmod{s^3}$ is the entry with indexes $(\min(a, b), \min(a, b))$ of the final matrix. We implement this rearrangement in Python as follows:

- If $a < b$ then we replace row a by a copy of row b and then we delete the b -th column and b -th row, since we want to stay with the $\min(a, b)$.
- If $a > b$ then we replace column b by column a and then we delete the a -th column and the a -th row, since we want to only keep the $\min(a, b)$.

We keep track of the changes in names of strands after some stitchings in the dictionary called `tangle_map`. For example, assume we have the following names for the strands [T1, T2, T3, T4, T5, T6] to which we assign the indices [0, 1, 2, 3, 4, 5] respectively. Suppose we want to stitch T1 with T3 (by default our Python function assigns to the stitching of the strands a and b the name $\min(a, b)$, i.e. it performs $m_{\min(a,b)}^{a,b}$). Then we can summarize the function that updates the names and indexes in the following table:

name	row	row>2?	result1	row==2?	result
T1	0	-	-	-	-
T2	1	-	-	-	-
T3	2	-	-	Yes	<code>tangle_map[T3]=0</code> .
T4	3	Yes	<code>tangle_map[T4]=2</code>	-	-
T5	4	Yes	<code>tangle_map[T4]=3</code>	-	-
T6	5	Yes	<code>tangle_map[T4]=4</code>	-	-

We get as a result the following dictionary:

$$\{\text{T1:0, T2:1, T3:0, T4:2, T5:3, T6:4}\},$$

which is precisely what we were after. Now we explain how to implement $\pmod{s^3}$ in Python. We do this inside the method `v2._stitching(self, i, j)` by taking the expansion in Taylor series around 1 up to degree 2. For the stitching rule (2.7) we do the same.

In this file we also included a function to create the Alexander polynomial of a tangle consisting of a single crossing by taking into account the names of the over and under strands and the sign of the crossing, and a function to compute the matrix of a disjoint union of tangles. We use the information coming from the file `intersection_of_lanes.py` from section 5.4 about the order in which the stitchings of strands need to be done, and also about the sign of the crossings.

We iterate over the list of strands to be stitched. If this strand does not appear in the matrix yet we add the crossing using a disjoint union and the corresponding 2×2 matrix. All crossings that do not get stitched are added at the end.

In the file `computation_v2_protein.py` from section 5.7 we include the necessary lines to run all the previous programs to obtain v_2 of a protein. This program receives the name of a protein (for example 1UAO) and it computes the simplified structure and the canonical spanning tree of the given protein we defined in chapter 3. It computes all the cycles coming from hydrogen and sulfur bonds, computes the list of crossings with their respective signs and names of strands, determines which strands need to be stitched together and finally gives all this information to the program `computation_v2_c2` to build the corresponding matrix and stitch strands together.

The last two files are called `draw_protein_xy.py` and `draw_protein.py` and appear in sections 5.8 and 5.9 respectively. These files draw a protein using PyGame. The first one presents the projection of the (simplified) structure in the xy -plane and the second rotates this projection to get a better idea of how this protein sits in the three dimensional space. By changing a line in this code we can get the picture for the non-simplified graph of a protein. For this, we remove the `‡` symbol at the last line in the second `for` loop inside the `while` loop and we write it at the beginning of the last line of the third `for` loop inside the `while` loop.

The steps to transform a graph into a picture in the computer screen of 640 times 480 pixels are as follows. First, if the center of our original screen S_1 is (x_0, y_0) , and we want it to be $(0, 0)$, then we consider $(x - x_0, y - y_0)$ for each one of the points $(x, y) \in S_1$. Then, we also need to multiply these coordinates to make the picture look bigger inside a bigger environment (if the new picture is supposed to be 640 times 480 pixels), so we do

$$\left((x - x_0) \cdot \frac{\text{width}}{14}, -(y - y_0) \cdot \frac{\text{height}}{12} \right),$$

in order to get the default zoom levels.

Since we want the new center at $(320, 240)$ the new coordinates are:

$$\left((x - x_0) \cdot \frac{\text{width}}{14} + 320, -(y - y_0) \cdot \frac{\text{height}}{12} + 240 \right).$$

Now, if we would also like to consider a rotating image, we need to consider a rotation matrix. We will use the following:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x \\ y \cos \theta - z \sin \theta \\ z \end{pmatrix}.$$

So the final screen coordinates is $\left((x - x_0) \cdot \frac{\text{width}}{14} + 320, -(y \cos \theta - z \sin \theta - y_0) \cdot \frac{\text{height}}{12} + 240 \right)$ for a 3d point (x, y, z) .

5.0.1 Useful Python data structures for this chapter

When we described the code for this thesis, we mentioned the concepts of *dictionary*, *list* and *set*. Here we include more explanation about them to facilitate the reading of the code. In Python, there are several data types for collections. The types we are going to use are *dictionary*, *list* and *set*. A dictionary is a collection of data which is unordered, mutable and indexed. We write it enclosed in curly brackets. It contains *keys* (in the next example these are 'name', 'last_name' and 'age' and *values* (in the example these are 'Ana', 'Garcia' and 32). For example:

```
dictionary1={'name':'Ana',
            'last_name':'Garcia',
            'age': 32} .
```

A list is ordered and mutable. We write it inside square brackets, for example:

```
list1=['green', 'red', 'pink', 'purple'] .
```

We access its elements indicating the index, for example:

```
list1[0]
```

returns 'green'. A set is a collection that is unordered and unindexed. We write it inside curly brackets, for instance:

```
set1={'apple', 'pear', 'orange'} .
```

It is not possible to access elements in a set by referring to an index nor change the original items. However, it is possible to add new elements.

Now, having said all this we present the code in Python:

5.1 File 1: graph.py

```
from collections import defaultdict
import math
import numpy as np

class Graph:
    def __init__(self):
        """Make an empty graph."""
        self.edges = defaultdict(list)

    def __or__(self, other):
        """This gives us the union of the edges of two graphs."""
        union_graph = Graph()
        return (union_graph.add_edges(self.edges)).add_edges(other.edges)

    def add_edges(self, graph_dict):
        """Given a dictionary of lists of neighbors, add each edge to the graph."""
        for vertex, neighbors in graph_dict.items():
            for neighbor in graph_dict[vertex]:
                edge = {'from': vertex, 'to': neighbor}
                edge = vertex, neighbor #We represent an edge as a tuple.
                self.add_edge(vertex, neighbor)
                assert self.has_edge(vertex, neighbor)
        return self

    def add_edge(self, vertex_1, vertex_2):
        """This function add an edge between two vertices.
        graph.has_edge(v1, v2) is True exactly if graph.add_edge(v1, v2) has been called.
        """
        self.edges[vertex_1].append(vertex_2)

    def add_edge_pairs(self, pairs_of_vertices): #pairs_of_vertices is a list.
        """This method receives a list of pairs of vertices and changes the graph object
        by adding an edge for this pair."""
        for pair in pairs_of_vertices:
            self.add_edge(pair[0], pair[1])

    def has_edge(self, vertex_1, vertex_2):
        """This function receives two vertices and decides if there is an edge between
        them in the object (graph) we are calling."""
        if vertex_1 in self.edges: #We check if a dict has a given key.
            if vertex_2 in self.edges[vertex_1]:
                return True
            else:
                return False
        else:
            return False

    def generate_edges(self):
        """This function takes a graph object and transforms its representation into
        a list of pairs (vertex1, vertex2).
        (v1, v2) is an element of the list if and only if there is an edge from v1 to v2.
        """
```

```

edges = set()
for vertex in self.edges:
    for neighbor in self.edges[vertex]:
        edges.add((vertex, neighbor))
return edges

def generate_edges_oneway(self):
    """This function takes a graph object and transforms its representation into
    a list of pairs (vertex1,vertex2).

    (v1, v2) is an element of the list if and only if there is an edge from v1 to v2,
    and if there is also an edge (v2, v1) then we only add one of them.
    """
    edges = set()
    for vertex in self.edges:
        for neighbor in self.edges[vertex]:
            # We say an edge goes from the 'smaller' vertex to the 'larger'.
            v1, v2 = min(vertex, neighbor), max(vertex, neighbor)
            edges.add((v1, v2))
return edges

def Prim_alg(self): #This gives the spanning tree in both directions.
    A = dict()
    B = set()
    x=next(iter(self.edges.keys()))
    tree = Graph() #tree is a Graph object.
    while True:
        # visit x.
        B.add(x)
        for neighbor in self.edges[x]:
            if neighbor not in B:
                A[neighbor] = x, neighbor
        # go to the next x.
        # invariant will hold here: tree is a spanning tree for the vertices in B.
        # invariant will hold here: the neighbors of vertices in B that are not in B
        # itself are keys of A. At each iteration, a key of A becomes an element
        # of B. The condition is that A is not empty.
        try:
            x, edge = next(iter(A.items()))
        except StopIteration:
            # There is no next x, so we are finished with the tree.
            break

        del A[x] #A has all the vertices neighbors to the tree but not in the tree.
        tree.add_edge(edge[0],edge[1]) #we add the first vertex in the edge
        # and the second vertex in the edge.
    return tree.add_inverse_direction()

def add_inverse_direction(self): #each edge is a tuple.
    graph_2 = Graph()
    for edge in self.generate_edges(): #this gives us the graph as a set with tuples.
        graph_2.add_edge(edge[1],edge[0])
    return self | graph_2 # | means union

def all_vertices(self):
    """This method gives us all the vertices in a graph."""

```

```

return self.edges.keys()

def find_all_paths(self, source):
    """Given a source vertex, gives a dictionary mapping any vertex to a path
    from the source to that vertex."""
    result = {source: [source]}
    neighbors = {} #we need to consider the vertices at distance 1 from the source.
    for neighbor in self.edges[source]:
        neighbors[neighbor] = source

    # invariant: every path in result.values() is a valid path from
    # the source to the corresponding key.
    # invariant: neighbors has keys: vertices with distance 1 from result.keys()
    # and the corresponding value: a vertex that has distance 1 to the key.
    while neighbors: #while neighbors is not empty.
        unvisited_vertex, visited_vertex = next(iter(neighbors.items())) # this line
        # chooses a random key and its respective value.
        path = result[visited_vertex] + [unvisited_vertex]
        result[unvisited_vertex] = path
        # restore the second invariant.
        del neighbors[unvisited_vertex]
        for vertex in self.edges[unvisited_vertex]:
            if vertex in result:
                continue
            if vertex in neighbors:
                continue
            neighbors[vertex] = unvisited_vertex
    return result

def find_path(self, source, target):
    """This method finds any path from the source to the target.

    Note: in the case of a ST, this path along the ST is unique.
    """
    all_paths = self.find_all_paths(source)
    return all_paths[target]

def find_unique_path_alongST(self, root, start, end):
    """This method receives a graph, a root vertex and an edge
    (not in ST) (written as start, end) and returns the unique
    path root-start-end-root along the spanning tree given by Prim_alg."""
    ST = self.Prim_alg()
    p1 = ST.find_path(root, start)
    p2 = ST.find_path(end, root)
    return p1+p2

def all_cycles(self, root): #iteration over edges
    """This method receives a graph and a root, computes one spanning tree
    given by Prim_alg and returns a dictionary. Key:edge, value:
    unique path (actually it's a cycle) along the given spanning tree.
    """
    Edges_out_of_tree = self.generate_edges()-self.Prim_alg()
    dict_all_cycles = defaultdict(list)
    for edge in Edges_out_of_tree:
        dict_all_cycles[edge] = self.find_unique_path_alongST(root, edge[0], edge[1])
    return dict_all_cycles

```

5.2 File 2: graph_position.py

```
from collections import defaultdict
import numpy as np
import prody
from protein_structure import ProteinStructure
from graph import Graph

#self.vertices is a dictionary.

class GraphWithPosition(Graph): #GraphWithPosition is a subclass from the Graph class.
    """Graph that stores a position for each vertex.

    When you create an edge between two vertices,
    you need to have assigned them a position before.
    """

    def __init__(self):
        """Make an empty graph."""
        super().__init__() # does Graph.__init__(self) but also works if our class hierarchy
        #is more complicated.
        self.vertices = {}

    def add_edge(self, vertex_1, vertex_2):
        """This function add ans edge between two vertices.

        graph.has_edge(v1, v2) is True exactly if graph.add_edge(v1, v2)
        has been called.

        When you create an edge between two vertices,
        you need to have assigned them a position before.
        """
        assert self.has_vertex(vertex_1)
        assert self.has_vertex(vertex_2)
        super().add_edge(vertex_1, vertex_2)

    def add_vertex(self, vertex_id, position):
        """Add a vertex to the graph with the given position.

        You cannot add the same vertex id twice.
        """
        assert not self.has_vertex(vertex_id)
        self.vertices[vertex_id] = position

    def add_vertices(self, vertex_dict):
        """Given a dictionary that maps vertex id to position, add each vertex to this
        graph."""
        for vertex_id, position in vertex_dict.items():
            self.add_vertex(vertex_id, position)

    def has_vertex(self, vertex_id):
        """Give a boolean: whether the vertex id corresponds to a vertex in this graph."""
        return vertex_id in self.vertices

    def get_position(self, vertex_id):
        return self.vertices[vertex_id]
```

5.3 File 3: lane_ordering.py

```
import numpy as np
import collections as col
import sympy as sp
import math

from graph_position import GraphWithPosition

# We will write cycles as a list of names of vertices,
# where the first and the last elements of a list are the same:

def is_a_cycle(cycle):
    return isinstance(cycle, list) and cycle[0] == cycle[-1] # we check the first and
    # last elements coincide.

def angle_v_w(v,w): #vectors in 2d.
    """We receive two vectors and we compute the angle between them,
    the result is given in degrees. The vectors are first projected onto
    their first 2 coordinates.

    The result is never negative. For a negative angle, use signed_angle.
    """
    v = v[:2] # Take the first 2 coordinates.
    w = w[:2]

    A_in_rad = math.acos(np.dot(v,w)/ (np.linalg.norm(v) * np.linalg.norm(w)))
    # Here we compute the arc-cosine, we get the result in radians
    #math.cos() returns the cosine of x radians.
    A_in_deg = (A_in_rad *180)/math.pi
    return A_in_deg

def perpendicular(a):
    perp_a = np.zeros_like(a)
    perp_a[0]=-a[1]
    perp_a[1]=a[0]
    return(perp_a)

def signed_angle(w_1, w_2):
    theta_no_sign = angle_v_w(w_1,w_2)
    return np.sign(np.dot(perpendicular(w_2),w_1))*theta_no_sign

def angle_between_edges(graph, v1, v2, v3):
    """Compute the angle between the edges (v1, v2) and (v2, v3)."""
    direction_1 = graph.vertices[v2] - graph.vertices[v1]
    direction_2 = graph.vertices[v3] - graph.vertices[v2]
    return signed_angle(direction_1, direction_2)

def angle_between_crossing_edges(graph, v1, v2, v3, v4):
    """Compute the angle between the edges (v1, v2) and (v3, v4)."""
    direction_1 = graph.vertices[v2] - graph.vertices[v1]
    direction_2 = graph.vertices[v4] - graph.vertices[v3]
    return signed_angle(direction_1, direction_2)

def angles_for_cycle(graph, cycle):
    """Give the angles for adjacent vertices in a cycle.
```

```

We don't give the angle at the first/last vertex.
"""
assert is_a_cycle(cycle)
angles = []
# Iterate over triplets of adjacent vertices in the cycle.
# Note that the first and last elements of the cycle are identical,
# so we have all angles except for the one at the first/last vertex.
for i in range(2, len(cycle)):
    v1 = cycle[i-2]
    v2 = cycle[i-1]
    v3 = cycle[i]
    angles.append(angle_between_edges(graph, v1, v2, v3))
return angles

def lanes_per_edge(graph, cycles):
    """Give a dictionary that maps edges (i.e. pairs of vertices)
    to all lanes that go through that edge in either direction.

    Here, a lane is a triplet (cycle_id, index, same_dir), where:
    if same_dir then cycles[cycle_id][index] == v1 and
    cycles[cycle_id][index + 1] == v2, else cycles[cycle_id][index] == v2
    and cycles[cycle_id][index + 1] == v1.
    Equivalently, the index is the number of the edge (or the reverse edge)
    within the cycle, and same_dir says whether it is the forward edge
    or the reverse edge.
    """

    lane_dict = col.defaultdict(set)
    for cycle_id, vertices in cycles.items():
        for i in range(0, len(vertices)-1):
            v1 = vertices[i]
            v2 = vertices[i+1]
            lane_dict[(v1, v2)].add((cycle_id, i, True))
            lane_dict[(v2, v1)].add((cycle_id, i, False))
    return lane_dict

def forward_angles(graph, cycle, index):
    return angles_for_cycle(graph, cycle)[index:]
def backward_angles(graph, cycle, index):
    return (angles_for_cycle(graph, cycle)[:index])[:-1]
def negate_angles(angle_list):
    return [-1 * angle for angle in angle_list]

def lane_order_key(graph, cycles, v1, v2, lane):
    """Give a value for the lane in the edge, such that sorting on
    this value orders the lanes in the geometric order.

    (Geometric means from left to right in the diagram).
    """

    # We will sort on the forward angles: all the angles, starting
    # from the end of this edge, and if those are the same on the backward
    # angles: all the angles up to the beginning of this edge.

    cycle_id, index, same_dir = lane
    cycle = cycles[cycle_id]
    if same_dir:

```

```

    assert v1 == cycle[index] and v2 == cycle[index+1]
    # We are going forwards.
    return forward_angles(graph, cycle, index), backward_angles(graph, cycle, index)
else:
    assert v2 == cycle[index] and v1 == cycle[index+1]
    # We are going backwards, so we negate all angles in the cycle.
    # Additionally, the backward angles and forward angles change places.
    return negate_angles(backward_angles(graph, cycle, index)),
           negate_angles(forward_angles(graph, cycle, index))

def sorted_lanes(graph, cycles, v1, v2):
    """Given the name of two vertices, return the sorted list of lanes
    in the corresponding edge."""
    unsorted_lanes = lanes_per_edge(graph, cycles)[(v1, v2)] #lanes_per_edge is a dictionary.
    def key(lane):
        return lane_order_key(graph, cycles, v1, v2, lane)
    return sorted(unsorted_lanes, key=key)

```

5.4 File 4: intersection_of_lanes.py

```

import numpy as np
import collections as col
import sympy as sp
import math

from graph_position import GraphWithPosition

import lane_ordering

def perpendicular(a):
    """Return a vector orthogonal to a, within the xy plane.

    Note that a should be nonzero within this plane, or we return the zero vector.
    """
    perp_a = np.zeros_like(a)
    perp_a[0] = -a[1]
    perp_a[1] = a[0]
    return(perp_a)

def find_crossings(a1, a2, b1, b2):
    """Get the distance along the line (a1, a2) and (b1, b2) where
    they intersect. If the lines are parallel, we raise a ZeroDivisionError.

    If the vectors are not 2d, we give the result projected onto the first two coordinates.
    """
    u = a2-a1
    v = b2-b1
    w = a1-b1
    # Note that we don't have to project because perpendicular(u)
    # is already in the xy plane.
    perp_u = perpendicular(u)
    perp_v = perpendicular(v)

    num_s = np.dot(-perp_v, w)
    denom_s = np.dot(perp_v, u)

```



```

num_t = np.dot(perp_u,w)
denom_t = np.dot(perp_u,v)

if denom_s == 0 or denom_t == 0:
    raise ZeroDivisionError()

s = num_s/denom_s.astype(float)
t = num_t/denom_t.astype(float)

return s,t

def intersection_between_segments(x1,x2,y1,y2):
    """It receives four points x1, x2, y1, y2 and decides if there is
    an intersection between the segments x1->x2 and y1->y2.

    Note this only considers the first two coordinates, so if we give 3d-coordinates,
    it will give the intersection of their projections w.r.t. z-coordinate
    (we project to the xy-plane).
    """
    s, t = find_crossings(x1,x2,y1,y2)
    if 0 < s < 1 and 0 < t < 1:
        return True, s, t
    else:
        return False, None, None

def intersect_edges_with_distances(graph): #iteration over edges
    """This method receives a list of edges (of a grap proj) and for
    every pair of edges, it decides if they intersect.

    It returns a dictionary where the keys are edges and values
    are lists of tuples (s, t, edge), where s and t indicate the distance
    along the two edges of the intersection, and edge is the other edge
    that intersects with the key.

    The tuples are ordered in increasing values of s.
    """
    intersections = col.defaultdict(list)
    crossings = set()
    for edge_1 in graph.generate_edges(): # The graph as a set with tuples.
        for edge_2 in graph.generate_edges_oneway(): #graph.vertices is a dictionary
            # If the edges share a vertex, skip them (because that will be their intersection)
            if set(edge_1) & set(edge_2):
                continue
            try:
                intersects, s, t = intersection_between_segments(graph.vertices[edge_1[0]],
                    graph.vertices[edge_1[1]], graph.vertices[edge_2[0]], graph.vertices[edge_2[1]])
            except ZeroDivisionError:
                continue
            if intersects:
                intersections[edge_1].append((s, t, edge_2))
                crossings.add(frozenset({(edge_1[0], edge_1[1]), (edge_2[0], edge_2[1])}))
    for key, intersection_list in intersections.items():
        intersections[key] = sorted(intersection_list)
    return (intersections, crossings)

```

```

def intersect_edges(graph):
    """This method receives a list of edges (of a graph proj) and for every
    pair of edges, it decides if they intersect. It returns a dictionary where
    the keys are edges and values a list of edges that intersect with
    the key (which is an edge)."""
    intersections, crossings = intersect_edges_with_distances(graph)
    for key, intersection_list in intersections.items():
        intersections[key] = [edge for s, t, edge in intersection_list]
    return intersections, crossings

def z_ordered_intersections(graph):
    """We use here that the intersect_edges function also works when
    projecting to the xy-plane."""
    intersections, crossings = intersect_edges_with_distances(graph)
    new_intersections = col.defaultdict(list)
    for (index1, index2) in intersections.keys():
        for s, t, (v,w) in intersections[(index1, index2)]:
            #intersections[(index1, index2)] is a list
            a1 = graph.get_position(index1)
            a2 = graph.get_position(index2)
            b1 = graph.get_position(v)
            b2 = graph.get_position(w)
            z1 = (a1 + s*(a2-a1))[2]
            z2 = (b1 + t*(b2-b1))[2]
            assert z1 != z2, "Intersection between {},{} ({} and {},{} ({} instead of crossing"
                .format(index1, index2, s, v, w, t)
            if z1 > z2:
                new_intersections[(index1, index2)].append((v,w))
            if z1 < z2:
                new_intersections[(v, w)].append((index1,index2))
    return new_intersections

def sorted_lanes_for_crossing(graph,cycles,v01,v02,w1,w2):
    """Takes the position of four vertices and calls sorted."""
    return (lane_ordering.sorted_lanes(graph, cycles, v01, v02),
            lane_ordering.sorted_lanes(graph, cycles, w1, w2))

def sorted_crossings_for_edge(v01,v02,w1,w2):
    s1 = find_crossings(v01,v02,w1,w2)[1]

def crossings_per_lane(graph, cycles, sorted_lanes):
    """Give a dictionary lane: list of crossings from start to end."""
    intersections, _ = intersect_edges(graph)
    crossing_labels = generate_crossing_labels(graph, cycles, intersections, sorted_lanes)
    return order_crossing_labels(graph, cycles, intersections, crossing_labels, sorted_lanes)

def all_sorted_lanes(graph, cycles):
    sorted_lanes = {}
    for edge in graph.generate_edges():
        if (edge[0], edge[1]) in sorted_lanes:
            continue
        sorted_lanes[edge[0],edge[1]] = lane_ordering.sorted_lanes(graph,cycles,edge[0],edge[1])
        sorted_lanes[edge[1],edge[0]] = lane_ordering.sorted_lanes(graph,cycles,edge[1],edge[0])
    return sorted_lanes

```

```

def crossings_per_lane_3d(graph, cycles):
    """Give a dictionary lane: list of crossings from start to end.

    The even numbered crossings are above the corresponding odd numbered
    crossings.
    """

    # Optimization: compute sorted lanes m times and read it
    # m * (total number of lanes)**2.
    print("sorting lanes...")
    sorted_lanes = all_sorted_lanes(graph, cycles)

    intersections, _ = intersect_edges(graph)
    intersections_3d = z_ordered_intersections(graph)
    print('intersections:', intersections_3d)
    crossing_labels = generate_crossing_labels(graph, cycles, intersections_3d, sorted_lanes)
    print('crossing labels:', crossing_labels)
    return order_crossing_labels(graph, cycles, intersections, crossing_labels, sorted_lanes)

#Plan: We need to give all the intersections to the second loop in
# crossings_per_lane but only the (over,under) to the first loop. We need
# to put these 2 loops in different functions and give the output of
# the 1st function to the 2nd one.

def generate_crossing_labels(graph, cycles, intersections, sorted_lanes):
    """If the intersections are (over, under) then the even strands
    with i.d. n go over the strands with i.d. n+1.

    :return: A dict sending each pair of lanes to the id of the crossing
    corresponding to that pair.
    """

    # for all pairs of intersecting edges
    # for all lanes in these two edges
    # get a new name for the crossing
    # record for the two lanes that we have a crossing

    print("recording crossings...")
    # We iterate over all unordered pairs of intersecting unordered edges.
    crossing_id = 0
    crossing_labels = dict() # Maps (lane1, lane2) to crossing_id
    for edge1 in graph.generate_edges():
        edge_intersections = intersections[edge1]
        for edge2 in edge_intersections:
            for lane1 in sorted_lanes[edge1]:
                lane1 = lane1[0], lane1[1]
            for lane2 in sorted_lanes[edge2]:
                lane2 = lane2[0], lane2[1]
            if (lane1, lane2) not in crossing_labels:
                crossing_labels[lane1, lane2] = crossing_id + 0
                crossing_labels[lane2, lane1] = crossing_id + 1
            crossing_id += 2
    return crossing_labels

def order_crossing_labels(graph, cycles, intersections, crossing_labels, sorted_lanes):
    """Note that if (edge1, edge2) is in the intersections, then we also need

```

```

    (edge2, edge1) in there."""
    # Now we need to order the crossings per lane.
    # For this, we can use the order of the edge's intersections.
    crossings_ordered = col.defaultdict(list)
    # We do the loop again, but we now know that the crossing_id is unique
    # for forwards and backwards.
    for edge1 in graph.generate_edges():
        edge_intersections = intersections[edge1]
        for edge2 in edge_intersections:
            lanes2 = sorted_lanes[edge2]
            if lane_ordering.angle_between_crossing_edges(graph, *edge1, *edge2) > 0:
                lanes2.reverse() #we reverse the list
            for lane1 in sorted_lanes[edge1]:
                lane1_oneway = lane1[0], lane1[1]
                # Note that this might be reversed, if the angle between edge1 and edge2
                # requires it.
                for lane2 in lanes2:
                    lane2_oneway = lane2[0], lane2[1]
                    crossings_ordered[lane1].append(crossing_labels[lane1_oneway, lane2_oneway])

    return crossings_ordered

def stitch_lanes_together(graph, cycles): #this gives us a list of pairs
#of crossings to be stitched together
    crossings = crossings_per_lane(graph, cycles)
    return stitch_lanes_together_for_crossings(graph, cycles, crossings)

def stitch_lanes_together_for_crossings(graph, cycles, crossings):
    to_be_stitched = [] # list of pairs (head, tail) of crossing ids to be stitched
    for cycle, vertices in cycles.items():
        cycle_length = len(vertices) - 1 # total number of lanes
        previous = None
        previous_lane = None
        for lane_number in range(0, cycle_length):
            lane = cycle, lane_number, True # consider only forwards lanes
            for crossing in crossings[lane]:
                if previous is not None:
                    assert previous_lane[0] == lane[0] # only stitch within the cycle
                    #print(previous_lane, lane, previous, crossing)
                    to_be_stitched.append((previous, crossing))
                previous = crossing
                previous_lane = lane
    return to_be_stitched

def get_crossing_directions(graph, cycles, crossings_per_lane):
    """Generate a dictionary that maps a crossing id to a tuple (over, under, is_positive)."""
    # First we compute the direction vector of each strand, so we can compare them later.
    strand_directions = {}
    for lane, crossings in crossings_per_lane.items():
        cycle, index, forward = lane
        # Ensure we get the correct direction for this lane:
        # we only want to stitch cycles in the forward direction.
        # a backward edge has the same vertices but the other order.
        if not forward:
            continue
        v1 = cycles[cycle][index]

```

```

v2 = cycles[cycle][index + 1]
direction = graph.get_position(v2) - graph.get_position(v1)

for strand_id in crossings:
    strand_directions[strand_id] = direction

result = {}

for strand_a in strand_directions:
    # strand_1 goes over strand_2, which we can find out using their id's
    if strand_a % 2 == 0:
        strand_1 = strand_a
        strand_2 = strand_a + 1
    else:
        strand_1 = strand_a - 1
        strand_2 = strand_a
    dir_1 = strand_directions[strand_1]
    dir_2 = strand_directions[strand_2]
    # The crossing is positive if the angle from strand_1 to strand_2 is to the left.
    sign = np.dot(dir_2, lane_ordering.perpendicular(dir_1))
    is_positive = sign > 0
    result[strand_a] = (strand_1, strand_2, is_positive)

return result

```

5.5 File 5: protein_structure.py

```

#We get covalent bonds following Bart van Munster thesis:
covalentradii={'H': 0.352,
               'C': 0.825,
               'N': 0.781,
               'O': 0.693,
               'S': 1.133,
               'P': 1.221}
max_radius = 2 * 1.221 # Maximum radius between two atoms

def get_sulfur_from_resnum(protein, resnum):
    selection = protein.select('resnum {}'.format(resnum)).select('element S')
    assert len(selection) == 1 # We assume that there are no residues with multiple sulfurs.
    return selection[0]

import collections as col
import numpy as np
import prody
import re #regular expression

class ProteinStructure:
    def __init__(self, protein_name):
        self.edges = col.defaultdict(list)
        self.bond_type = dict()
        self.protein = prody.parsePDB(protein_name)

        # Covalent bonds
        for atom_1 in self.protein:
            for atom_2 in self.protein.select('within {} of index {}'.format(

```

```

.format(max_radius, atom_1.getIndex())):
    elem1 = atom_1.getElement()
    elem2 = atom_2.getElement()
    try:
        sum1_2 = covalentradii[elem1]+covalentradii[elem2]
    except KeyError:
        continue
    c1 = atom_1.getCoords() #c1 is an array with the coords of atom_1.
    c2= atom_2.getCoords() #c2 is an array with the coords of atom_2.
    D=np.sqrt(np.dot(c1-c2,c1-c2))
    if D < sum1_2 and D>0:
        self.add_edge(atom_1, atom_2)

# Sulfur bonds
# Ensure we have the PDB file for parsing.
prody.fetchPDB(protein_name, compressed=False)
with open('{}.pdb'.format(protein_name)) as file_protein:
    for line in file_protein:
        if line.startswith('SSBOND'):
            fields =re.split('\s+', line)
            atom_1 = get_sulfur_from_resnum(self.protein, int(fields[4]))
            atom_2 = get_sulfur_from_resnum(self.protein, int(fields[7]))
            # Bonds are given in one direction, but we store edges in two directions.
            # Thus we add an edge in both directions.
            self.add_edge(atom_1, atom_2, 'SS')
            self.add_edge(atom_2, atom_1, 'SS')

# Hydrogen bonds
for atom_1, atom_2 in self.iter_neighbors():
    elem1 = atom_1.getElement()
    elem2 = atom_2.getElement()
    if elem1 not in {'N', 'O', 'S'} or elem2 != 'H':
        continue
    for atom_3 in self.protein:
        elem3 = atom_3.getElement()
        if elem3 not in {'N', 'O', 'S'}:
            continue
        if self.has_edge(atom_2, atom_3):
            continue
        c2 = atom_2.getCoords() #c2 is an array with the coords of atom_2
        c3 = atom_3.getCoords() #c3 is an array with the coords of atom_3
        D2_3=np.sqrt(np.dot(c2-c3,c2-c3))
        if D2_3 < 3:
            self.add_edge(atom_2, atom_3, 'HB')
            self.add_edge(atom_3, atom_2, 'HB')

def add_edge(self, atom_1, atom_2, bond_type=None):
    # Only adds an edge in one direction!
    self.edges[atom_1.getIndex()].append(atom_2)
    if bond_type:
        self.bond_type[atom_1.getIndex(), atom_2.getIndex()] = bond_type

def has_edge(self, atom_1, atom_2):
    """"Did we add an edge from atom_1 to atom_2?"""""
    return atom_2 in self.edges[atom_1.getIndex()]

def iter_neighbors(self):

```

```

        """Iterate over all pairs of atoms that have a bond.
        This function returns a generator, that we can use in a list, for example."""
    for index_1, neighbors in self.edges.items():
        for atom_2 in neighbors:
            yield self.protein[index_1], atom_2

def get_center(self): # This will become useful when we draw the protein in PyGame.
    """Return the point that is in the center of this protein."""
    x_min, y_min, z_min = x_max, y_max, z_max = self.protein[0].getCoords()
    for atom in self.protein:
        x, y, z = atom.getCoords()
        x_min = min(x_min, x)
        y_min = min(y_min, y)
        z_min = min(z_min, z)
        x_max = max(x_max, x)
        y_max = max(y_max, y)
        z_max = max(z_max, z)
    return (x_min + x_max) / 2, (y_min + y_max) / 2, (z_min + z_max) / 2

def recenter(self): # This will become useful when we draw the protein in PyGame.
    """Move the atoms so that the center is at (0, 0, 0)."""
    dx, dy, dz = self.get_center()
    for atom in self.protein:
        x, y, z = atom.getCoords()
        atom.setCoords((x - dx, y - dy, z - dz))

def relevant_residues(self):
    """This function receives a protein and ignores the residues that are
    not involved in a hydrogen bond or a disulfide bond.

    It returns a set containing only the indices of the relevant Residues."""
    relevant_res = set()
    for k, neighbors in self.edges.items():
        v = l2y_g.protein[k]
        for atom_2 in neighbors:
            k2 = atom_2.getIndex()
            bond_type = self.bond_type.get((k, k2))
            if bond_type in {'SS', 'HB'}:
                Res_v = v.getResnum()
                relevant_res.add(Res_v)
    return relevant_res

def simplified_protein(self):
    """This function constructs the graph of a protein using only the
    relevant residues and the backbone.

    :return: A dictionary of edges for this simplified protein,
    and a canonical spanning tree for it."""
    simp_prot_edges = col.defaultdict(set)
    spanning_tree = col.defaultdict(set)

    # A residue has the position of its CA atom
    residue_to_CA = dict()
    for atom in self.protein: #SIMP1: the backbone consists only on alpha-carbons
        if atom.getName() == 'CA':

```

```

        residue_to_CA[atom.getResnum()] = atom
        #SIMP3: we ignore the atoms that are not involved in a bond.

# Connect hydrogen and sulfur bonds, but only one per pair of amino acids. #SIMP4
# Maps pairs of amino acids to pairs of atoms, where the atoms form the bond.
bonds_between_residues = dict()
for atom_1, atom_2 in self.iter_neighbors():
    if atom_1.getResnum() == atom_2.getResnum(): #SIMP2: we don't draw edges between
        #an amino acid with itself
        continue
    bond_type = self.bond_type.get((atom_1.getIndex(),atom_2.getIndex()))
    if bond_type in {'SS', 'HB'}:
        bonds_between_residues[(atom_1.getResnum(),atom_2.getResnum())] = atom_1,atom_2

# Connect the backbone
# We assume the residues are numbered sequentially starting from 1
for i in range(1, len(residue_to_CA)):
    atom_1 = residue_to_CA[i]
    atom_2 = residue_to_CA[i + 1]
    k_1 = atom_1.getIndex()
    k_2 = atom_2.getIndex()
    bonds_between_residues[(i, i+1)] = atom_1, atom_2
    bonds_between_residues[(i+1, i)] = atom_2, atom_1
    self.bond_type[(k_1, k_2)] = 'backbone'
    self.bond_type[(k_2, k_1)] = 'backbone'
    spanning_tree[k_1].add(k_2)
    spanning_tree[k_2].add(k_1)

# Now we have a dict of which amino acids to connect and how.
for (resnum_1, resnum_2), (atom_1, atom_2) in bonds_between_residues.items():
    # Connect the bond itself.
    simp_prot_edges[atom_1.getIndex()].add(atom_2.getIndex())
    # Connect these to the CA atom.
    ca_1 = residue_to_CA[resnum_1]
    ca_2 = residue_to_CA[resnum_2]
    if ca_1 != atom_1:
        simp_prot_edges[ca_1.getIndex()].add(atom_1.getIndex())
        spanning_tree[ca_1.getIndex()].add(atom_1.getIndex())
    if ca_2 != atom_2:
        simp_prot_edges[atom_2.getIndex()].add(ca_2.getIndex())
        spanning_tree[atom_2.getIndex()].add(ca_2.getIndex())

return simp_prot_edges

```

5.6 File 6: computation_v2_c2.py

```

import numpy as np
import sympy as sp
t=sp.Symbol('t')

def column_row_mult(a,b): #we multiply a row by a column to get a matrix
    c=np.ndarray((a.shape[0], a.shape[0]), object)
    for i in range(0,a.shape[0]):
        for j in range(0,a.shape[0]):
            c[i,j]=a[i, 0]*b[0, j]

```



```

    return(c)

def delete_rc(matrix, r, c):
    """This function erases the rth row and cth column of the original matrix"""
    return np.delete(np.delete(matrix, r, axis=0), c, axis=1)

def kthDerivative(f,k):
    fprimes=f
    for i in range(1, k+1):
        fprimes= sp.sympify(sp.sympify(fprimes).diff(t))
    return sp.sympify(fprimes)

def fac(n):
    if type(n) is not int:
        raise ValueError("the input must be an integer")
    if n==0:
        return 1
    if n<0:
        raise ValueError("the input must be an integer greater or equal to 0")
    fac=1
    i=1
    while i<n:
        fac=fac*(i+1)
        i=i+1
    return(fac)

def TaylorSeries(f,a,n):
    """This function computes the n-th Taylor polynomial of the
    function f around the point a."""
    i=0
    g=0
    while i<=n:
        g= g + (kthDerivative(f,i).subs(t,a))/fac(i)*((t-a)**i)
    i=i+1
    return(g)

#This function receives two strings to attach: a,b
#It does the stitching m~{a,b}_{min(a,b)}, i.e. it renames the string as the min(a,b).

def v2Matrix(A,i,j):
    row_i = A[i, :] #row_n = B[n, :] returns the n-th row of B (we start to count from 0)
    row_i.shape = (1, A.shape[1]) #we need to make explicit the dim of row_n.
    row_j = A[j, :]
    row_j.shape = (1, A.shape[1])
    col_i = A[:, i] #col_n = B[:, n] returns the n-th column of B (we start to count from 0)
    col_i.shape = (A.shape[0], 1)
    col_j = A[:,j]
    col_j.shape = (A.shape[0], 1)

    alpha = A[i,i] #this is entry i,i of the original matrix A
    beta = A[i,j] #this is entry i,j of the original matrix A
    gamma = A[j,i] #this is entry j,i of the original matrix A
    delta = A[j,j] #this is entry j,j of the original matrix A

    #We now multiply column j by row i, we call the product matrix B
    B=column_row_mult(col_j,row_i)

```

```

#Here we simplify the following: divide every entry of the matrix B by 1-beta
# Often, beta is 0 so we can skip the division step.
if beta == 0:
    B1 = B
else:
    B1=np.empty_like(B)
    for a in range (0,len(A)): #len(A) gives us the number of rows in A
        for b in range (0, len(A)):
            B1[a,b] = B[a,b] / (1 - beta)
#We now sum the matrix B1 to A
A1 = B1 + A
#We now delete row i and column j from the matrix sum A+B1
# The tangle map only supports deleting row i and column i,
# so we permute the entries of the matrix
# to make sure we can delete them correctly.
for a in range(0, len(A)):
    #in practice we copy a row or a column but actually both copies happen,
    #we only notice one of them
    A1[min(i, j), a]=A1[j, a] #(*)
    A1[a, min(i, j)]=A1[a, i] #(*)
A_final=delete_rc(A1, max(i,j), max(i, j))
return A_final

class v2:
def __init__(self, matrix, names,omega=1): #if we don't enter a value for
# omega, it will be 1. Otherwise the program will take the value given
# for omega.
self.matrix = matrix
self.omega = omega
self.tangle_map = {}
for row, name in enumerate(names):
    self.tangle_map[name] = row

def beta(self, i, j):
return self.matrix[i, j]

def disjoint_union(self, other):
"""This function computes the disjoint union of v2 of two tangles.

This method updates the current v2 object in place.
"""

# We should have a square matrix.
assert self.matrix.shape[0] == self.matrix.shape[1],
    "{} is not square but {}".format(self.matrix, self.matrix.shape)
size = self.matrix.shape[0]

# The other tangle map should be offset by the first matrix.
for k, v in other.tangle_map.items():
    assert k not in self.tangle_map
    self.tangle_map[k] = v + size

self.omega = self.omega * other.omega

top_right = np.empty((size, other.matrix.shape[1]))
top_right.fill(sp.Float(0))

```

```

bottom_left = np.empty((other.matrix.shape[0], size))
bottom_left.fill(sp.Float(0))
self.matrix = np.block([
    [self.matrix, top_right],
    [bottom_left, other.matrix],
])

return self

def _stitching(self, i, j):
    """Stitch the components with indices i and j together.

    We attach the head of i to the tail of j.
    Note that we don't require i<j.
    """
    # Update the omega value.
    beta = self.matrix[i, j]
    #self.omega = sp.simplify((1 - beta) * self.omega)
    self.omega = TaylorSeries((1 - beta) * self.omega, 1, 2) #we consider here
    #[(1- beta)*omega]mod(1-t)^3
    print('omega: ', self.omega, 'shape: ', self.matrix.shape)

    # Compute the new matrix.
    self.matrix = v2Matrix(self.matrix, i, j)

    # Update the map of name to index.
    for name, row in self.tangle_map.items():
        deleted_row, kept_row = max(i, j), min(i, j)
        if row > deleted_row: # we need to move it down
            self.tangle_map[name] = row-1
        elif row == deleted_row: # we need to rename it
            self.tangle_map[name] = kept_row

    # Finished!
    return self

def stitching(self, name_i, name_j):
    """Stitch the components with names name_i and name_j together.

    We attach the head of name_i to the tail of name_j.
    """
    # Look up the matrix index from the names.
    i = self.tangle_map[name_i]
    j = self.tangle_map[name_j]
    # Stitch using the matrix indices.
    self._stitching(i, j)
    #self.taylor()
    return self

def lookup(self, name_i, name_j):
    i = self.tangle_map[name_i]
    j = self.tangle_map[name_j]
    return self.matrix[i, j]

def taylor(self):
    A1=np.empty_like(self.matrix)

```

```

for x in range(0, len(self.matrix)):
    for y in range(0, len(self.matrix)):
        A1[x,y]=TaylorSeries(self.matrix[x,y],1,2) #we chop Higher Order Terms,
#i.e. we consider mod(1-t)^3
self.matrix = A1

def __repr__(self):
    return 'omega={} matrix={} tangle_map={}'\
        .format(self.omega,self.matrix,self.tangle_map)

def single_crossing_matrix(strand1,strand2,is_positive):
    if is_positive:
        matrix = np.array([[1, 1-t], [0, t]])
    else:
        matrix = np.array([[1, 1-1/t],[0, 1/t]])
    return v2(matrix, [strand1, strand2])

def optimize_stitching_order(stitchings):
    """We are given a list of stitchings and want to return a permutation of them,
such that stitching them is as fast as possible.
    """
    original_stitchings = set(stitchings)

    # First optimization: try to stitch in the order of the crossing ids:
    # this way, a crossing is completely stitched soon after they first appear.
    stitchings = sorted(stitchings, key=lambda s: min(s[0], s[1]))

    # Next, we can stitch everything as soon as it appears in the matrix.
    # We iterate over the stitchings in order of appearance,
    # so first_appearance will always be empty or less than i.
    first_appearance = {}
    for i, (head, tail) in enumerate(stitchings):
        if head // 2 not in first_appearance:
            first_appearance[head // 2] = i
        if tail // 2 not in first_appearance:
            first_appearance[tail // 2] = i

    # We maintain the invariant that stitchings[0:i] has no delayed stitchings.
    for i in range(len(stitchings)):
        head, tail = stitching = stitchings[i]
        first_head = first_appearance[head // 2]
        first_tail = first_appearance[tail // 2]
        j = max(first_head, first_tail) + 1
        if j >= i:
            # This stitching is not delayed, continue!
            continue
        # Delete stitchings[i] from the list and insert it at point j.
        stitchings.pop(i)
        stitchings.insert(j, stitching)
        # Update the first appearances to reflect this moving.
        for k, v in first_appearance.items():
            assert v != i
            if j <= v < i:
                v += 1

    # Now everything for indices in range(j, i+1) don't need to move,

```

```

        # since j is after the first appearance for stitching[j].

    assert set(stitchings) == original_stitchings

    return stitchings

def generate_and_stitch(crossing_dict, stitchings):
    """Compute a v2 matrix for the crossings after stitching them as given.

    :param crossing_dict: maps strands to a tuple (strand1, strand2, is_positive),
    where strand1 goes over strand2 and is_positive is True if the sign of the
    crossing is positive.
    :param stitchings: list of pairs of strands, where the head of the first
    is stitched to the tail of the second.
    """

    # We start out with the first crossing, which we will need to add anyway.
    # If there are no crossings, the result is empty.
    if not crossing_dict:
        result = v2(np.array([[[]]]), [])
        return result

    strand1, strand2, is_positive = next(iter(crossing_dict.values()))
    result = single_crossing_matrix(strand1, strand2, is_positive)

    print("optimizing stitchings...")
    stitchings = optimize_stitching_order(stitchings)

    total = len(stitchings)
    for index, (head, tail) in enumerate(stitchings):
        print("progress: ", index / total * 100)
        if head not in result.tangle_map:
            strand1, strand2, is_positive = crossing_dict[head]
            crossing_matrix = single_crossing_matrix(strand1, strand2, is_positive)
            result = result.disjoint_union(crossing_matrix)
        if tail not in result.tangle_map:
            strand1, strand2, is_positive = crossing_dict[tail]
            crossing_matrix = single_crossing_matrix(strand1, strand2, is_positive)
            result = result.disjoint_union(crossing_matrix)
        result.stitching(head, tail)

    # If there are any crossings left that we haven't added, add them.
    for strand1, strand2, is_positive in crossing_dict.values():
        if strand1 in result.tangle_map:
            assert strand2 in result.tangle_map
            continue
        print("separate crossing: ", strand1, strand2)
        result = result.disjoint_union(single_crossing_matrix(strand1, strand2, is_positive))

    return result

```

5.7 File 7: compute_v2_protein.py

```

import collections as col
import numpy as np

```

```

import prody
import sympy
from graph import Graph
from graph_position import GraphWithPosition
from protein_structure import ProteinStructure
import intersection_of_lanes
import computation_v2_c2
import sys

t=sympy.Symbol('t')

# sys.argv contains the command line arguments, e.g:
# ['create_cycles_protein.py', '1l2y']
if len(sys.argv) > 1:
    protein_name = sys.argv[1]
else:
    protein_name = input("Please enter the entry of the protein> ")
protein = ProteinStructure(protein_name)

print("simplification...")
simp_str, spanning_tree_dict = protein.simplified_protein()

print("computing positions...")
graph = GraphWithPosition() #for 1l2y this graph has 305 vertices
for atom in protein.protein:
    graph.add_vertex(atom.getIndex(), atom.getCoords()) #with add_vertex we create
    #a dictionary called vertices self.vertices[vertex_id] = position
graph.add_edges(simp_str) #Given a dictionary of lists of neighbors (atoms indices),
#add each edge to the graph with add_edges.

spanning_tree_graph = Graph()
spanning_tree_graph.add_edges(spanning_tree_dict) #we do not care about coordinates
#for the vertices of the spanning tree

simplified_graph = GraphWithPosition() #for 1l2y, this graph has only 50 vertices, coords in 3d
for k in simp_str.keys():
    atom = protein.protein[k]
    simplified_graph.add_vertex(atom.getIndex(),atom.getCoords())
simplified_graph.add_edges(simp_str)

def all_cycles(graph,spanning_tree,root):
    """This method receives a graph and a root, a determined spanning tree and
    returns a dictionary. Key:edge, value:unique path (actually it's a cycle)
    along the given spanning tree.
    """
    Edges_out_of_tree = graph.generate_edges()-spanning_tree.generate_edges()
    dict_all_cycles = col.defaultdict(list)
    for edge in Edges_out_of_tree:
        if edge[0]>edge[1]:
            continue
        dict_all_cycles[edge] = spanning_tree.find_path(root,edge[0])
            +spanning_tree.find_path(edge[1],root)
    return dict_all_cycles

def projection_xy(vector):
    return np.array([vector[0],vector[1]])

```

```

print("computing cycles...")
cycles = all_cycles(simplified_graph,spanning_tree_graph,1)
print("computing crossings per cycles...")
crossings = intersection_of_lanes.crossings_per_lane_3d(simplified_graph, cycles)
crossing_directions = intersection_of_lanes.get_crossing_directions(simplified_graph, cycles,
                                                                    crossings)
to_be_stitched = intersection_of_lanes.stitch_lanes_together_for_crossings(simplified_graph,
                                                                            cycles, crossings)
v2 = computation_v2_c2.generate_and_stitch(crossing_directions, to_be_stitched)

omega = v2.omega
matrix = v2.matrix
tangle_map = v2.tangle_map
sec_der = sympy.diff(v2.omega.simplify(), t, 2)
k_for_v2= 0.5*sec_der.subs(t,1)
print("The matrix is:")
print(matrix)
print("The tangle map is:")
print(tangle_map)
print("The value of omega is:")
print(omega)
print("The value of one half of the second derivative of omega evaluated at 1 is:")
print(k_for_v2)

```

5.8 File 8: draw_protein_xy.py

```

#!/usr/bin/env python3

# With this code we show the simplified structure of the protein
# projected to the xy-plane.

from protein_structure import ProteinStructure

import collections
import math
import sys, pygame
import numpy as np

protein_name = input("Please enter the entry of the protein> ")
protein = ProteinStructure(protein_name)
protein.recenter()

pygame.init()

size = width, height = 640, 480
black = 0, 0, 0
white = 255, 255, 255

origin = [0, 0]
scale = min(width / 14, height / 12)

screen = pygame.display.set_mode(size)

def transform(x, y, z):

```

```

    return (x - origin[0])*scale + width/2, (origin[1]-y)*scale + height/2

def inverse_transform(px, py):
    return (px - width/2) / scale + origin[0], origin[1] - (py - height/2) / scale

class Vertex:
    def __init__(self, id, x, y, *neighbors):
        self.id = id
        self.x = x
        self.y = y
        self.neighbors = set(neighbors)

color_map = collections.defaultdict(lambda: white)
color_map['HB'] = (0, 0, 255)
color_map['SS'] = (255, 0, 0)
color_map['backbone'] = (255, 255, 0)

simplified_structure = protein.simplified_protein()[0]

while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: sys.exit()
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1: # left click
                origin = inverse_transform(event.pos[0], event.pos[1])
                scale *= 1.2
            elif event.button == 3: # right click
                origin = inverse_transform(event.pos[0], event.pos[1])
                scale *= 0.8

    screen.fill(black)
    for k, neighbors in simplified_structure.items():
        v = protein.protein[k]
        coords = v.getCoords()
        start = transform(coords[0], coords[1], coords[2])
        for k2 in neighbors:
            v2 = protein.protein[k2]
            coords2 = v2.getCoords()
            end = transform(coords2[0], coords2[1], coords2[2])
            bond_type = protein.bond_type.get((k, k2))
            pygame.draw.line(screen, color_map[bond_type], start, end)

pygame.display.flip()

```

5.9 File 9: draw_protein.py

```

#!/usr/bin/env python3

from protein_structure import ProteinStructure

import collections
import math
import sys, pygame
import numpy as np

```



```

protein_name = input("Please enter the entry of the protein> ")
protein = ProteinStructure(protein_name)
protein.recenter()

pygame.init()

size = width, height = 640, 480
black = 0, 0, 0
white = 255, 255, 255

origin = [0, 0]
scale = min(width / 14, height / 12)

screen = pygame.display.set_mode(size)

def transform(x, y, z):
    x = x * cos_angle - z * sin_angle
    return (x - origin[0])*scale + width/2, (origin[1]-y)*scale + height/2

def inverse_transform(px, py):
    return (px - width/2) / scale + origin[0], origin[1] - (py - height/2) / scale

class Vertex:
    def __init__(self, id, x, y, z, *neighbors):
        self.id = id
        self.x = x
        self.y = y
        self.z = z
        self.neighbors = set(neighbors)

rotationSpeed = 0.01
angle = 0

color_map = collections.defaultdict(lambda: white)
color_map['HB'] = (0, 0, 255)
color_map['SS'] = (255, 0, 0)
color_map['backbone'] = (255, 255, 0) #yellow

simplified_structure = protein.simplified_protein()[0]

while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: sys.exit()
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1: # left click
                origin = inverse_transform(event.pos[0], event.pos[1])
                scale *= 1.2
            elif event.button == 3: # right click
                origin = inverse_transform(event.pos[0], event.pos[1])
                scale *= 0.8

    angle += rotationSpeed
    cos_angle = math.cos(angle)
    sin_angle = math.sin(angle)

    screen.fill(black)

```

```

#In this loop we show all atoms, and the covalent bonds between them.
for k, neighbors in protein.edges.items():
    v = protein.protein[k]
    element1 = v.getElement()
    coords = v.getCoords()
    start = transform(coords[0], coords[1], coords[2])
    for v2 in neighbors:
        k2 = v2.getIndex()
        element2 = v2.getElement()
        coords2 = v2.getCoords()
        end = transform(coords2[0], coords2[1], coords2[2])
        bond_type = protein.bond_type.get((k, k2))
        #pygame.draw.line(screen, color_map[bond_type], start, end)

# In this loop we show the simplified structure of the protein.
# The backbone is shown in yellow, HBONDS in blue, SSBONDS in red and
# the relevant bonds are shown in white.
for k, neighbors in simplified_structure.items():
    v = protein.protein[k]
    coords = v.getCoords()
    start = transform(coords[0], coords[1], coords[2])
    for k2 in neighbors:
        v2 = protein.protein[k2]
        coords2 = v2.getCoords()
        end = transform(coords2[0], coords2[1], coords2[2])
        bond_type = protein.bond_type.get((k, k2))
        pygame.draw.line(screen, color_map[bond_type], start, end)

pygame.display.flip()

```

Appendix A

Algorithms to find Spanning Trees

The version of Prim's algorithm in Python presented here has been adapted so that for a graph with several connected components, it finds a spanning tree for each one of the components [Cor+09]:

```
A = dict()
B = set() # The set of nodes we have visited.
x=next(iter(graph.keys()))
tree = set() # a set of edges (v, w)
while True:
    # visit x
    B.add(x)
    for neighbor in graph[x]:
        if neighbor not in B:
            A[neighbor] = x, neighbor
    # go to the next x
    try:
        x, edge = next(iter(A.items())) #give an arbitrary element or raise StopIteration
    except StopIteration:
        # There is no next x, so we are finished with the tree.
        # But there might be another component to take the tree of.
        try:
            x = next(iter(set(graph.keys()) - B))
            continue
        except StopIteration:
            break

    del A[x] #A has all the vertices neighbors to the tree but not in the tree
    tree.add(edge)
print(tree)
```

Kruskal's algorithm automatically finds a spanning tree in each one of the connected components of a given graph.

```
# Kruskal's algorithm

# remember in which connected component we can find vertices
components = dict()
for vertex in graph:
    components[vertex] = vertex
tree = set()
n = len(graph)
# add all edges between different components
for x, y in edges_to_pairs(graph):
```

```
cx = components[x]
cy = components[y]
if components[x] != components[y]:
    tree.add((x, y))
    # merge the components
    for vertex in graph:
        if components[vertex] == cx:
            components[vertex] = cy
    print(tree)
# early exit when we are finished
if len(tree) >= n - 1:
    print('We have enough edges')
    break
```

Appendix B

Extra definitions

1. The *linking number* of two curves A and B indicates, intuitively, how many times curve A winds around curve B . It is defined as follows:

$$lk(A, B) := \frac{1}{2} \sum_s \epsilon(s)$$

where s is a crossing between curve A and B and $\epsilon(s)$ is the sign of the crossing.

2. The *Jones polynomial*. We use the term *polynomial* to denote a Laurent polynomial, i.e. a polynomial which can have both positive and negative powers. In order to define the Jones polynomial of a knot we first present a polynomial called *bracket polynomial*. Let K_0 denote the trivial knot. The bracket polynomial is a map

$$\langle \cdot \rangle : \text{Diagrams} \rightarrow \mathbb{Z} [A, A^{-1}]$$

given by the following relations:

$$(a) \left\langle \begin{array}{c} \diagup \diagdown \\ \diagdown \diagup \end{array} \right\rangle = A \left\langle \begin{array}{c} \text{---} \text{---} \\ \text{---} \text{---} \end{array} \right\rangle + A^{-1} \left\langle \begin{array}{c} \text{---} \text{---} \\ \text{---} \text{---} \end{array} \right\rangle$$

$$\left\langle \begin{array}{c} \diagdown \diagup \\ \diagup \diagdown \end{array} \right\rangle = A \left\langle \begin{array}{c} \text{---} \text{---} \\ \text{---} \text{---} \end{array} \right\rangle + A^{-1} \left\langle \begin{array}{c} \text{---} \text{---} \\ \text{---} \text{---} \end{array} \right\rangle$$

$$(b) \langle K_0 \sqcup L \rangle = (-A^2 - A^{-2}) \langle L \rangle$$

$$(c) \langle K_0 \rangle = 1.$$

Now, let L be a projection of a knot K with an orientation given. We can assign a sign to each crossing, as in figure 1.1. The *writhe* of L , $\omega(L)$, is defined as the sum of all the +1s and -1s corresponding to the crossings of K .

It turns out that the bracket polynomial is affected by type I Reidemeister move. Hence, we define a new polynomial, the *X polynomial* as follows:

$$X(L) := (-A^3)^{\omega(L)} \langle L \rangle.$$

The X polynomial is invariant under the three Reidemeister moves.

Definition 40. Let $X(L)$ denote the X polynomial of the knot diagram L . Then the *Jones polynomial* of L is obtained by replacing A by $t^{-\frac{1}{4}}$ in $X(L)$. \diamond

3. A 1-dimensional cell complex X is called a *graph*. It is formed by *vertices* (the 0-cells) and *edges* (the 1-cells). We denote the set of vertices by V and the set of edges by E , so that $X = (V, E)$. We will usually work with the embedding of a graph into \mathbb{R}^3 and we will also call this a *graph*.

Attribution

- The molecular graphics included in this thesis were created using UCSF Chimera [Pet+12].
- Figure 3.4 comes from the Proteopedia website [Hod+17].

Bibliography

- [Ada94] C.C. Adams. *The Knot Book: An Elementary Introduction to the Mathematical Theory of Knots*. New York: W.H. Freeman, 1994.
- [Alb+07] B. Alberts et al. *Molecular Biology of the Cell*. New York: W.H. Freeman, 2007.
- [Bar15] Dror Bar-Natan. “Balloons and hoops and their universal finite-type invariant, BF theory, and an ultimate Alexander invariant”. In: *Acta Math. Vietnam.* 40.2 (2015), pp. 271–329. ISSN: 0251-4184. DOI: 10.1007/s40306-014-0101-0. URL: <https://doi.org/10.1007/s40306-014-0101-0>.
- [Bar95] D. Bar-Natan. “On the Vassiliev knot invariants”. In: *Topology* 34.2 (1995), pp. 423–472. ISSN: 0040-9383. URL: [https://doi.org/10.1016/0040-9383\(95\)93237-2](https://doi.org/10.1016/0040-9383(95)93237-2).
- [BL93] J.S. Birman and X.-S. Lin. “Knot polynomials and Vassiliev’s invariants”. In: *Invent. Math.* 111.2 (1993), pp. 225–270. ISSN: 0020-9910. DOI: 10.1007/BF01231287. URL: <https://doi.org/10.1007/BF01231287>.
- [BMB11] A. Bakan, L.M. Meireles, and I. Bahar. “ProDy: Protein Dynamics Inferred from Theory and Experiments”. In: *Bioinformatics* 27.11 (2011), pp. 1575–1577.
- [Bru08] P. Y. Bruice. *Química Orgánica*. Prentice Hall, 2008.
- [CDM12] S. Chmutov, S. Duzhin, and J. Mostovoy. *Introduction to Vassiliev knot invariants*. Cambridge University Press, Cambridge, 2012, pp. xvi+504. ISBN: 978-1-107-02083-2. DOI: 10.1017/CB09781139107846. URL: <https://doi.org/10.1017/CB09781139107846>.
- [CF77] R. H. Crowell and R. H. Fox. *Introduction to Knot Theory*. Springer-Verlag, 1977.
- [Cor+09] T. H. Cormen et al. *Introduction to Algorithms*. MIT Press, 2009.
- [Die17] R. Diestel. *Graph Theory*. Springer-Verlag, 2017.
- [GPV00] Mikhail Goussarov, Michael Polyak, and Oleg Viro. “Finite-type invariants of classical and virtual knots”. In: *Topology* 39.5 (2000), pp. 1045–1068. ISSN: 0040-9383. DOI: 10.1016/S0040-9383(99)00054-3. URL: [https://doi.org/10.1016/S0040-9383\(99\)00054-3](https://doi.org/10.1016/S0040-9383(99)00054-3).
- [Hod+17] E. Hodis et al. *Proteopedia: the free, collaborative 3D-encyclopedia of proteins and other molecules*. Accessed: 2018-08-23. 2017. URL: http://proteopedia.org/wiki/index.php/Main%5C_Page.
- [Kon93] M. Kontsevich. “Vassiliev’s knot invariants”. In: *I. M. Gel fand Seminar*. Vol. 16. Adv. Soviet Math. Amer. Math. Soc., Providence, RI, 1993, pp. 137–150.
- [Liv93] C. Livingston. *Knot Theory*. Mathematical Association of America, 1993.
- [Lon05] E. Long. *Topological invariants of knots: three routes to the Alexander Polynomial*. Accessed: 2018-12-01. 2005. URL: <http://www.ucl.ac.uk/~ucbpeal/alexandermac.pdf>.
- [McR99] D. McRee. *Practical Protein Crystallography*. Burlington: Elsevier, 1999.
- [Mun17] B. van Munster. “Computing the Alexander Polynomial of Proteins”. Universiteit Leiden, 2017.
- [Pet+12] E.F. Pettersen et al. “UCSF Chimera—a visualization system for exploratory research and analysis”. In: *J. Comput Chem.* 25.13 (2012).

- [PV01] M. Polyak and O. Viro. “On the Casson knot invariant”. In: *J. Knot Theory Ramifications* 10.5 (2001). Knots in Hellas '98, Vol. 3 (Delphi), pp. 711–738. ISSN: 0218-2165. URL: <https://doi.org/10.1142/S0218216501001116>.
- [Sun12] D. Sunday. *Intersections of lines and planes*. Accessed: 2018-10-08. 2012. URL: http://geomalgorithms.com/a05-%5C_intersect-1.html.
- [Vas90] V. A. Vassiliev. “Cohomology of knot spaces”. In: *Theory of singularities and its applications*. Vol. 1. Adv. Soviet Math. Amer. Math. Soc., Providence, RI, 1990, pp. 23–69.
- [Vo18] H. Vo. “Alexander Invariants of Tangles via Expansions”. PhD thesis. University of Toronto, 2018.