Computing the Alexander polynomial of proteins

Munster, B. van

Bart van Munster

Computing the Alexander Polynomial of Proteins

Master thesis

Thesis advisor: dr. R.I. van der Veen

Date master exam: August 18, 2017

Mathematisch Instituut, Universiteit Leiden

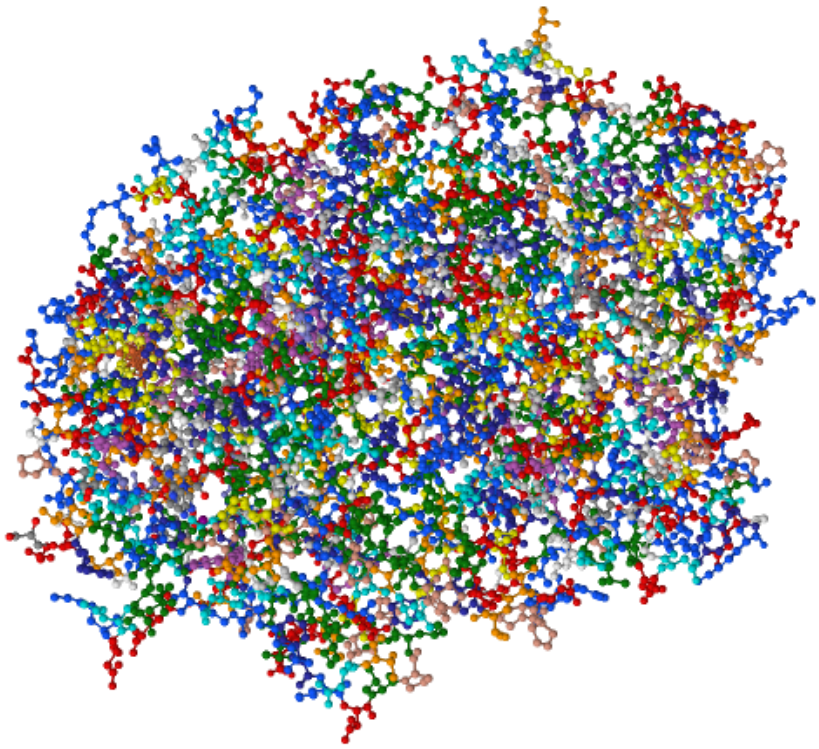


Figure 1: a protein, colored by amino acid

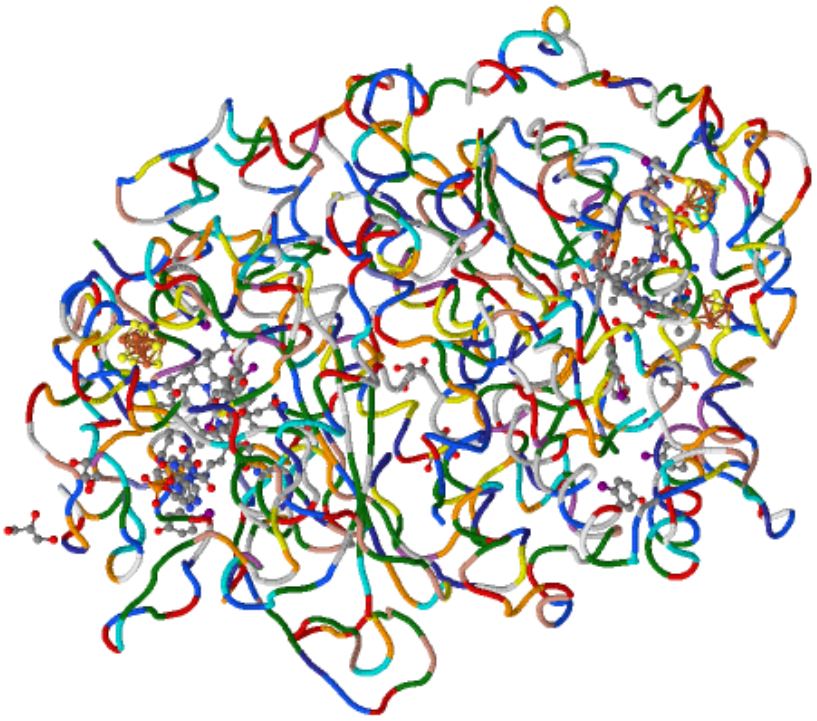


Figure 2: underlying structure of the protein from figure 1

Abstract

The Alexander polynomial is a link invariant and is used in this thesis to analyze the spatial structure of proteins. It can efficiently be computed by the Jacobian-adjugate method, derived from the adjugate of a Jacobian matrix of the link group. It does so by building a link diagram from smaller tangle diagrams using the operations disjoint union ($\sqcup$) and stitching (m_{ij} stitches edge i to edge j) and similarly computing the Jacobian-adjugates $\mathrm{JA}(T)$ of the tangle diagrams T. This Jacobian-adjugate is a pair (Δ, A) with $\Delta \in \mathbb{Z}[G^{ab}]$ and $A \in \mathrm{Mat}(\mathbb{Z}[G^{ab}])$, where G is the group of a link diagram. It satisfies the following rules:

$$\begin{aligned}(\Delta_A, A) \sqcup (\Delta_B, B) &= (\Delta_A \cdot \Delta_B, \Delta_B \cdot A \oplus \Delta_A \cdot B)\\ m_{ij}((\Delta, A)) &= (\Delta - A^i_j, (1 - \Delta^{-1} A^i_j)\widehat{A^i} + \Delta^{-1}\widehat{A^i_j} \cdot A^i)\,.\end{aligned}$$

Positive and negative crossings with over-strand a and outgoing and incoming under-strand edges b and c have as Jacobian-adjugates (Δ is written in the upper left corner)

$$\begin{array}{c|ccc} 1 & a & b & c \\ \hline a & 1 & 0 & 1-b \\ b & 0 & 1 & a \end{array} \quad \text{and} \quad \begin{array}{c|ccc} 1 & a & b & c \\ \hline a & 1 & 0 & a^{-1}(b-1) \\ b & 0 & 1 & a^{-1} \end{array}\,.$$

The Alexander polynomial equals the gcd of the remaining $1 \times n$ matrix when there is just one stitching left. In the single-variable case, at each stage of the computation we only need those columns corresponding to incoming edges, and the final 1×1 matrix gives the Alexander polynomial.

This Alexander polynomial can be used to compute topological invariants of proteins. Using Sage code we can reconstruct proteins from files from the Protein Data Bank and compute certain (single-variable) Alexander polynomials corresponding to it. Firstly, a link is obtained by closing chains along hydrogen bonds. Running the code on a sample of 1200 PDB files suggests that almost no protein contains a knotted structure. So it seems that the methods of studies such as [2] often infer a knotted structure that is absent in the actual protein. Secondly, a link is obtained from the protein by replacing chains and hydrogen bonds by mutually linked loops. The corresponding Alexander polynomial is very large and some improvements can still be made with regard to the computation time.

Acknowledgements

The risk of thanking many people in this section is, as I see it, suggesting that the writing of this thesis was a tedious process. But most of the time I really enjoyed working on it. That being said, I sincerely want to thank Roland, my supervisor, for all of his time and effort he invested. His expertise, ideas, enthusiasm and encouragements helped me a lot in writing this thesis. I would also like to thank all fellow Ichthus members with whom I spent many hours working (and having breaks) in the university library. Lastly, I would like to thank all people around me, including my girlfriend Jacolien and my family, who supported me during this process.

Contents

1 Introduction

Knots have been studied by mathematicians since the 19th century. In the beginning of the 20th century, knot theory became a part of the new field of topology, which provided the right mathematical tools to develop this theory. In the course of time, much theoretical progress has been made, and also various applications of knot theory have been discovered. These applications mainly fall within the area of physics, but there are applications in other natural sciences such as chemistry and molecular biology as well.

In this thesis, I will study a polynomial invariant of knots and links (chapter 3) that was discovered in 1923 by James Waddell Alexander II, which is therefore called the Alexander polynomial [1]. A new, efficient way to compute this polynomial is derived (chapter 6) by building a link diagram from tangle diagrams (chapter 2) and using the adjugate (chapter 4) of the Jacobian matrix (chapter 5) of a link group presentation. In chapter 7 it is investigated how this Jacobian-adjugate method relates to two similar tangle invariants, Γ-calculus [6] and rMVA [13].

The Alexander polynomial will be used in chapter 9 to compute a topological invariant of proteins (chapter 8). This has been done before ([2] and [20]), yet mostly conceding mathematical rigour: by artificially gluing loose ends together in various ways, the resulting 'invariants' are still very sensitive to small bendings. Even more, the structures that are studied are topologically all trivial. By not only considering atom bonds, but also hydrogen bonds, the resulting structure is a spatial graph which is not necessarily trivial. By using hydrogen bonds that close chains, I can find knots and links of which the Alexander polynomial can be computed. This will be applied to a sample from the RCSB Protein Data Bank. Secondly, I will explore another way of computing an Alexander polynomial of proteins, for which proteins are replaced by closely related links.

2 Knots & Links

All spaces are topological spaces and all maps are assumed to be continuous, unless stated otherwise. S^n is the n-sphere and I is the unit interval. Definitions 2.1, 2.2, 2.4, 2.5, 2.9, 2.15 and 2.13 can be found in various versions in the first chapters of [8], [17] and [19].

Definition 2.1. A *link* is a subset of S^3 (or $\mathbb{R}^3$) that is homeomorphic to the disjoint union of finitely many circles, each called a *link component*. An *oriented link* is a link with a fixed orientation on each circle. A link consisting of 1 circle is called a *knot*. Two links L_1, L_2 are equivalent if there is a homeomorphism $S^3 \to S^3$ mapping L_1 onto L_2. A link is called *tame* if it is equivalent to a polygonal link.

Knot theorists are often interested in whether two knots are ambient isotopic, which is a more distinctive equivalence relation.

Definition 2.2. An *isotopy* is a homotopy $H\colon X \times I \to Y$ such that for each $t_0 \in I$, $H(-, t_0)\colon X \to Y$ is a homeomorphism. An *ambient isotopy* between two subsets $A, B \subset M$ is an isotopy $F\colon M \times I \to M$ such that $F_0 = \mathrm{id}_M$ and $F_1(A) = B$. If such a map exists, A and B are called ambient isotopic.

Oriented links will be called ambient isotopic if there is such an F such that F_1 preserves orientation of the circles.

Remark 2.3. An ambient isotopy between two links induces an orientation-preserving equivalence F_1. In fact, every orientation-preserving homeomorphism of S^3 (or equivalently $\mathbb{R}^3$) is isotopic to the identity [11], so two links are ambient isotopic if and only if there is such an orientation-preserving equivalence.

From now on, all links are assumed to be tame and we will not distinguish between a link and its ambient isotopy class.

Definition 2.4. The *group* of a link is the fundamental group of its complement.

Note that this is well-defined without the use of a base point since the complement of a (tame) link is path-connected and any two ambient isotopic links have homeomorphic complements.

The rest of this section will be about ways to present a link and its group: link diagrams and Wirtinger presentations.

Definition 2.5. A *link diagram* of a link $L \subset \mathbb{R}^3$ is a subset $p(L) \subset \mathbb{R}^2 \subset \mathbb{R}^3$ with a projection $p\colon \mathbb{R}^3 \to \mathbb{R}^3$ onto a plane such that $p(\mathbb{R}^3) \cap L = \emptyset$ and for each $x \in p(L)$, $|p^{-1}(x) \cap L| \leq 2$. Whenever equality holds, this is a transversal intersection. Moreover, these intersections contain the information which strand goes over and which one under, i.e. which strand was closer to the projection surface in $\mathbb{R}^3$. Usually, this is done by removing a small interval around the intersection point of the under-strand. If the link is oriented, this orientation is also a part of the data of the the link diagram, usually indicated by arrows.

Figure 3: example of a link diagram, from [17]

A more general type of diagrams is defined as follows.

Definition 2.6. A *tangle diagram* is a finite directed graph in which every vertex has either degree 1 or 4 and every 4-valent vertex has 2 incoming and 2 outgoing edges. Each 4-valent vertex has a sign (± 1) and a distinguished incoming and outgoing edge. The class of tangle diagrams is denoted as TD.

An oriented link diagram can be seen as a tangle diagram with no 1-valent vertices (leaves). All crossings are 4-valent vertices, the over-strand of a crossing corresponds with the indicated incoming and outgoing edge and the under-strand with the other edges. The sign of each crossing is as in figure 4.

Figure 4: positive and negative crossing

Now that we have seen how tangle diagrams are generalizations of link diagrams, we can safely use the following definitions.

Definition 2.7. A 4-valent vertex in a tangle diagram is called a *crossing*. The indicated incoming and outgoing edge form the *over-strand*, the other two edges form the *under-strand*.

Example 2.8. A *crossing diagram* is a connected tangle diagram with one crossing. It can be represented as one of the crossings in figure 4.

Definition 2.9. A *bridge* in a tangle diagram is a maximal (non-repeating) series of edges in which each pair of subsequent edges forms the over-strand of some crossing.

Definition 2.10. *Disjoint union* is a binary operation $\sqcup\colon \mathrm{TD}\times\mathrm{TD} \to \mathrm{TD}$ which yields the disjoint union of graphs. *Stitching* is a unary operation $m_{ij}\colon \mathrm{TD} \to \mathrm{TD}$ for each pair of edges i, j such that i ends at a leaf and j begins at a leaf. It deletes edges i and j and both leaves involved and creates a new edge that starts where i started and ends where j ended. We may also refer to this operation as 'i is stitched to j'.

Example 2.11. The Solomon's knot, which is actually not a knot but a link, looks like figure 5. This link diagram can be obtained as the disjoint union of four positive crossings, with stitchings as indicated by the dotted lines.

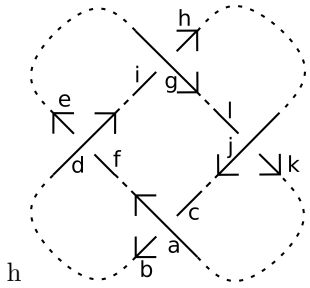


Figure 5: Solomon's knot

Definition 2.12. The *group* of a tangle diagram with edges E is a group G together with a map $t\colon E \to G$. It is defined recursively as follows: the group of a crossing diagram is the group $\langle a, b, c \,|\, aba^{-1}c^{-1}\rangle$. The edges of the over-strand are mapped to a and the edges of the under-strand to b and c. If the crossing is positive, b belongs to the outgoing edge and c to the incoming one, and vice versa if the crossing is negative. The group of a diagram consisting of one edge e is isomorphic to $\mathbb{Z}$ with $t(e) = 1$. The group of a disjoint union of tangle diagrams is the free product of their groups together with the union of their maps. After applying stitching to a tangle diagram with group G and map $t\colon E \to G$, its group is obtained by adding the relation $t(i) = t(j)$ to G. The new map sends the new edge to this $t(i)$.

Note that this is well-defined and in this way, each bridge has exactly one corresponding generator in G. This definition actually describes the fundamental group of the complements of topological objects called *tangles*, but elaborating on this is not needed and will therefore be omitted.

Definition 2.13. The *Wirtinger presentation* of a tangle diagram is recursively defined using the crossing group presentations from definition 2.12. The disjoint union of tangle diagrams gives a straightforward union of presentations and stitching i to j means replacing the generators $t(i)$ and $t(j)$ by one single generator. If instead of this we add a relator $t_i t_j^{-1}$ and never stitch edges both beginning and ending at a leaf, we obtain the *extended Wirtinger presentation.*

Note that the Wirtinger presentation and extended Wirtinger presentation are also well-defined.

Example 2.14. The group of the Solomon's knot from example 2.11 has Wirtinger presentation

$$\langle s, t, u, v \,|\, sts^{-1}u^{-1}, tvt^{-1}s^{-1}, vuv^{-1}t^{-1}, usu^{-1}v^{-1}\rangle\,.$$

Definition 2.15. The *linking number* $\mathrm{lk}(K_1, K_2)$ of two disjoint oriented knots $K_1, K_2 \subset \mathbb{R}^3$ given a certain link diagram of $K_1 \cup K_2$ equals $\frac{1}{2}(p - n)$, where p is the number of positive crossings and n the number of negative crossings involving both K_1 and K_2.

Proposition 2.16. *Up to a sign ± 1, the linking number is independent of the chosen link diagram.*

Proof. See section 5D of [19]. □

Theorem 2.17. *The group of a link L is isomorphic to the group of any link diagram of L. Leaving out one of the relators in an (extended) Wirtinger presentation always results in a presentation of the same group.*

Proof. We consider a Wirtinger presentation of L. First, the topological interpretation of its generators should be explained. A basepoint is chosen on the same side of the projection plane as the link, but further from this plane than

any $x \in L$. Now the generators are loops around exactly one bridge which have linking number 1 with the link. The case of an ordinary Wirtinger presentation is further covered by theorem 3.4 and corollary 3.6 of [8], and by using van Kampen's theorem in [19] Theorem 3.D.2. Leaving out one relator of the form $aba^{-1}c^{-1}$ in an extended Wirtinger presentation is therefore also possible. Leaving out a relator of the form ab^{-1} can be visualized as cutting the link at the point where the corresponding strands would have been stitched and infinitely extending the loose strands in the projection direction. Now such a tangle also has a Wirtinger presentation, and the same proofs apply to show that this actually is a presentation of the fundamental group of its complement. Here we can see that $a = b$: a loop around one infinite end of the tangle can be pulled around the entire link to turn it into a loop around the other infinite end. □

3 The Alexander Module

Recall the following important theorem in algebraic topology, which can be found, for example, as theorem 1.38 and proposition 1.39 of [14].

Theorem 3.1. *For every path-connected, locally path-connected, semi-locally simply connected space B with base point $b_0 \in B$, the assignment*

$$(p\colon (E, e_0) \to (B, b_0)) \mapsto p_*(\pi_1(E, e_0))$$

induces an anti-equivalence between the category of coverings of B and the category of subgroups of $\pi_1((B, b_0))$. If $p_*(\pi_1(E, e_0))$ is a normal subgroup of $\pi_1(B, b_0)$, for the group $\mathrm{Aut}_p(E, e_0)$ of covering automorphisms we have

$$\mathrm{Aut}_p(E, e_0) \cong \pi_1(B, b_0)/p_*(\pi_1(E, e_0))\,.$$

The automorphism corresponding to an element $\alpha \in \pi_1(B, b_0)$ sends each point to the endpoint of the lift of a loop representing this α.

From now on, the choice of base points is implicit. The subgroups of $\pi_1(E)$ that will be used to find a covering are normal, so this does not lead to ambiguity.

Definition 3.2. The *universal abelian covering* X_∞ of a space X with fundamental group G is the covering space with fundamental group $[G, G]$. So it has covering automorphism group $G/[G, G] = G^{ab}$. We will denote the covering map $X_\infty \to X$ by p.

Remark 3.3. If $q\colon G \to G/H$ is some quotient map of groups, I will not make notational distinction between an element $g \in G$ and $p(g)$ whenever it is clear from the context what is meant.

Observation 3.4. We have $H_1(X_\infty) = [G, G]^{ab}$ as it is the abelianization of the fundamental group $[G, G]$ of X_∞. We have an action of G^{ab} on $H_1(X_\infty)$

as the former is a group of automorphisms of X_∞. This action is in fact the conjugation action of G^{ab} on $[G,G]^{ab}$: for $g \in G$ and a loop h in X_∞ starting at x_0, $\overline{g}h$ is homologous to $l(g) \cdot \overline{g}h \cdot l(g)^{-1}$, where $l(g)$ is the lift of g which starts at x_0. This projects down to $g \cdot p(h) \cdot g^{-1}$. Note also that since $H_1(X_\infty)$ is abelian, we can extend the action of G^{ab} to an action of the group ring $\mathbb{Z}[G^{ab}]$, which turns $H_1(X_\infty)$ into a $\mathbb{Z}[G^{ab}]$-module.

Example 3.5. Let L be the Hopf link, as the one on the left in figure 6. Considering it as a link in S^3 with with one point at infinity, it looks like the picture on the right in $\mathbb{R}^3$. Here we can see that its complement retracts onto a torus T. So it has fundamental group $G = \mathbb{Z}^2 = G^{ab}$. This T has a universal covering $p\colon \mathbb{R}^2 \to T$ which is equivalent to the quotient map obtained from the relation $(a,b) \sim (c,d) \Leftrightarrow a-c, b-d \in \mathbb{Z}$ on $\mathbb{R}^2$. This is also its universal abelian covering T_∞ because $[G,G]$ is trivial. The action of G^{ab} on $H_1(X_\infty) = [G,G]^{ab}$ is obviously also trivial, but can nevertheless be pictured as if any covering automorphism $(a,b) \mapsto (a+x, b+y)$ with $x, y \in \mathbb{Z}$ shifts loops in $\mathbb{R}^2 = T_\infty$.

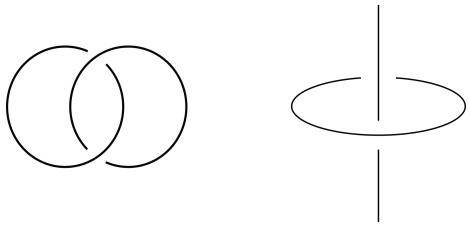


Figure 6: Hopf link

Example 3.6. The Solomon's knot from example 2.11 has group

$$G = \langle s,t,u,v \,|\, sts^{-1}u^{-1}, tvt^{-1}s^{-1}, vuv^{-1}t^{-1}, usu^{-1}v^{-1} \rangle \,.$$

Because of proposition 2.17, we can leave out the last relator. We use the first two to rewrite u and v in s and t, which gives $u = sts^{-1}$ and $v = t^{-1}st$. Entering these into the third relator gives

$$vuv^{-1}t^{-1} = t^{-1}ststs^{-1}t^{-1}s^{-1}tt^{-1} = t^{-1}ststs^{-1}t^{-1}s^{-1} \,.$$

Conjugating this relator with s^{-1} gives the group presentation

$$G = \langle s,t \,|\, [s^{-1},t^{-1}][s,t] \rangle \,.$$

So $G^{ab} = \langle s,t \,|\, [s,t] \rangle$ and by lemma 3.20, $[G,G]$ is generated by $[s,t] = sts^{-1}t^{-1}$ as a $\mathbb{Z}[G^{ab}]$-module.

Definition 3.7. Let L be a link with group G. The *Alexander module* M of L is the $\mathbb{Z}[G^{ab}]$-module $[G,G]^{ab}$.

We will also need a variation of the universal abelian covering, although this yields the same covering for all topological spaces to be considered in this thesis. For this we also need the following definition.

Definition 3.8. Let G be a group with normal subgroup H such that G/H is abelian and let $q : G \to G/H$ be the canonical projection. Then the *radical* of H is the normal subgroup of G defined as

$$\sqrt{H} = q^{-1}(\operatorname{Tors} G/H) = \{g \in G : g^n \in H \text{ for some } n\}.$$

Definition 3.9. Let X be a path-connected, locally path-connected, semi-locally simply connected topological space with fundamental group G. The *maximal free abelian covering* of X is the covering space $\overline{X}$ of X with fundamental group $\sqrt{[G,G]}$.

Remark 3.10. By lemma 3.11, any link complement X satisfies $\overline{X} = X_\infty$.

Lemma 3.11. *Let n be the number of components of L. G^{ab} is generated by loops linked with exactly one link component, with linking number ± 1. Hence $G^{ab} \cong \mathbb{Z}^n$.*

Proof. See [8] proposition 3.5. Note that $G^{ab} \cong H_1(X)$. □

Definition 3.12. Let R be a commutative ring. A *presentation* for an R-module M is a pair (α, ϕ) fitting in a short exact sequence

$$F \xrightarrow{\alpha} E \xrightarrow{\phi} M \to 0$$

of R-modules where F and E are free. The map α will also be called a presentation. M is said to be finitely presented if F and E are finitely generated. A corresponding *presentation matrix* for M is a matrix representing α with respect to some bases of F and E.

Definition 3.13. Let M be a finitely presented R-module. The *r-th elementary ideal* E_r of M is the ideal of R generated by the $(m-r) \times (m-r)$ minors of an $m \times n$ presentation matrix. A generator of the smallest principal ideal containing E_r is denoted as $\Delta_r(M)$.

Lemma 3.14. *The r-th elementary ideals are well-defined.*

Proof. This follows from theorem 6.1 of [17]. □

Definition 3.15. An Alexander matrix of a link L is a presentation matrix for its Alexander module.

Definition 3.16. The *r-th Alexander ideal* E_r is the r-th elementary ideal of the Alexander module M, i.e. the ideal of $\mathbb{Z}[G^{ab}]$ generated by the $(m-r) \times (m-r)$ minors of an $m \times n$ Alexander matrix. The *r-th Alexander polynomial* equals $\Delta_r(M)$. Δ_0 will just be called the *Alexander polynomial*.

Example 3.17. Again returning to the Hopf link, we can take a presentation $0 \to 0 \to M \to 0$ with $M = 0$ its Alexander module. Hence it has a 0×0 Alexander matrix and its Alexander ideals are all 0 except for E_0. This one equals (1) and hence the Alexander polynomial is 1.

Example 3.18. The Solomon's knot has group $G = \langle s, t \,|\, [s^{-1}, t^{-1}][s, t]\rangle$ and $[G, G]^{ab}$ is generated by $[s, t]$ as a $\mathbb{Z}[G^{ab}]$-module. Let

$$f \colon \mathbb{Z}[G^{ab}] \to [G, G]^{ab}, 1 \mapsto [s, t]\,.$$

Let $\cdot$ denote the action of $\mathbb{Z}[G^{ab}]$ on $[G, G]^{ab}$. Since $ts \cdot [s^{-1}, t^{-1}] = [s, t]$ and $[s^{-1}, t^{-1}] = [s, t]^{-1}$, we have

$$(-1 - ts) \cdot [s, t] = [s, t]^{-1}(ts \cdot [s, t]^{-1}) = [s, t]^{-1}[s, t] = e\,.$$

Now it follows from proposition 3.22 that $\ker f = \langle st + 1\rangle$. So a presentation of the Alexander module is given by

$$\mathbb{Z}[G^{ab}] \to \mathbb{Z}[G^{ab}], 1 \mapsto st + 1\,.$$

This has Alexander matrix $(st + 1)$, so its Alexander polynomial equals $st + 1$.

Lemma 3.19. *Let G be a group. There is a well-defined conjugation action of G on $[G, G]$, which induces a conjugation action of G^{ab} on $[G, G]^{ab}$.*

Proof. The first statement is immediate since $[G, G]$ is a normal subgroup. So conjugation gives a well-defined homomorphism $G \to \mathrm{Aut}([G, G])$. Composing this with $\mathrm{Aut}([G, G]) \to \mathrm{Aut}([G, G]^{ab})$ gives a map with $[G, G]$ in its kernel. □

Now let $G = \langle a, b\rangle \cong \mathbb{Z} * \mathbb{Z}$.

Lemma 3.20. *The $\mathbb{Z}[G^{ab}]$-module $[G, G]^{ab}$ is generated by $aba^{-1}b^{-1}$.*

Proof. First note that any $a^n b^m a^{-n} b^{-m}$ with $n, m \in \{-1, 1\}$ can be obtained by conjugating $aba^{-1}b^{-1}$ with some element from G. For $n > 1$ we have that $a^n ba^{-n}b^{-1} = a \cdot a^{n-1}ba^{-n+1}b^{-1} \cdot a^{-1} \cdot aba^{-1}b^{-1}$. So by induction it follows that $a^n ba^{-n}b^{-1} \in \langle aba^{-1}b^{-1}\rangle$ for each $n \geq 0$ and similarly for each $n \leq 0$. Since $a^{n-1} \cdot ab^m a^{-1}b^{-m} \cdot a^{-n+1} \cdot a^{n-1}b^m a^{-n+1}b^{-m} = a^n b^m a^{-n} b^{-m}$ it also follows by induction that each $a^n b^m a^{-n} b^{-m}$ is in the submodule of $[G, G]$ generated by $aba^{-1}b^{-1}$. Together with the next lemma, this proves the claim. □

Lemma 3.21. *$[G, G]$ is generated (as a group) by $\{a^n b^m a^{-n} b^{-m} : n, m \in \mathbb{Z}\}$.*

Proof. $x = a^{n_1}b^{m_1}...a^{n_k}b^{m_k}$ such that $\sum n_i = \sum m_i = 0$ and $n_i, m_i \neq 0$ for all $i < k$ and $n_k \neq 0$. Then $(a^{n_1}b^{m_1}a^{-n_1}b^{-m_1})b^{m_1}a^{n_1+n_2}b^{m_2}a^{n_3}b^{m_3}...a^{n_k}b^{m_k} = x$. This can be easily adapted to the case where $n_1 = 0$ by switching the roles of a and b. (Note that $b^m a^n b^{-m} a^{-n} = (a^n b^m a^{-n} b^{-m})^{-1}$.) So by induction on the length of the word x, we have proved that $[G, G] = \{a^{n_1}b^{m_1}...a^{n_k}b^{m_k} \,|\, \sum n_i = \sum m_i = 0\}$ and it is generated by $\{a^n b^m a^{-n} b^{-m} : n, m \in \mathbb{Z}\}$. □

Proposition 3.22. *$[G, G]^{ab}$ has a trivial presentation $0 \to \mathbb{Z}[G^{ab}]$.*

Proof. From the foregoing lemma we obtain a surjective $\mathbb{Z}[G^{ab}]$-module homomorphism $f\colon \mathbb{Z}[G^{ab}] \to [G,G]^{ab}, x \mapsto x \cdot aba^{-1}b^{-1}$, where $\cdot$ denotes the action of $\mathbb{Z}[G^{ab}]$ on $[G,G]^{ab}$. We have to prove that its kernel is trivial. This follows from the following geometric interpretation: consider the topological space X that looks like ∞, i.e. take the disjoint union of two intervals $[0,1]$ and identify all endpoints. This has fundamental group (isomorphic to) G. Its universal abelian covering X_∞ is the Cayley graph of $\mathbb{Z} \times \mathbb{Z}$, i.e. $\mathbb{Z} \times \mathbb{R} \cup \mathbb{R} \times \mathbb{Z}$. The covering map induces an isomorphism of groups $\pi_1(X_\infty, (0,0)) \cong [G,G]$. The action of G^{ab} on $[G,G]^{ab}$ is now the same as the action of the covering translations of X_∞ on $H_1(X_\infty)$. Let $x = m_1 a^{i_1} b^{j_1} + ... + m_n a^{i_n} b^{j_n} \in \mathbb{Z}[G^{ab}]$ with all $m_k \in \mathbb{Z}$ and $(i_k, j_k) \neq (i_l, j_l)$ for $k \neq l$. Then $f(x)$ is a loop that goes exactly m_k times (counterclockwise) around the 1×1 square with (i_k, j_k) as its lower left vertex. (A negative m_k means clockwise loops.) Hence $f(x)$ is a non-trivial element of $H_1(X_\infty) \cong [G,G]^{ab}$ for $x \neq 0$, since it goes around some 1×1 square S. (As a 1-cell, it would also be a non-trivial element of $H_1(\mathbb{R}^2 \setminus \{p\})$, where p is in the middle of S.) □

Let us try to generalize this to an arbitrary finitely generated group G with generating set $S = \{s_1, ..., s_n\}$ and set of relators R.

Lemma 3.23. *The $\mathbb{Z}[G^{ab}]$-module $[G,G]^{ab}$ is generated by $\{s_i s_j s_i^{-1} s_j^{-1} | i < j\}$.*

Proof. Let $x'yx'^{-1}y^{-1}$ be an element of $[G,G]$, with $x', y \in G$. We can write x' as a finite word in $S \cup S^{-1}$; let s be its first letter. We write $x' = sx$. Now we get $x'yx'^{-1}y^{-1} = sxyx^{-1}s^{-1}y^{-1} = s \cdot xyx^{-1}y^{-1} \cdot s^{-1} \cdot sys^{-1}y^{-1}$. So by induction, we only need elements of the form $xyx^{-1}y^{-1}$ with $x \in S \cup S^{-1}$ to generate $[G,G]^{ab}$. If we write an element as $y(sx)y^{-1}(x^{-1}s^{-1})$, we see that it is the inverse of $sxyx^{-1}s^{-1}y^{-1}$. Hence (cf. lemma 3.21) we only need generators of the form $xyx^{-1}y^{-1}$ with $x, y \in S \cup S^{-1}$. Note that as in lemma 3.20, any element $s_i^n s_j^m s_i^{-n} s_j^{-m}$ or $s_j^n s_i^m s_j^{-n} s_i^{-m}$ with $n, m \in \{-1, 1\}$ can be obtained by conjugating $s_i s_j s_i^{-1} s_j^{-1}$ with some element from $\langle S \rangle$. Hence the lemma follows. □

Corollary 3.24. *For every finitely generated group $G = \langle s_1, ..., s_n | R \rangle$, the $\mathbb{Z}[G^{ab}]$-module $[G,G]^{ab}$ is finitely presented.*

Proof. Because it is finitely generated, we obtain a surjective homomorphism $f\colon \mathbb{Z}[G^{ab}]^{\binom{n}{2}} \to [G,G]^{ab}$, which sends the i'th basis vector to the i'th generator. Since $\mathbb{Z}[G^{ab}] \cong \mathbb{Z}[s_1, ..., s_n]/\langle R \rangle$ is Noetherian, $\ker f$ is finitely generated. This gives us a presentation $\alpha\colon \mathbb{Z}[G^{ab}]^m \to \mathbb{Z}[G^{ab}]^{\binom{n}{2}}$, which sends the i'th basis vector to the i'th generator of $\ker f$. □

Corollary 3.25. *Every link has a finitely presented Alexander module and hence a well-defined Alexander polynomial.*

The rest of this section goes deeper into the explicit algebraic description of $[G,G]^{ab}$ as a $\mathbb{Z}[G^{ab}]$-module. The reader may skip this part in a first read, since it will not be used in subsequent chapters.

Definition 3.26. Consider $f\colon \mathbb{Z}[G^{ab}]^{\binom{n}{2}} \to [G,G]^{ab}$ as before, where $(\mathbb{Z}[G^{ab}])^{\binom{n}{2}}$ is indexed by pairs (i,j) such that $1 \leq i < j \leq n$ and basis vector $e_{(i,j)}$ is sent to $s_i s_j s_i^{-1} s_j^{-1}$. The *cube relators* of $\mathbb{Z}[G^{ab}]^{\binom{n}{2}}$ are elements of the form $(1-s_k)e_{(i,j)} + (s_j - 1)e_{(i,k)} + (1-s_i)e_{(j,k)}$ with $i < j < k$.

The reason why they are called 'cube relators' is the following: if G is free of rank n, then it is the fundamental group of a complex consisting of one 0-cell and n 1-cells. Its universal abelian covering X_∞ looks like the Cayley graph of $\mathbb{Z}^n$, i.e. $\mathbb{R} \times \mathbb{Z}^{n-1} \cup \mathbb{Z} \times \mathbb{R} \times \mathbb{Z}^{n-2} \cup ... \cup \mathbb{Z}^{n-1} \times \mathbb{R}$. Now f maps a cube relator r to something that looks like an element of $C_1(X_\infty)$ that loops around all 6 faces of a cube in X_∞. This loop passes each edge two times and it does so in opposite directions, hence $f(r)$ is trivial:

Lemma 3.27. *The cube relators are in the kernel of* $f\colon \mathbb{Z}[G^{ab}]^{\binom{n}{2}} \to [G,G]^{ab}$.

Proof. Let $r = (1-s_k)e_{(i,j)} + (s_j-1)e_{(i,k)} + (1-s_i)e_{(j,k)}$ be a cube relator and write $(a,b,c) = (s_i, s_j, s_k)$. Then we obtain:

$$\begin{aligned} f(r) &= aba^{-1}b^{-1} \cdot c \cdot (aba^{-1}b^{-1})^{-1} \cdot c^{-1} \cdot b \cdot aca^{-1}c^{-1} \cdot b^{-1} \\ &\quad \cdot (aca^{-1}c^{-1})^{-1} \cdot bcb^{-1}c^{-1} \cdot a \cdot (bcb^{-1}c^{-1})^{-1} \cdot a^{-1} \\ &= aba^{-1}b^{-1} \cdot b \cdot aca^{-1}c^{-1} \cdot b^{-1} \cdot bcb^{-1}c^{-1} \\ &\quad \cdot c \cdot (aba^{-1}b^{-1})^{-1} \cdot c^{-1} \cdot (aca^{-1}c^{-1})^{-1} \cdot a \cdot (bcb^{-1}c^{-1})^{-1} \cdot a^{-1} \\ &= aba^{-1}b^{-1} \cdot b \cdot aca^{-1}c^{-1} \cdot b^{-1} \cdot bcb^{-1}c^{-1} \\ &\quad \cdot c \cdot bab^{-1}a^{-1} \cdot c^{-1} \cdot cac^{-1}a^{-1} \cdot a \cdot cbc^{-1}b^{-1} \cdot a^{-1} \\ &= e \end{aligned}$$

□

Lemma 3.28. *If* $G = \langle s_1, ..., s_n \rangle$*, a free group on* $n \geq 3$ *generators, the set of cube relators* C *spans the kernel of* $f\colon \mathbb{Z}[G^{ab}]^{\binom{n}{2}} \to [G,G]^{ab}$.

Proof. Let $x \in \ker f$. We can write $f(x) = \sum_{i<j} a_{i,j}(s_i s_j s_i^{-1} s_j^{-1})^{m_{i,j}} a_{i,j}^{-1}$ with all $a_{i,j} \in G^{ab}$ by lemma 3.23. Geometrically, we can interpret this as an element of $H_1(X_\infty)$ (as before, X_∞ is the Cayley graph of $\mathbb{Z}^n$) represented by a collection C of loops $l_{i,j} = a_{i,j}(s_i s_j s_i^{-1} s_j^{-1})^m_{i,j} a_{i,j}^{-1}$. Any of these loops goes around exactly one 1×1 square. The starting point is marked by $a_{i,j}$, (i,j) indicates its direction and $m_{i,j}$ is the number of times the loop goes around this square. We can count how many times this representation D goes along any 1-cell/edge in X_∞; I claim that this is 0 for all edges v if we count opposite directions with opposite signs. To see this, consider $\mathbb{R}^n \setminus S$, where $S \cong S^{n-2}$ is an $(n-2)$-sphere in $\mathbb{R}^n \setminus X_\infty$ around v. Since $f(x)$ is trivial in $H_1(X_\infty)$, it is also trivial in $H_1(\mathbb{R}^n \setminus S)$; from this the claim follows. The idea is now to sequentially subtract suitable cubes from the corners of D to reduce it to 0. Thus let p be a vertex of D with maximal distance to 0. There is at least one square $S_1 \in D$ adjacent to p; by the foregoing consideration about how the representation goes along edges,

there is a square $S_2 \in D$ adjacent to S_1 with $p \in S_2$ such that S_1 and S_2 have opposite directions in the edge they share. Because of the maximality of p, S_1 and S_2 are not in one plane. Hence these two squares span a cube. Let r be the cube relator related to this cube. Replace r by $-r$ if necessary to match the orientations with S_1 and S_2. We can see that $f(x-r)$ has a representation that has less squares (counted with absolute multiplicity) adjacent to p. By maximality of p, all vertices $q \neq p$ of the cube related to r are closer to 0 than p. Hence this procedure can be repeated until p is no longer in D and ultimately until $D = \emptyset$. In conclusion, $x \in \langle C \rangle$. □

Corollary 3.29. *Let $G = \langle s_1, ..., s_n \rangle$ be a free group on $n \geq 3$ generators. Let $R = \mathbb{Z}[G^{ab}]$ and index the basis vectors of $R^{\binom{n}{m}}$ by increasing m-tuples. A presentation of $[G,G]^{ab}$ is given by $R^{\binom{n}{3}} \to R^{\binom{n}{2}}, e_{i,j,k} \mapsto (1-s_k)e_{(i,j)} + (s_j - 1)e_{(i,k)} + (1-s_i)e_{(j,k)}$.*

4 Adjugate matrices and determinants

In the following section A is an $n \times n$ matrix over a field and $i, j \in \{1, ..., n\}$. Row i of A is denoted by A^i (I will not use matrix powers), column j by A_j and A^i_j means the entry of A in row i and column j.

Definition 4.1. The (i,j) *minor* M^i_j of A is the determinant of the submatrix of A formed by deleting the i'th row and the j'th column. The (i,j) *cofactor* C^i_j equals $(-1)^{i+j} \cdot M^i_j$. These cofactors form a *cofactor matrix* C. The *adjugate* $\widetilde{A}$ of A is the transpose of its cofactor matrix.

Lemma 4.2. *The adjugate $\widetilde{A}$ of A satisfies $A\widetilde{A} = \widetilde{A}A = \det(A) I_n$.*

Proof. It is easily seen that $(A\widetilde{A})^i_i$ equals $\det(A)$, computed by expansion along the i'th row. For $i \neq j$, define A' as the matrix obtained from A by replacing row i with row j. Then entry $(A\widetilde{A})^i_j$ can be seen as the determinant of A', computed by expansion along row i. Since A' has two equal rows, this results in 0. Hence $A\widetilde{A} = \det(A) I_n$. The proof for $\widetilde{A}A = \det(A) I_n$ is similar. □

Observation 4.3. If $\operatorname{rk} A = n$, then we have $\widetilde{A} = \det(A) A^{-1}$. If $\operatorname{rk} A < n-1$, then all M^i_j are zero because any set of $n-1$ columns of A is linearly dependent. If $\operatorname{rk} A = n-1$, then there is a set of $n-1$ columns of A that is linearly independent. Hence $\widetilde{A}$ is non-trivial. Let $\ker A = \langle v \rangle$ and $\ker A^T = \langle w \rangle$. Since $A\widetilde{A} = \widetilde{A}A = 0$, the columns of $\widetilde{A}$ are in $\ker A$ and the rows in $\ker A^T$. Therefore $\widetilde{A} = \lambda \cdot v \cdot w^T$ for some $\lambda \neq 0$.

Proposition 4.4. *Let A' be obtained from A by replacing A^i_j by $A^i_j + x$, let $\Delta = \det(A)$ and $\Delta' = \det(A')$. Δ' and $\widetilde{A}'$ are related to Δ and $\widetilde{A}$ as follows:*

$$\Delta' = \Delta + x \cdot \widetilde{A}^j_i \tag{1}$$

$$\Delta \widetilde{A}' = \Delta' \widetilde{A} - x \widetilde{A}_i \cdot \widetilde{A}^j \tag{2}$$

Proof. Equation (1) is clear by expansion along row i or column j. For equation (2) it will first be proved that $B = \Delta' \widetilde{A} - x\widetilde{A}_i \cdot \widetilde{A}^j$ satisfies $A'B = \Delta\Delta' I_n$. Let X be the zero matrix except for entry $X^i_j = x$ and let $Y = \widetilde{A}^i \cdot \widetilde{A}_j$. We obtain

$$A'B = (A + X)(\Delta'\widetilde{A} - xY) = \Delta\Delta' I_n + \Delta' X\widetilde{A} - x(AY + XY)\,.$$

Define the $n \times n$ matrix Z by $Z_k = \delta_{ik}\widetilde{A}^j$. Now observe that $X\widetilde{A} = xZ$, $XY = x\widetilde{A}^j_i Z$ and $(AY)^k_l = A_k\widetilde{A}_i \cdot \widetilde{A}^j_l = \delta_{ik}\Delta\widetilde{A}^j_l$. So $AY = \Delta Z$ and hence

$$A'B = \Delta\Delta' I_n + \Delta' X\widetilde{A} - x(AY + XY) = \Delta\Delta' I_n + x(\Delta' - \Delta - x\widetilde{A}^j_i)Z = \Delta\Delta' I_n\,.$$

If A and A' are invertible, by uniqueness of inverses this directly implies that $B = \Delta\widetilde{A}'$. The map that sends any matrix A to $\Delta\widetilde{A}'$ and the map that sends A to B as described are both described by polynomials and therefore morphisms of algebraic (affine) varieties. They coincide on the open and hence dense subset of invertible matrices A such that A' is also invertible, so they are equal. □

Observation 4.5. By equations (1) and (2), $\widetilde{A}'$ can be derived from $\widetilde{A}$ and Δ when $\Delta \neq 0$. However, when $\Delta = 0$, it follows that $\Delta'\widetilde{A} = x\widetilde{A}_i \cdot \widetilde{A}^j$. Therefore $\widetilde{A}$ has rank 1, or $\widetilde{A}_k = 0$ and $\widetilde{A}^l = 0$ for any $k, l \in \{1, ..., n\}$ and hence $\widetilde{A} = 0$. So generally we cannot deduce $\widetilde{A'}$ from $\widetilde{A}$ and $\Delta = 0$.

As a last ingredient for what we will need later on, this section ends with a little lemma about adjugates of block diagonal matrices.

Lemma 4.6. *Let $A = B \oplus C$ be the block diagonal matrix with blocks B and C. Then $\widetilde{A} = \det(C)\widetilde{B} \oplus \det(B)\widetilde{C}$.*

Proof. Let $m, n \in \mathbb{N}$ be the sizes of resp. B and C. Let $i, j \in \{1, ..., m + n\}$. If $i, j \leq m$, the matrix obtained from A by deleting row i and column j is the block diagonal sum of B with row i and column j deleted and C. Hence $\widetilde{A}^j_i = \det(C)\widetilde{B}^j_i$. Similarly $i, j < n$ yields $\widetilde{A}^{m+j}_{m+i} = \det(B)(-1)^{m+m}\widetilde{C}^j_i = \det(B)\widetilde{C}^j_i$. Moreover, for $i \leq m < j$ or $j \leq m < i$, the matrix obtained from deleting row i and column j from A has rank $\leq m - 1 + n - 1$. Thus $\widetilde{A}^j_i = 0$. □

5 Fox derivatives

This section will provide an easy way to obtain the Alexander polynomial from a presentation of the link group. Using Fox derivatives, a Jacobian matrix is obtained. The gcd of its cofactors equals Δ_0.

Definition 5.1. Let G be a free group with generators g_i. The *Fox derivative*

with respect to g_i is the function $\frac{\partial}{\partial g_i}\colon G \to \mathbb{Z}[G]$ uniquely defined by

$$\begin{aligned}\frac{\partial}{\partial g_i}(e) &= 0\\ \frac{\partial}{\partial g_i}(g_j) &= \delta_{ij}\\ \frac{\partial}{\partial g_i}(uv) &= \frac{\partial}{\partial g_i}(u) + u\frac{\partial}{\partial g_i}(v)\end{aligned}$$

These axioms imply the following formula for inverses:

$$\frac{\partial}{\partial g_i}(u^{-1}) = -u^{-1}\frac{\partial}{\partial g_i}(u)$$

A proof that the Fox derivatives are well-defined can be found in [12].

Definition 5.2. Let $G = \langle g_1, ..., g_n | r_1, ..., r_m\rangle$ be a finitely presented group. Define the free group $F = \langle g_1, ..., g_n\rangle$ and let $\alpha\colon \mathbb{Z}[F] \to \mathbb{Z}[G^{ab}]$ induced by the canonical projection $F \to G^{ab}$. The *Jacobian matrix* of G related to the presentation $G = \langle g_1, ..., g_n | r_1, ..., r_m\rangle$ is the $n \times m$-matrix A with entries $A^i_j = \alpha(\frac{\partial}{\partial g_i}(r_j))$.

Proofs of the following facts can be found in section 16.2 of [21]. We let $g\colon \mathbb{Z}[G^{ab}] \to \mathbb{Z}[G/\sqrt{[G,G]}]$ be the canonical projection.

Proposition 5.3. *Let G be the fundamental group of a finite CW-complex X. Let A^i denote the i-skeleton of a CW-complex A. Applying g to any Jacobian matrix F of G yields a presentation matrix $g(F)$ of the $g(\mathbb{Z}[G^{ab}])$-module $H_1(\overline{X}, \overline{X}^0)$.*

Remark 5.4. Here $\overline{X}^0 = p^{-1}(X^0)$ is the 0-skeleton of $\overline{X}$, with $p\colon \overline{X} \to X$ the covering map.

Proposition 5.5. *For a finite CW-complex X and $k \geq 1$, we have*

$$\Delta_{k-1}(H_1(\overline{X})) = \Delta_k(H_1(\overline{X}, \overline{X}^0))\,.$$

The point of this section is that by remark 3.10, we have $\overline{X} = X_\infty$ for a link complement X. This can be retracted onto a CW-complex. Then proposition 5.3 applies and proposition 5.5 thus tells us that the Alexander polynomial Δ_0 equals the gcd of all cofactors of a Jacobian matrix corresponding to a group presentation of the link group. This will be exploited in the next section. But first this section concludes with a definition that will be useful in further computations.

Definition 5.6. Let L be an oriented link with group G which has Wirtinger generators $g_1, .., g_n$. Let $\beta\colon G^{ab} \to \langle t\rangle, g_i \mapsto t$ and C the set of cofactors of its Jacobian matrix. The *single-variable Alexander polynomial* of L equals $\gcd(\{\beta(c) : c \in C\})$.

6 The Jacobian-adjugate method for computing the Alexander Polynomial

By definition the cofactors of a Jacobian matrix A of a link group, of which the gcd equals its Alexander polynomial, are precisely the entries in $\widetilde{A}$. In this section we will consider an extended Wirtinger presentation and use the results from section 3 to compute this $\widetilde{A}$. As in section 2, this is done through tangle diagrams.

First the Fox derivatives of all relators involved in tangle diagrams need to be computed, and simplified by applying the map α. This gives the following results:

$$\begin{aligned}
\alpha \circ \frac{\partial}{\partial a}(ab^{-1}) &= 1 \\
\alpha \circ \frac{\partial}{\partial b}(ab^{-1}) &= -ab^{-1} &&= -1 \\
\alpha \circ \frac{\partial}{\partial a}(aba^{-1}c^{-1}) &= 1 - aba^{-1} &&= 1 - b \\
\alpha \circ \frac{\partial}{\partial b}(aba^{-1}c^{-1}) &= 1 - aba^{-1} &&= a \\
\alpha \circ \frac{\partial}{\partial c}(aba^{-1}c^{-1}) &= -aba^{-1}c^{-1} &&= -1
\end{aligned}$$

Note that $\frac{\partial}{\partial a}(x) = 0$ if a is not in the word x and $b = c$ in G^{ab} if $aba^{-1}c^{-1}$ is a relator of G. Normally we would obtain the matrix A by choosing an order of generators and an order of relators. But here, we do not identify the rows and columns with the set $\{0, 1, .., n\}$, but with the set (or a subset) of generators. Hence matrices in this section are represented by tables of the following form:

	t_1	t_2	$\dots$	t_n
t_1	x_{t_1,t_1}	x_{t_1,t_2}	$\dots$	x_{t_1,t_n}
t_2	x_{t_2,t_1}	x_{t_2,t_2}	$\dots$	x_{t_2,t_n}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
t_n	x_{t_n,t_1}	x_{t_n,t_2}	$\dots$	x_{t_n,t_n}

So in this section all matrix invariants of tangle diagrams are (implicitly) defined as maps $S \times S \to \mathbb{Z}[G^{ab}]$ where G is the tangle diagram group and S is the set or a certain subset of the generators in the extended Wirtinger presentation.

Because the rows of a Jacobian matrix are already in a natural correspondence with generators, we only need to define which relator corresponds to which column/generator of A. Any relator of the form ab^{-1}, where a is stitched to b, will correspond to column a and $aba^{-1}c^{-1}$ will correspond to the incoming edge of the under-strand. This is c if the crossing is positive and b if the crossing is negative. In order to proceed from tangle diagrams via disjoint unions and stitchings to a Jacobian matrix of the resulting link, a definition of an alternative Jacobian matrix is needed.

Definition 6.1. The *tangle-Jacobian matrix* of a tangle diagram is formed by the Jacobian matrix of the extended Wirtinger presentation of its group, together with unit columns e_s in column s for outgoing edges s.

For positive and negative crossings this gives us tangle-Jacobian matrices of the form

	a	b	c
a	1	0	$1-b$
b	0	1	a
c	0	0	-1

and

	a	b	c
a	1	0	$1-b$
b	0	1	-1
c	0	0	a

.

The disjoint union of tangle diagrams corresponds with the block sum of their tangle-Jacobian matrices. Stitching i to j means that the unit column e_i is replaced by one representing the relation ij^{-1}. So -1 is added in entry j, i. It should be noted that after this stitching, $i = j$ holds in G^{ab}.

Since the determinants we want to find are only relevant up to multiplication with a unit, we can multiply the third column of the *crossing matrices* above with a unit to obtain the more convenient forms

	a	b	c
a	1	0	$b-1$
b	0	1	$-a$
c	0	0	1

and

	a	b	c
a	1	0	$a^{-1}(1-b)$
b	0	1	$-a^{-1}$
c	0	0	1

.

Now using the results from section 3, we can compute $\widetilde{A}$ by building the link diagram from crossings using disjoint union and stitching. We begin with the crossing matrices. These have the following adjugates:

1	a	b	c
a	1	0	$1-b$
b	0	1	a
c	0	0	1

and

1	a	b	c
a	1	0	$a^{-1}(b-1)$
b	0	1	a^{-1}
c	0	0	1

.

Note that the determinant of the original matrix is written in the upper left corner. We consider what happens to the adjugates when disjoint union and stitching are applied to tangle diagrams and thus obtain operations on pairs $(\Delta, \widetilde{A})$, where $\Delta = \det A$. For the sake of notational convenience, we denote these operations just as the original disjoint union and stitching. By lemma 4.6 and proposition 4.4 this gives

$$\begin{aligned}(\Delta_A, \widetilde{A}) \sqcup (\Delta_B, \widetilde{B}) &= (\Delta_A \cdot \Delta_B, \Delta_B \cdot \widetilde{A} \oplus \Delta_A \cdot \widetilde{B}) \\ m_{ij}((\Delta, \widetilde{A})) &= (\Delta - \widetilde{A}^i_j, (1 - \Delta^{-1}\widetilde{A}^i_j)\widetilde{A} + \Delta^{-1}\widetilde{A}_j \cdot \widetilde{A}^i)\end{aligned}$$

provided that $\Delta \neq 0$. In this way, by starting with the adjugate crossing matrices and computing the new adjugate matrix after every disjoint union or stitching, we finally obtain $\widetilde{A}$, where A is the Jacobian matrix associated to the extended Wirtinger presentation of the link diagram. Now the Alexander polynomial is the greatest common denominator of all entries of $\widetilde{A}$. As mentioned in

observation 4.3, all columns of $\widetilde{A}$ are multiples of the same vector v that spans $\ker A$, as well as all rows are multiples of some vector w that spans $\ker A^T$. So we can deduce the entire matrix $\widetilde{A}$ if we know just one column and one row of it. Moreover, we know from proposition 2.17 that we can delete any relator and hence any column from the Jacobian matrix A. The minors from the resulting matrix are all entries of the corresponding row from $\widetilde{A}$. So only one row needs to be computed. And if we use just one variable (e.g. in the case of a knot), we can immediately see that $\ker A^T = \langle 1, 1, ..., 1 \rangle$. Hence in this case any entry of $\widetilde{A}$ gives us the Alexander polynomial.

From the foregoing discussion, we can apply the following reduction, which leaves us with only one row or one entry in the end. In the multivariable case, a lot of rows are superfluous, either already from the start, or at some point after stitching strands. We can delete the third rows from the adjugate crossing matrices, to obtain these *crossing tables*:

$$\begin{array}{c|ccc} 1 & a & b & c \\ \hline a & 1 & 0 & 1-b \\ b & 0 & 1 & a \end{array} \quad \text{and} \quad \begin{array}{c|ccc} 1 & a & b & c \\ \hline a & 1 & 0 & a^{-1}(b-1) \\ b & 0 & 1 & a^{-1} \end{array}.$$

And after stitching (or rather whilst stitching, to save calculation time) strand i to strand j, we can leave out row i of $\widetilde{A}$. It is a matter of inspection to see that these reductions can be done with no harm to the rest of the calculation. When there is only one stitching left, the matrix has one row and the Alexander polynomial equals the gcd of all of its entries. In the single-variable case, it gets even easier. The crossing tables are further reduced by deleting the second column. Replacing a and b by a single variable t gives

$$\begin{array}{c|cc} 1 & a & c \\ \hline a & 1 & 1-t \\ b & 0 & t \end{array} \quad \text{and} \quad \begin{array}{c|cc} 1 & a & c \\ \hline a & 1 & 1-t^{-1} \\ b & 0 & t^{-1} \end{array}.$$

After stitching strand i to strand j, both row i and column j of $\widetilde{A}$ are left out. The Alexander polynomial equals the single remaining entry when there is only one stitching left. This entry also equals the determinant at that stage, since the last stitching turns the diagram into a link diagram, which has a Jacobian matrix with determinant 0. We summarize all of this in the following definition and theorem.

Definition 6.2. Let T be a tangle diagram with group G. The *Jacobian-adjugate* $JA(T)$ is a pair (Δ, A) with $\Delta_A \in \mathbb{Z}[G^{ab}]$ and $A \in \mathrm{Mat}(\mathbb{Z}[G^{ab}])$. It is defined recursively by disjoint union and stitching as follows:

$$\begin{aligned} (\Delta_A, A) \sqcup (\Delta_B, B) &= (\Delta_A \cdot \Delta_B, \Delta_B \cdot A \oplus \Delta_A \cdot B) \\ m_{ij}((\Delta, A)) &= (\Delta - A^i_j, (1 - \Delta^{-1} A^i_j) A^{\widehat{i}} + \Delta^{-1} A^{\widehat{i}}_j \cdot A^i)\,. \end{aligned}$$

The hats ($\widehat{\ }$) indicate which row or column is omitted. The Jacobian-adjugate of a diagram with only one vertex equals $(1, 1)$ and stitchings involving such

a vertex are not allowed (nor needed); positive and negative crossings with over-strand a and outgoing and incoming under-strand edges b and c have as Jacobian-adjugates (Δ is written in the upper left corner)

$$\begin{array}{c|cc} 1 & a & b \\ \hline a & 1 & 0 \\ b & 0 & 1 \end{array} \quad \begin{array}{c} c \\ \hline 1-b \\ a \end{array} \quad \text{and} \quad \begin{array}{c|cc} 1 & a & b \\ \hline a & 1 & 0 \\ b & 0 & 1 \end{array} \quad \begin{array}{c} c \\ \hline a^{-1}(b-1) \\ a^{-1} \end{array} .$$

The *single-variable Jacobian-adjugate* svJA is a pair (Δ_A, A) with $\Delta_A \in \mathbb{Z}[t]$ and $A \in \mathrm{Mat}(\mathbb{Z}[t])$ and uses

$$\begin{array}{c|cc} 1 & a & c \\ \hline a & 1 & 1-t \\ b & 0 & t \end{array} \quad \text{and} \quad \begin{array}{c|cc} 1 & a & c \\ \hline a & 1 & 1-t^{-1} \\ b & 0 & t^{-1} \end{array}$$

for positive and negative crossings. It satisfies

$$\begin{aligned} (\Delta_A, A) \sqcup (\Delta_B, B) &= (\Delta_A \cdot \Delta_B, \Delta_B \cdot A \oplus \Delta_A \cdot B) \\ m_{ij}((\Delta, A)) &= (\Delta - A^i_j, (1 - \Delta^{-1} A^i_j) A^{\widehat{i}}_{\widehat{j}} + \Delta^{-1} A^{\widehat{i}}_j \cdot A^i_{\widehat{j}}) . \end{aligned}$$

Theorem 6.3. *Let T be a tangle diagram such that one stitching would turn it into a link diagram of a link L. The Alexander polynomial of L equals the* gcd *of the $1 \times n$ matrix of* JA(T). svJA(T) *is a pair (Δ, A) with A a 1×1 matrix for which both Δ and A^1_1 equal the single-variable Alexander polynomial of L.*

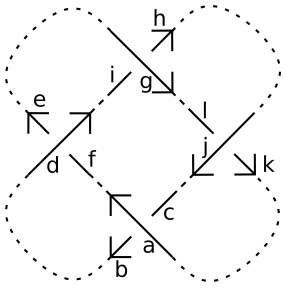


Figure 7: Solomon's knot

Example 6.4. Let us again consider the Solomon's knot (figure 7). We can first compute the tangle-Jacobian adjugate of one half of its diagram. This starts with taking the disjoint union of two positive crossing tables. They differ by an exchange of the variables we will use, s and t.

1	a	b	c	d	e	f
a	1	0	$-t+1$	0	0	0
b	0	1	s	0	0	0
d	0	0	0	1	0	$-s+1$
e	0	0	0	0	1	t

Stitching a to f and b to d gives in succesion:

1	a	b	c	d	e	f
b	0	1	s	0	0	0
d	$-s+1$	0	$ts-t-s+1$	1	0	$-s+1$
e	t	0	$-t^2+t$	0	1	t

1	a	b	c	d	e	f
d	$-s+1$	1	$ts-t+1$	1	0	$-s+1$
e	t	0	$-t^2+t$	0	1	t

.

Since the other half looks exactly the same, we can take the disjoint union of this matrix with itself.

1	a	b	c	d	e	f	g	h	i	j	k	l
d	$-s+1$	1	$ts-t+1$	1	0	$-s+1$	0	0	0	0	0	0
e	t	0	$-t^2+t$	0	1	t	0	0	0	0	0	0
j	0	0	0	0	0	0	$-s+1$	1	$ts-t+1$	1	0	$-s+1$
k	0	0	0	0	0	0	t	0	$-t^2+t$	0	1	t

Then we stitch f to i, e to g and j to c. This eventually results in the transpose of

$-t^2s^2+t^2s-ts+t$	k
a	$-t^2s^2+t^2s-ts+t$
b	$-t^3s+t^2s-t^2+t$
c	$-t^3s+t^2s-t^2+t$
d	$-t^3s+t^2s-t^2+t$
e	$-t^2s^2+t^2s-ts+t$
f	$-t^2s^2+t^2s-ts+t$
g	$-t^2s^2+t^2s-ts+t$
h	$-t^3s+t^2s-t^2+t$
i	$-t^3s+t^2s-t^2+t$
j	$-t^3s+t^2s-t^2+t$
k	$-t^2s^2+t^2s-ts+t$
l	$-t^2s^2+t^2s-ts+t$

.

Therefore its Alexander polynomial equals $\gcd(-t^2s^2+t^2s-ts+t, -t^3s+t^2s-t^2+t) = ts+1$, just as we saw before.

7 Equivalence to rMVA and similarity with Γ-calculus

The Jacobian-adjugate method to compute the Alexander polynomial was inspired by the tangle invariants of Bar-Natan ([7], [6]) and Halacheva ([13]). They both work with the two operations disjoint union and stitching and at each stage the invariant is a table with outgoing strands as row labels and incoming strands as column labels, together with some Δ written in the upper

left corner. Bar-Natan's Γ-calculus has the following tables for positive and negative crossings:

$$\begin{array}{c|cc} 1 & a & c \\ \hline a & 1 & 1-a \\ b & 0 & a \end{array} \text{ and } \begin{array}{c|cc} 1 & a & c \\ \hline a & 1 & 1-a^{-1} \\ b & 0 & a^{-1} \end{array}.$$

Here we still let a be (the variable belonging to) the over-strand and b and c (the variables belonging to) the (outgoing resp. incoming) under-strands. Computation works as follows:

$$\begin{aligned} (\Delta_A, A) \sqcup^\Gamma (\Delta_B, B) &= (\Delta_A \cdot \Delta_B, A \oplus B) \\ m^\Gamma_{ij}(\Delta, A) &= (\Delta(1-A^i_j), A^{\widehat{i}}_{\widehat{j}} + (1-A^i_j)^{-1} A^{\widehat{i}}_j \cdot A^i_{\widehat{j}}) \end{aligned}$$

Proposition 7.1. *The Γ-calculus computes a part of the inverse of an adjusted tangle-Jacobian matrix of the extended Wirtinger presentation, and this adjusted tangle-Jacobian matrix equals the actual Jacobian matrix in the single-variable case.*

Proof. We form the matrix A in exact the same way as the tangle-Jacobian of the Jacobian-adjugate method, except for the crossing matrices: instead of using

$$\begin{array}{c|ccc} & a & b & c \\ \hline a & 1 & 0 & b-1 \\ b & 0 & 1 & -a \\ c & 0 & 0 & 1 \end{array} \text{ and } \begin{array}{c|ccc} & a & b & c \\ \hline a & 1 & 0 & a^{-1}(1-b) \\ b & 0 & 1 & -a^{-1} \\ c & 0 & 0 & 1 \end{array}$$

for positive and negative crossings, we replace all variables b by a to obtain

$$\begin{array}{c|ccc} & a & b & c \\ \hline a & 1 & 0 & a-1 \\ b & 0 & 1 & -a \\ c & 0 & 0 & 1 \end{array} \text{ and } \begin{array}{c|ccc} & a & b & c \\ \hline a & 1 & 0 & a^{-1}-1 \\ b & 0 & 1 & -a^{-1} \\ c & 0 & 0 & 1 \end{array}.$$

Hence in the crossing tables we also replace any variable a by b, which yields exactly (after reduction to 2×2 tables)

$$\begin{array}{c|cc} 1 & a & c \\ \hline a & 1 & 1-a \\ b & 0 & a \end{array} \text{ and } \begin{array}{c|cc} 1 & a & c \\ \hline a & 1 & 1-a^{-1} \\ b & 0 & a^{-1} \end{array}.$$

Here it is possible to leave out columns and delete columns after stitching, because even in the multivariable case it holds that $\ker A^T = \langle 1, 1, ..., 1 \rangle$. But it is not clear whether leaving out rows causes a loss of information.

Let Ω be the set of pairs (Δ, A) with $\Delta \in \mathrm{Frac}(R)$ nonzero, A a square matrix over R and $R = \mathbb{Z}[t_1, t_2, ...t_n]$. We will use the following transformation:

$$\phi : \Omega \to \Omega, (\Delta, A) \mapsto (\Delta, \Delta A).$$

For the moment, we do not remove any rows or columns. Since for invertible matrices A it holds that $\widetilde{A} = \det A \cdot A^{-1}$ and it is clear that $\phi(C) = C$ for C a crossing table as above, the proposition now comes down to proving that the following diagrams commute:

$$\begin{array}{ccc} \Omega \times \Omega & \xrightarrow{\sqcup^\Gamma} & \Omega \\ {\scriptstyle \phi \times \phi}\downarrow & & \downarrow{\scriptstyle \phi} \\ \Omega \times \Omega & \xrightarrow{\sqcup} & \Omega \end{array} \qquad \begin{array}{ccc} \Omega & \xrightarrow{m_{ij}^\Gamma} & \Omega \\ {\scriptstyle \phi}\downarrow & & \downarrow{\scriptstyle \phi} \\ \Omega & \xrightarrow{m_{ij}} & \Omega \end{array}$$

The first one is easy and needs no writing down; elaborating the second one gives

$$\begin{aligned} (\phi \circ m_{ij}^\Gamma)((\Delta, A)) &= \phi((\Delta(1 - A_j^i), A + (1 - A_j^i)^{-1} A_j \cdot A^i)) \\ &= (\Delta - \Delta A_j^i, \Delta(1 - A_j^i)A + \Delta A_j \cdot A^i) \\ (m_{ij} \circ \phi)((\Delta, A)) &= m_{ij}((\Delta, \Delta A)) \\ &= (\Delta - \Delta A_j^i, (1 - A_j^i)\Delta A + \Delta A_j \cdot A^i)\,. \end{aligned}$$

□

Example 7.2. Applying Γ-calculus to the Solomon's knot we saw before in example 2.11 gives us the polynomial $-ts^2 + ts - s + 1$. However, if our last stitching was k to a instead of j to c, we would have obtained $-t^2 s + ts - t + 1$ and the gcd of these equals the Alexander polynomial.

Question 7.3. It is not clear if there is a topological interpretation of the Γ-calculus, since it generally does not compute the multivariable Alexander polynomial. But we can try to do multivariable Γ-calculus without deleting rows. This means also using 3×2 crossing tables

$$\begin{array}{c|cc|c} 1 & a & b & c \\ \hline a & 1 & 0 & 1-a \\ b & 0 & 1 & a \end{array} \text{ and } \begin{array}{c|cc|c} 1 & a & b & c \\ \hline a & 1 & 0 & 1-a^{-1} \\ b & 0 & 1 & a^{-1} \end{array},$$

obtained just as the 2×2 tables in the proof of proposition 7.1. Investigating a few simple links, I have not found an example for which the gcd of the last $n \times 1$ matrix does not equal the Alexander polynomial. Will such examples exist, or can a proof be found that this also computes the Alexander polynomial?

Let us now consider Halacheva's rMVA. It has as its crossing tables

$$a^{-\frac{1}{2}} \cdot \begin{array}{c|cc} a & a & c \\ \hline a & -a & 0 \\ b & 1-b & -1 \end{array} \text{ and } a^{-\frac{1}{2}} \cdot \begin{array}{c|cc} 1 & a & c \\ \hline a & -1 & 0 \\ b & b-1 & -a \end{array}.$$

For disjoint union and stitching, these rules apply:

$$\begin{aligned}(\Delta_A, A) \sqcup^H (\Delta_B, B) &= (\Delta_A \cdot \Delta_B, \Delta_B \cdot A \oplus \Delta_A \cdot B)\\ m^H_{ij}((\Delta, A)) &= (\Delta + A^i_j, A^{\widehat{i}}_{\widehat{j}} + \Delta^{-1}(A^i_j A^{\widehat{i}}_{\widehat{j}} - A^{\widehat{i}}_j \cdot A^i_{\widehat{j}}))\end{aligned}$$

Proposition 7.4. *rMVA is equivalent to computing a part of the cofactor matrix of the tangle-Jacobian matrix A of an extended Wirtinger presentation.*

Proof. Starting with an extended Wirtinger presentation of a link projection, we first reverse the direction of all generators. This means that any relator of the form $aba^{-1}c^{-1}$ is replaced by $a^{-1}b^{-1}ac$, or equivalently $aca^{-1}b^{-1}$. So this interchanges the roles of the the two generators b and c around the understrand of a crossing. So our matrices (from section 6) for positive and negative crossings are swapped. Secondly, we reverse all strands (without reversing the generators back) in order to stitch in the opposite direction. To be able to do so, we also swap the row/column labels of the under-strand (b and c). Altogether the crossing tables of positive and negative crossings now look like

1	a	c	b
a	1	0	$a^{-1}(b-1)$
c	0	1	a^{-1}
b	0	0	1

and

1	a	c	b
a	1	0	$1-b$
c	0	1	a
b	0	0	1

.

But we are interested in the cofactor matrix, so we transpose these blocks to obtain

1	a	c	b
a	1	0	0
c	0	1	0
b	$a^{-1}(b-1)$	a^{-1}	1

and

1	a	c	b
a	1	0	0
c	0	1	0
b	$1-b$	a	1

.

Remember that we reversed all stitchings, so the original stitching rule now actually applies to the cofactor matrix. Now we delete row c (by which we might loose information) and column b, which yields

1	a	c
a	1	0
b	$a^{-1}(b-1)$	a^{-1}

and

1	a	c
a	1	0
b	$1-b$	a

.

Note that the following equalities hold for a nonzero factor λ and pairs (A, Δ_A) and (B, Δ_B):

$$\begin{aligned}(\lambda(\Delta_A, A)) \sqcup (\Delta_B, B) &= \lambda((\Delta_A, A) \sqcup (\Delta_B, B))\\ m_{ij}(\lambda(\Delta, A)) &= \lambda m_{ij}(\Delta, A)\,.\end{aligned}$$

Hence it is no problem to multiply the crossing tables we obtained with the units $a^{\frac{1}{2}}$ and $a^{-\frac{1}{2}}$ and obtain

$a^{-\frac{1}{2}} \cdot$

a	a	c
a	a	0
b	$b-1$	1

and $a^{-\frac{1}{2}} \cdot$

1	a	c
a	1	0
b	$1-b$	a

.

The last step is the transformation

$$\phi : \Omega \to \Omega, (\Delta, A) \mapsto (\Delta, -A)\,,$$

for which it is clear that it produces an equivalence between the method as described above and rMVA without deleting rows and columns. □

Example 7.5. Computing rMVA of the Solomon's knot of example 2.11 gives $-ts^2 + ts - s + 1$. Again, as in example 7.2, if our last stitching was k to a instead of j to c, we would have obtained $-t^2s + ts - t + 1$ and the gcd of these equals the Alexander polynomial.

8 Proteins & the Protein Data Bank

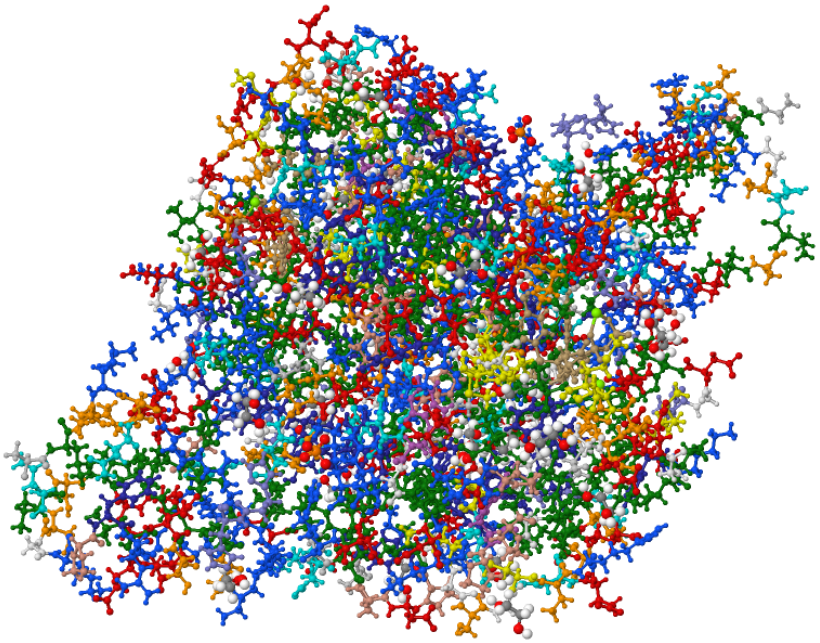


Figure 8: a protein, colored by amino acid

Proteins are macromolecules that are present in all living organisms. They consist of one or multiple chains, which are formed of many amino acids. The structure of amino acids looks like figure 9, where C stands for a carbon atom, H for hydrogen, N for nitrogen and O for oxygen. Multiple atoms form the organic side chain R, unique to each amino acid. Atoms are bonded by covalent bonds, which means that each pair of bonded atoms shares one or more pairs of electrons. The amino acids in a protein are joined together by so-called peptide bonds: covalent bonds between a nitrogen atom of one amino acid and a carbon atom of another amino acid (figure 10).

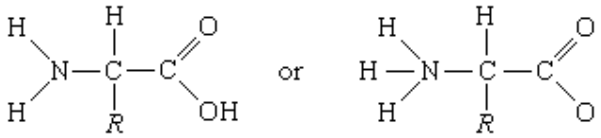


Figure 9: general structure of amino acids, from [15]

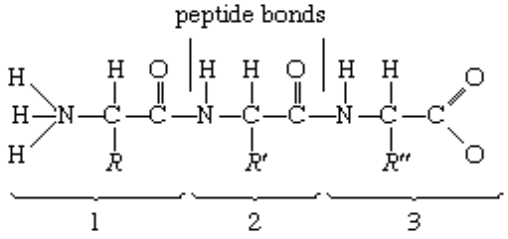


Figure 10: three amino acids joined by peptide bonds, from [15]

This results in one or multiple backbones (figure 11) of the protein formed by carbon and nitrogen atoms, where each amino acid contributes two carbon atoms and one nitrogen atom. As we can see in figure 10, oxygen and hydrogen atoms and side chains are attached to such a backbone, and there is one side chain for each amino acid.

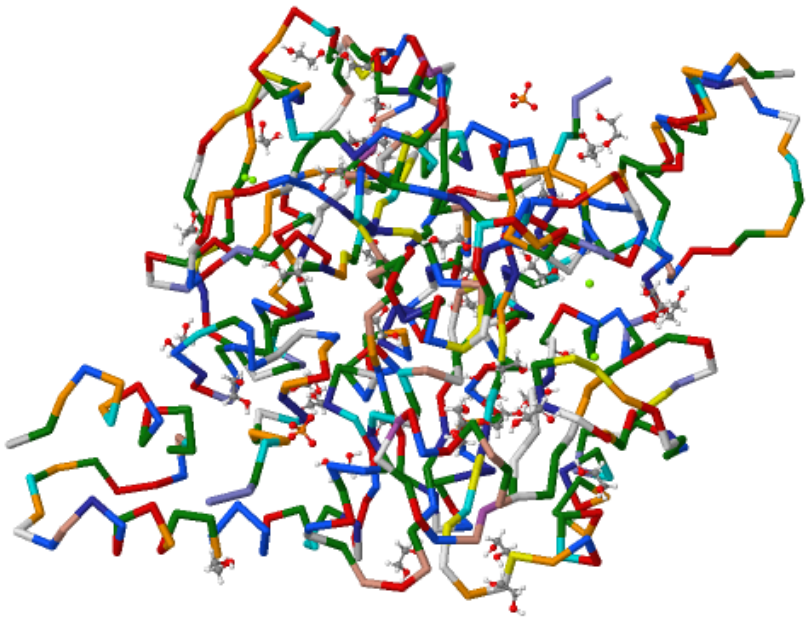


Figure 11: backbone of the protein from figure 8

In some proteins, disulfide bonds also occur, connecting different side chains by a covalent bond between two sulfur atoms. Another type of atom bonding that is

weaker, but also frequently present in proteins, is hydrogen bonding. One atom of the pair, generally an oxygen, nitrogen or fluorine atom, is covalently bonded to a hydrogen atom, which by an unequal distribution of electrons has a slight positive charge. The other atom of the pair, also oxygen, nitrogen or fluorine, has an unshared electron pair and is therefore slightly negatively charged. This results in electrostatic attraction between the latter and the hydrogen atom.

The distance between atoms in a covalent bond depends mainly on the kind of atoms involved and the bond order, i.e. the number of electron pairs they share. For determining bonds from just the coordinates of atoms, I will use the covalent radii of atoms [22]. As mentioned in [9], the sum of the covalent radii of two atoms determines the bond length. I will therefore consider atoms as connected by a covalent bond if the distance between them is less than the sum of their covalent radii plus ± 10 percent. Following [16] (page 12), the program will detect hydrogen bonds (if the right atoms are involved and it is not a covalent bond) at a distance smaller than 0.25 nm. In line with VSEPR theory [5], hydrogen bonds are only considered to exist when any angle formed by the two atoms of the possible hydrogen bond and an atom that is connected to one of those, is at least 90 degrees.

The information I need about the 3-dimenional structure of proteins comes from the RCSB Protein Data Bank (PDB) [18]. This is a repository of 10 thousands of files, which each list the coordinates of all atoms of some unique protein. Such a PDB file also includes to which amino acid any atom belongs and in which order these amino acids form one or more chains.

9 Computing Alexander polynomials of proteins

As mentioned in the introduction, knots in proteins have been investigated before ([2] and [20]); in these investigations the Alexander polynomial was also used. The added value of this thesis in this context is that also hydrogen bonds are considered. Therefore, in contrast to much other research on this topic, it does not depend on various projections or closures of links, which may result in a knot that is strange to the protein.

Before explaining what methods were used to compute an Alexander polynomial of proteins, first something should be said about the spatial structure of proteins and how it can and cannot be used to compute Alexander polynomials. A protein with coordinates for each atom can mathematically be considered as a spatial graph (a graph embedded in $\mathbb{R}^3$) with various types of vertices and various types of edges, i.e. with a graph labeling. Topologically, there is a straightforward extension of the Alexander polynomial to spatial graphs using the same definitions as before. However, one can see that the existence of a vertex with at least 3 edges will imply that there is a representation of its group with at least 2 more generators than relators. Therefore its Alexander ideal is 0.

One way to bypass this problem is looking at certain cycles in the graph, which will be done in the first method. Another way is replacing the graph by a closely related link, which will be done in the second method. This can be done by replacing edges by two parallel lines (possibly twisted around each other) and vertices by a certain tangle that connects the edges. This should be done in a way that is independent of different choices that can be made, such as a projection. Ideally, it would also be invariant under topologically trivial deformations. But assuming that the structure has a certain degree of rigidity, this condition can be loosened. In the case of proteins, bond distances will indeed not vary and neither do angles between bonds [5].

The first method that will be used to compute an Alexander polynomial of proteins, is by detecting the longest cycles formed by any amino chain and one hydrogen bond. As a cycle embedded into $\mathbb{R}^3$, this has a well-defined single-variable Alexander polynomial. If there are multiple chains, it also computes a single-variable Alexander polynomial of the link formed by these cycles, using the standard directions of the chains as given in the PDB files.

The second method I will use is as follows: for each amino acid, I take one point located at the carbon atom in the middle. The (first) backbone is formed by drawing straight lines between consequent amino acids. A parallel (second) backbone is formed by taking points located at the first carbon atom of each side chain. The amino acid glycine has no side chain starting with a carbon atom, so the location of the point is taken as the average of the coordinates of the outer carbon and nitrogen atom in the backbone part of the amino acid, and then reflecting it in the middle carbon atom. Connecting the two parallel backbones at their ends gives a simple loop. Each hydrogen bond connects two amino acids and is therefore taken as an elongated loop that goes around the second backbone once at both amino acids. This loop is located close to the straight line segment between the two amino acids involved. Each ending loops positively around the second backbone at the amino acid it is connected to. If one amino acid has hydrogen bridges to multiple other amino acids, the order of hydrogen bonds is determined by the angle it makes with the (average) direction of the second backbone at that point. From the resulting link, a single-variable Alexander polynomial is computed. More technical details will follow in the rest of this section.

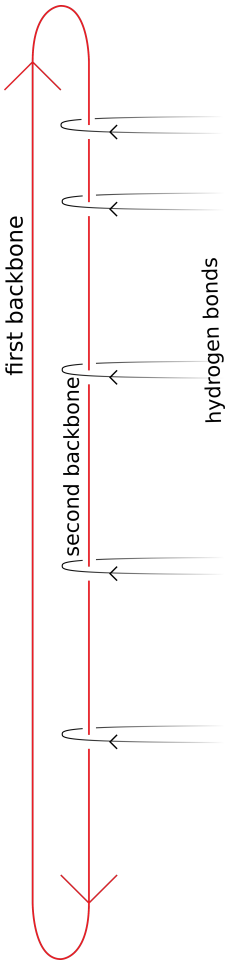
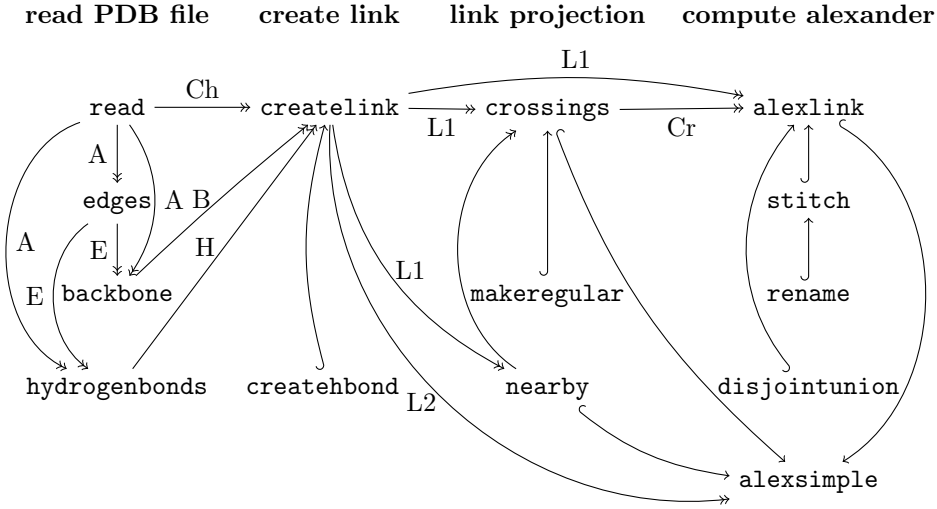


Figure 12: schematic picture of the constructed graph

The code (see the appendix) used to compute the Alexander polynomials of the proteins is divided over four Sage files plus a worksheet that runs the code for each protein.

It is best summarized in the following diagram of functions, in which each column contains the functions of one Sage file, indicated by its column head. I use the monomorphism arrows (↪) to indicate that a certain function is called inside another function, and epimorphism arrows (↠) to denote that the output of the first function is needed as input for the second one. The latter are provided with a symbol which indicates what objects are involved.



Symbols

A: a list of atoms with coordinates
B: the first & the second backbone
Ch: the indices of amino acids that form the end of a chain
Cr: a list of crossings for each line segment
& a list of signs of these crossings
E: a list of covalent bonds
H: a list of hydrogen bonds & a list that indicates for
all amino acids if there is a hydrogen bond attached to it
L1: the link formed by the second method described in this chapter
L2: the (first) backbone(s), each cut off before its first and after
its last hydrogen bond & a corresponding list of hydrogen bonds

The function `read` needs a PDB file as input, which is a textfile using certain key words to indicate what sort of information follows on the rest of the line. The function `alexlink` computes the single-variable Alexander polynomial of a given link, which is returned as a list of coefficients. The function `alexsimple` uses `alexlink` to compute the single-variable Alexander polynomials of a protein as described in the first method in this chapter. If only this method is applied, all arrows with L1 can be omitted.

What follows are explanations of some methods that are used to extract infor-

mation, create a link and compute the Alexander polynomial from it. Beside this, the code in the appendices is provided with some comments.

9.1 Creating hydrogen bonds

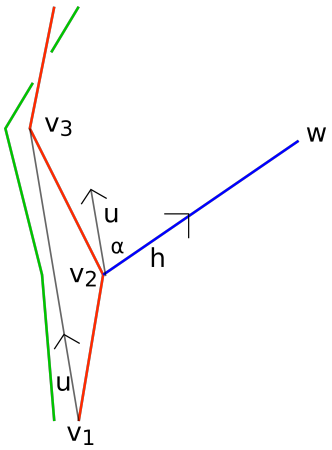


Figure 13: a first backbone (green), a parallel backbone (red) and a hydrogen bond (blue)

The loops representing a hydrogen bond are created as hexagons, of which three points are located at each amino acid it connects. As described earlier on, an amino acid may have hydrogen bonds to multiple other amino acids. The order of these hydrogen atoms along the backbone is then determined by the cosine of the angle α as in figure 13, where u and h are normalized vectors. If $\cos\alpha > 0$, we let $r = \frac{v_3 - v_2}{||v_3 - v_2||}$ and $p = \epsilon_1 \cdot r \cdot \cos\alpha$ for some fixed small $\epsilon_1 > 0$. As a vector starting at v_2 it ends somewhere on the line segment v_2v_3 This endpoint is the new starting point s of the hydrogen bridge. Now we need to define three points that determine one side of the hexagon. We take some small fixed $\epsilon_2 > 0$ and let $q_2 = -\epsilon_2 \cdot h$, $q_3 = q_2 \times r$ and $q_1 = -q_3$. Thus we obtain three points in a plane perpendicular to $v_3 - v_2$. Adding v_2 to these points makes q_2 collinear with v_2 and w, and the tetragon $(wq_1q_2q_3)$ loops positively around the parallel backbone, assuming its direction is upwards. Finally adding p to q_1, q_2 and q_3 brings them in the right position. If $\cos\alpha < 0$ this all goes similarly, using v_1 instead of v_3 and changing some signs.

9.2 Making a projection regular

The function `crossings` starts with calling the function `makeregular`. Let $p\colon \mathbb{R}^3 \to \mathbb{R}^2, (x, y, z) \mapsto (x, y)$ be the projection and S the set of line segments in the link. `makeregular` lists all directions of $\{p(s) : s \in S\}$. If one of these equals 0, it shifts one of the original endpoints a little to make this direction nonzero. If two directions coincide, it checks whether the two line segments are actually on one line. If so, it slightly moves one of the original endpoints, causing the lines to not be collinear. It thus ensures that the projection is injective up to a finite number of points. The function `crossings` further adapts coordinates to make the projection regular if needed, as described in the next subsection.

9.3 Computing crossings

In the function `crossings`, all crossings in the projection $p\colon \mathbb{R}^3 \to \mathbb{R}^2, (x, y, z) \mapsto (x, y)$ are computed. So let $a, b, c, d \in \mathbb{R}^3$ with $a = (a_1, a_2, a_3)$, $b = (b_1, b_2, b_3)$,

$c = (c_1, c_2, c_3)$ and $d = (d_1, d_2, d_3)$. Define a', b', c' and d' as the projections of a, b, c and d respectively. We want to compute whether the line segments $(a'b')$ and $(c'd')$ intersect. So we will solve $\lambda b' + (1-\lambda)a' = \mu d' + (1-\mu)c'$. But first we use a sort of prefilter to rule out a lot pairs of such line segments that do not cross. For this, we let m be the maximal length of any projected line segment. If $(a'b')$ and $(c'd')$ intersect, the distance from c' and d' to $(a'b')$ is at most m. Now consider the minimum of the distances between c' and either a' or b'. This minimum is maximal if c is on the perpendicular bisector of $(a'b')$. Hence it is at most $\sqrt{m^2 + (\frac{1}{2}m)^2} = \sqrt{\frac{5}{4}}m$. This is used by the function nearby, which lists all unconnected points at a distance no bigger than $\sqrt{\frac{5}{4}}m$ for each vertex. So the function `crossings` checks if c' and d' are close enough to either a' or b' to possibly cause an intersection. If so, we continue with solving $\lambda b' + (1-\lambda)a' = \mu d' + (1-\mu)c'$.

$$\begin{aligned}
\lambda \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} + (1-\lambda) \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} &= \mu \begin{pmatrix} d_1 \\ d_2 \end{pmatrix} + (1-\mu) \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} \\
\lambda \begin{pmatrix} b_1 - a_1 \\ b_2 - a_2 \end{pmatrix} &= \begin{pmatrix} \mu(d_1 - c_1) + c_1 - a_1 \\ \mu(d_2 - c_2) + c_2 - a_2 \end{pmatrix} \\
(b_1 - a_1)(\mu(d_2 - c_2) + c_2 - a_2) &= (b_2 - a_2)(\mu(d_1 - c_1) + c_1 - a_1) \\
\mu((d_2 - c_2)(b_1 - a_1) - (d_1 - c_1)(b_2 - a_2)) &= (b_2 - a_2)(c_1 - a_1) - (b_1 - a_1)(c_2 - a_2)
\end{aligned}$$

Now because the function `makeregular` was executed, the two line segments do not intersect if they are parallel. Assuming they are not parallel, we now obtain a unique μ. If $\mu \notin [0,1]$, the two line segments do not intersect. If $\mu \in [0,1]$, λ can easily be computed. If $\lambda \in [0,1]$, the two line segments intersect. The largest of $\lambda b_3 + (1-\lambda)a_3$ and $\mu d_3 + (1-\mu)c_3$ indicates the over-strand. If $\mu \in \{0,1\}$ or $\lambda \in \{0,1\}$, it is harder to determine whether a strand really crosses another strand or not. Therefore in these cases the coordinates are slightly adapted in order to deal with this. If $\lambda = 0$, then a' lies on $(c'd')$; so we push a' along the line segment $(a'b')$ by adding $\epsilon \cdot (b' - a')$. In this way, $(a'b')$ and $(c'd')$ do not intersect. The other cases go similarly. Now each point in the list is at the edge of two line segments. But in any case where such a point lies on another line segment, the point is shifted at the first moment the algorithm uses a line segment it adjoins. So it does not save a crossing that is removed later on, provided ϵ is small enough.

9.4 Computing the Alexander polynomial

The function `alexlink` starts with a 0×0 matrix and applies disjoint union with a new crossing table every time the next stitching to be done is a stitching to a crossing not encountered before. In this way, the size of the matrix is - up to stitching order - minimal at each stage. The function loops around all link components, and for each link component it works (almost) simultaneously in two directions, starting from one single point and stitching the link component

together. When working with a double backbone, the benefit of this strategy is that all crossings these two backbones have with each other do not leave us with a lot of leaves in the tangle diagram when the link component is not yet completed. Since the size of the matrix is determined by the number of leaves, this saves calculation time.

Whenever a stitching would cause the determinant to be 0, it is saved to be done at the end. There it will be checked if the remaining stitchings can be done without causing the determinant to be zero. Except for the very last stitching of course, since this completes the Jacobian matrix, of which the columns are linearly dependent.

To save calculation time, all polynomials are computed modulo some prime number p. In the resulting polynomial, all coefficients $c > \frac{p}{2}$ are represented as the negative number $c - p$. If p is large enough (at least twice as large as the absolute value of any coefficient), the resulting polynomial will hence be equal to the actual Alexander polynomial.

10 Results

The first method of computing Alexander polynomials was applied to a random sample of 1197 PDB files. Many other files, however, could not be processed. This is mostly because many PDB files do not list hydrogen atoms and partly because other files may be incomplete or the program still needs improvement. Almost all created links were trivial, except for 1 protein which had two components linked as a Hopf link and single-variable Alexander polynomial $t - 1$. All polynomials were computed modulo $p = 101$; altogether computing took several hours for an Intel Core i3 processor.

The Alexander polynomials following from the second method get very large, illustrated by example 10.1. This protein is very small compared to almost all other proteins. For larger proteins, computing the Alexander polynomial takes several hours on an Intel Core i3 processor. Example 10.1 was computed modulo both primes within half a minute. Computing its Alexander polynomial over $\mathbb{Z}$, however, was problematic and could not be completed within an hour.

Example 10.1. PDB entry 2L5R is a very small protein that looks like figure 14. Its backbone can be seen in figure 15. In figures 16 and 17 we can see what the link from the second method looks like. The red component comes from its double backbone and the blue components from hydrogen bonds. Its Alexander polynomial was equal modulo the primes 252097800623 and 252097800629, so

we can conclude that this is its actual Alexander polynomial:

$$
\begin{aligned}
&343\,t^{30} - 2604\,t^{29} + 9033\,t^{28} - 18957\,t^{27} + 24080\,t^{26} - 317\,t^{25} - 102080\,t^{24} \\
&+ 336845\,t^{23} - 685343\,t^{22} + 1002851\,t^{21} - 1098892\,t^{20} + 890069\,t^{19} \\
&- 420747\,t^{18} - 257431\,t^{17} + 1087546\,t^{16} - 1823833\,t^{15} + 2024256\,t^{14} \\
&- 1439692\,t^{13} + 371832\,t^{12} + 529768\,t^{11} - 840407\,t^{10} + 641465\,t^{9} - 287205\,t^{8} \\
&+ 41796\,t^{7} + 44884\,t^{6} - 41983\,t^{5} + 19319\,t^{4} - 5384\,t^{3} + 838\,t^{2} - 49\,t - 1\,.
\end{aligned}
$$

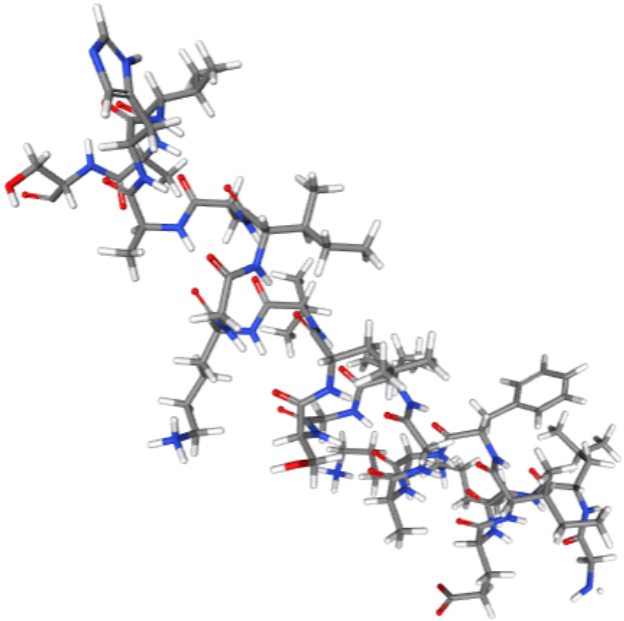


Figure 14: a small protein

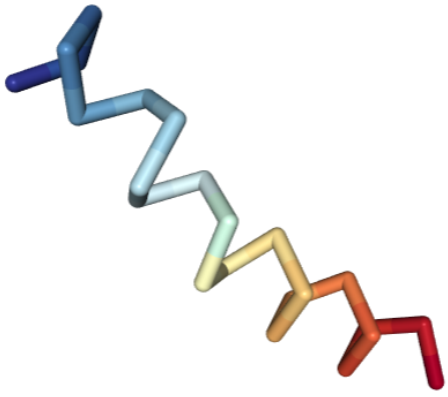


Figure 15: backbone of the protein

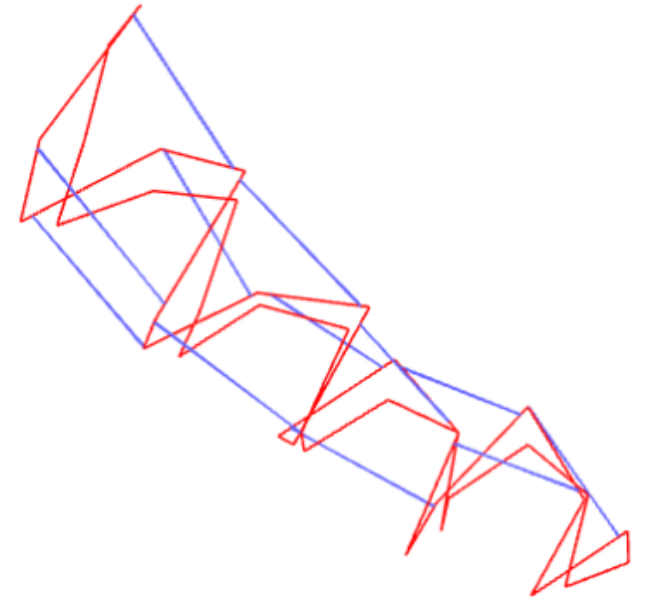


Figure 16: corresponding link formed by the second method

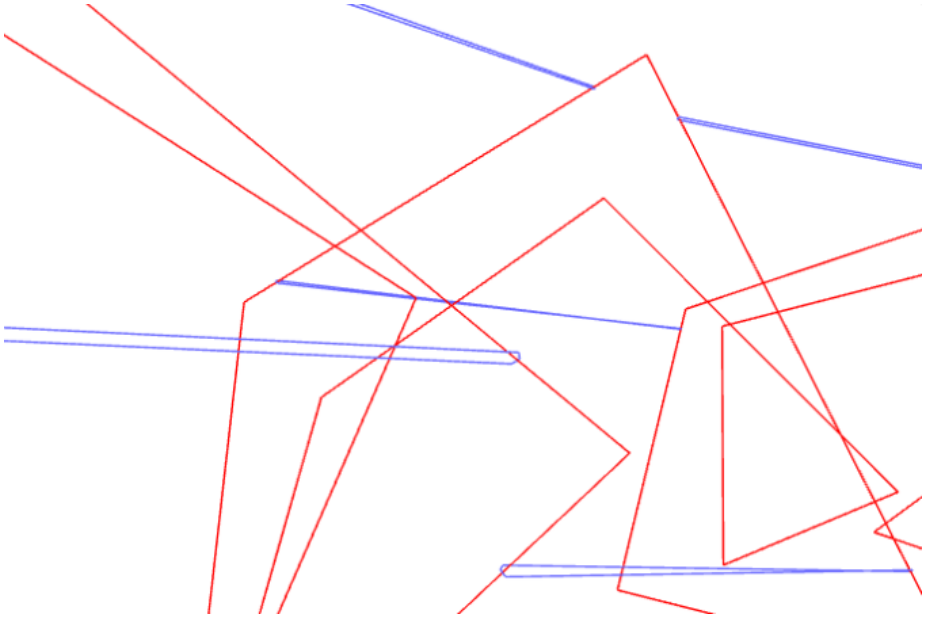


Figure 17: details of the link

11 Summary

A link is a subset of S^3 that is homeomorphic to the disjoint union of finitely many circles, each called link components. A link consisting of only 1 component is called a knot. Two links L_1, L_2 are equivalent if there is a homeomorphism $S^3 \to S^3$ mapping L_1 onto L_2. We will only consider tame links, i.e. links equivalent to a polygonal one.

Figure 18: example of a link diagram, from [17]

Using a regular projection of a link L, we can construct the resulting link diagram from smaller tangle diagrams using the operations disjoint union ($\sqcup$) and stitching (m_{ij} means stitching edge i to edge j). The smallest tangle diagrams are the positive and negative crossings (figure 19).

Figure 19: crossings

For any finitely presented group G, the conjugation action of G^{ab} turns $[G,G]^{ab}$ into a finitely presented $\mathbb{Z}[G^{ab}]$-module. If G is the fundamental group of a link complement, the gcd of all $m \times m$ minors of any $m \times n$ matrix representing this module is called the Alexander polynomial of this link. It equals the gcd of all $(k-1)\times(k-1)$ minors of any $k \times l$ Jacobian matrix of a presentation of G. The tangle-Jacobian matrix of tangle diagrams is a generalization of the Jacobian matrix of the so-called extended Wirtinger presentation of the group of a link.

The adjugate $\widetilde{A}$ of a matrix A is the transpose of the matrix formed by all cofactors. Writing the determinant of the original matrix in the upper left corner, the adjugates of the tangle-Jacobian matrices of the positive and resp. negative crossings look like

1	a	b	c
a	1	0	$1-b$
b	0	1	a
c	0	0	1

and

1	a	b	c
a	1	0	$a^{-1}(b-1)$
b	0	1	a^{-1}
c	0	0	1

.

Here a stands for the over-strand, and b and c for the outgoing and resp. incoming part of the under-strand. While they are fixed as column and row labels, the variables inside the table correspond to link components and should hence

be identified accordingly. The equations

$$\begin{aligned}(\Delta_A, \widetilde{A}) \sqcup (\Delta_B, \widetilde{B}) &= (\Delta_A \cdot \Delta_B, \Delta_B \cdot \widetilde{A} \oplus \Delta_A \cdot \widetilde{B}) \\ m_{ij}((\Delta, \widetilde{A})) &= (\Delta - \widetilde{A}^i_j, (1 - \Delta^{-1}\widetilde{A}^i_j)\widetilde{A} + \Delta^{-1}\widetilde{A}_j \cdot \widetilde{A}^i)\end{aligned}$$

now give a method of computing the adjugate and the determinant of a given link, provided $\Delta \neq 0$ before each stitching. Some reductions are possible to reduce the size of the matrices involved; the only rows needed are those corresponding to outgoing edges and if only one variable is used (e.g. in the case of a knot), the only columns needed are those corresponding to incoming edges. The Alexander polynomial equals the gcd of all remaining entries when there is just one stitching left.

The Alexander polynomial can be used to compute topological invariants of proteins. These are large molecules consisting of one or multiple chains of amino acids. Structural information about many proteins can be found in the RCSB Protein Data Bank, in which each PDB file contains the information about one protein. Using Sage code we can reconstruct the protein from it and compute certain (single-variable) Alexander polynomials corresponding to it. Firstly, a link is obtained by closing chains along hydrogen bonds. Running the code on a sample of PDB files suggests that almost no protein has a knotted structure. Secondly, a link is obtained from the protein by doubling all chains and connecting these double chains at the endpoints. Hydrogen bonds are then added as loops which are linked at both endings with the chain they are connected to. The corresponding Alexander polynomial, however, is very large and takes a lot of time to be computed.

12 Discussion

In the purely mathematical part of this thesis, a few loose ends have remained. More could be said about the relation between lemma 3.23 and corollary 3.29 on the one hand and the Jacobian matrix on the other hand, using the fundamental formula from proposition 9.8 of [8] or possibly group cohomology. It might also be possible to apply a further reduction to the tangle-Jacobian method of the multi-variable Alexander polynomial. In addition, I have not investigated if there is a topological interpretation of it for tangles. And it is unclear whether the Γ-calculus and rMVA really throw away valuable information and if the extension of the Γ-calculus as mentioned in question 7.3 always computes the multi-variable Alexander polynomial.

Concerning the computation of Alexander polynomials of proteins there are some improvements that can be made. Firstly, disulfide bonds were not taken into consideration, which are important to the structure of certain proteins. They could be treated just as hydrogen bonds. We could also consider only hydrogen bonds that really contribute to the global structure of proteins and e.g. not those that hold a helix together.

Secondly, the calculation time can be improved by taking a different order of stitching that is aimed at working with the smallest possible matrices. This comes down to minimizing the number of leaves in all tangle diagrams, which is related to finding minimum cuts in the graph: computing the Jacobian-adjugate of both sides of a minimum cut and then applying disjoint union minimizes the maximal size of the used matrices. This method could be applied recursively. Moreover, Sage does not seem very good at reducing fractions of polynomials, which probably adds a lot to the calculation time and partly explains the huge difference between the time it takes to compute a polynomial over $\mathbb{Z}$ and over $\mathbb{Z}/p\mathbb{Z}$. So one could try to adapt the code by adding a reduction step after each computation, or use other mathematical software.

Also, being a first attempt, the method of creating a link from the protein probably can be improved. Other choices of links related to proteins may be possible, or the exact positioning of the backbones and loops representing hydrogen bonds might be improved.

Finally, one could examine if higher order Alexander ideals and polynomials or smaller minors of the Jacobian matrix give rise to non-trivial invariants of spatial graphs.

Despite these possibly interesting improvements, it is far from clear that in general proteins really posses a structure that we could consider as being knotted. Considering the results of the first method, it seems that the methods of researches such as [2] often infer a knotted structure that is absent in the actual protein.

A Sage code: read PDB file

```
def read(filename): # Input is a pdb file, output a list of atoms [coordinates, atom letter, chain number, amino acid number] + a list of numbers of the last amino acid in each chain.
  file=open(filename)
  pdbfile=[l for l in file]
  L=[]
  z=len(pdbfile)
  finished=false
  chainends=[]
  chainnumber=0
  aminoacidnumber=0
  for i in range(z):
    l=pdbfile[i]
    firstword=l[0:6]
    if firstword=="ATOM  ":
      n=int(l[23:26])
      break
  for j in range(i,z):
    l=pdbfile[j]
    if l[0:6]=="ATOM  " and (l[77]=="H" or l[77]=="D"):
      break
  if j==z-1:
    return 0
  for j in range(i,z):
    l=pdbfile[j]
    firstword=l[0:6]
    if firstword=="ATOM  ":
      m=int(l[23:26])
      if m>n:
        if m>n+1:
          chainnumber+=1
          chainends.append(aminoacidnumber)
        aminoacidnumber+=1
        n=m
      v=[vector([float(l[30:38]),float(l[38:46]),float(l[46:54])]),l[77],chainnumber,aminoacidnumber]
      if v[1]=="D":
        v[1]="H"
      L.append([vector([float(l[30:38]),float(l[38:46]),float(l[46:54])]),l[77],chainnumber,aminoacidnumber])
    elif firstword=="TER   ":
      chainnumber+=1
      chainends.append(aminoacidnumber)
      for k in range(j+1,z):
        l=pdbfile[k]
        firstword=l[0:6]
        if firstword=="ENDMDL":
          finished=true
          break
        elif firstword=="ATOM  ":
          n=int(l[23:26])-1
          break
    elif firstword=="ENDMDL":
      chainends.append(aminoacidnumber)
      break
    if finished:
      break
```

```
54    i=0
55    while i<len(L)-1: # delete double atoms
56      j=i+1
57      while j<len(L):
58        if L[j][2]==L[i][2] and L[j][3]==L[i][3]:
59          if (L[j][0][0]-L[i][0][0])^2+(L[j][0][1]-L[i][0][1])
      ^2+(L[j][0][2]-L[i][0][2])^2<0.25 and L[i][1]==L[j][1]: #
      0.5^2=0.25
60            if (not L[i][1]=="H") or (L[j][0][0]-L[i][0][0])
      ^2+(L[j][0][1]-L[i][0][1])^2+(L[j][0][2]-L[i][0][2])
      ^2<0.1:
61              del L[j]
62            else:
63              j+=1
64          else:
65            j+=1
66        else:
67          break
68      i+=1
69    length=len(L)
70    return [L,chainends]
71
72  covalentradii={"H":float(0.43),"C":float(0.9),"N":float
      (0.83),"O":float(0.81),"S":float(1.12), "P":float(1.17)}
      #a small margin of error is added
73
74  def edges(A): # Input is a list of atoms as generated in
      read, output is a list of lists in which list nr i is the
       set of atoms with covalent bonds to atom i.
75    l=len(A)
76    E=[[] for i in range(l)]
77    for i in range(l-1):
78      j=i+1
79      while j<l and A[j][3]-A[i][3]<=1:
80        if A[i][2]==A[j][2]:
81          if A[i][3]==A[j][3] or (A[i][1]=="C" and A[j][1]=="N
      ") or (A[i][1]=="N" and A[j][1]=="C") or (A[i][1]=="S"
      and A[j][1]=="S"): #covalent bonds (generally) only
      appear inside amino acids or in peptide or disulfide
      bonds
82            if not (A[i][1]=="H" and A[j][1]=="H"): #extra
      check to make sure no H-H bond is listed, since they only
       occur in H_2-molecules.
83              v=covalentradii[A[i][1]]+covalentradii[A[j][1]]
84              if (A[j][0][0]-A[i][0][0])^2+(A[j][0][1]-A[i
      ][0][1])^2+(A[j][0][2]-A[i][0][2])^2<v^2:
85                E[i].append(j)
86                E[j].append(i)
87        j+=1
88    return E
89
90  def hydrogenbonds(A,E): # Input is a list of atoms 'A' (see
      'read') and a list of covalent bonds 'E' (see 'edges'),
      output is a list of lists in which list nr i is the set
      of amino acids connected to amino acid i by a hydrogen
      bond.
91    B=sorted([[A[t][0],A[t][1],A[t][2],A[t][3],t] for t in
      range(len(A))]) #'A', 's' and 't' are used for the
      original list of atoms ordered by amino acid, 'B', 'i'
      and 'j' are used for the list of atoms ordered by
      coordinates.
```

```
92    l=len(B)
93    z=A[-1][3]+1
94    hydrogenbonds=[[] for i in range(z)]
95    hbondpresent=[false for i in range(z)]
96    for i in range(l-1):
97      s=B[i][4]
98      j=i+2
99      while j<l and B[j][0][0]-B[i][0][0]<2.5:
100       t=B[j][4]
101       if not (B[i][2]==B[j][2] and abs(B[i][3]-B[j][3])<2)
        and (not j in E[s]) and ((B[i][1]=="H" and len(E[s])>0
        and B[j][1] in ["O","N","P"]) or (B[j][1]=="H" and len(E[
        t])>0 and B[i][1] in ["O","N","P"])):
102         d=(B[j][0][0]-B[i][0][0])^2+(B[j][0][1]-B[i][0][1])
        ^2+(B[j][0][2]-B[i][0][2])^2
103         if d<6.25: #2.5^2
104           hbond=true
105           for c in E[t]:
106             if (A[c][0][0]-B[i][0][0])^2+(A[c][0][1]-B[i
        ][0][1])^2+(A[c][0][2]-B[i][0][2])^2<d+(A[c][0][0]-B[j
        ][0][0])^2+(A[c][0][1]-B[j][0][1])^2+(A[c][0][2]-B[j
        ][0][2]): #other bondings should be at the right side of
        atom j such that there can be a hydrogen bond
107               hbond=false
108               break
109           if hbond:
110             for c in E[s]:
111               if (A[c][0][0]-B[j][0][0])^2+(A[c][0][1]-B[j
        ][0][1])^2+(A[c][0][2]-B[j][0][2])^2<d+(A[c][0][0]-B[i
        ][0][0])^2+(A[c][0][1]-B[i][0][1])^2+(B[c][0][2]-B[i
        ][0][2]): #other bondings should be at the right side of
        atom i such that there can be a hydrogen bond
112                 hbond=false
113                 break
114             if hbond:
115               a=sorted([B[i][3],B[j][3]])
116               hydrogenbonds[a[0]].append(a[1])
117               hbondpresent[B[i][3]]=true
118               hbondpresent[B[j][3]]=true
119       j+=1
120   hydrogenbonds=[sorted(list(set(b))) for b in hydrogenbonds
      ]
121   hydrogenbonds=[[a,b] for a in range(z) for b in
      hydrogenbonds[a]]
122   return [hydrogenbonds,hbondpresent]
123
124 def backbone(B,E): # Input is a list of atoms 'B' (see 'read
      ') and a list of covalent bonds 'E', output is the two
      parallel backbones b0 and b1 (as described in section 8)
      as lists of coordinates. If there are multiple backbones,
       b0 and b1 still need to be cut into pieces to find these
      .
125   b0=[]
126   b1=[]
127   i=0
128   j=0
129   remember=0
130   tryagain=false
131   while j<len(B):
132     t=B[j]
133     if t[3]==i:
```

```
        if t[1]=="C":
          if len(E[j])>=2:
            d=[B[c][1] for c in E[j]]
            if "N" in d and "C" in d:
              e=[c for c in E[j] if B[c][1]=="C"]
              for x in e:
                f=[y for y in E[x] if B[y][1]=="N" or B[y][1]=="O"]
                if len(f)+tryagain==2:
                  b0.append(t[0])
                  s=[B[c][0] for c in E[j] if c!=x and B[c][1]!="N" and B[c][1]!="H"]
                  if len(s)==1:
                    b1.append(s[0])
                  else:
                    g=[B[c][0] for c in E[j] if B[c][1]!="H"]
                    b1.append(2*t[0]-0.5*(g[0]+g[1])) # If there is no side chain (i.e. if the amino acid is glycine), the next point on b1 is the average of the central C atom's neighbours, reflected in this C atom.
                  i+=1
                  remember=i
                  tryagain=false
                  break
      elif t[3]==i+1:
        if tryagain: # stop if the second attempt to find a central C atom failed
            show('unable to find complete backbone, probably due to an incomplete pdb file or because this is not a protein')
            return [[],[]]
        tryagain=true #If no central C atom found, try it again with looser conditions
        j=remember-1
      j+=1
      if j==len(B) and i==B[-1][3]:
        if tryagain:
          show('unable to find complete backbone, probably due to an incomplete pdb file')
          return [[],[]]
        tryagain=true
        j=remember-1
    return [b0,b1]
```

B Sage code: create link

```
def createhbond(v1,v2,v3,w): # Input v1, v2 and v3 are three consecutive points in a chain; a hydrogen bond connects v2 and w. Output is a list of 3 points that gives a part of the loop that represents this hydrogen bond at the side of v2
  u=(v3-v1)/(v3-v1).norm()
  u=vector([float(round(a,4)) for a in u])
  h=(w-v2)/(w-v2).norm()
  h=vector([float(round(a,4)) for a in h])
  cosalpha=float(round(u.dot_product(h),4))
  if cosalpha>0:
    r=(v3-v2)/(v3-v2).norm()
  else:
```

```
      r=(v1-v2)/(v1-v2).norm()
    p=r*(abs(cosalpha)+0.01)/20
    p=vector([float(round(a,6)) for a in p])
    q2=(-h)/50
    q3=q2.cross_product(r)*sign(cosalpha)
    q1=-q3
    q1+=v2+p
    q2+=v2+p
    q3+=v2+p
    q1=vector([float(round(a,5)) for a in q1])
    q2=vector([float(round(a,5)) for a in q2])
    q3=vector([float(round(a,5)) for a in q3])
    return [q1,q2,q3]


def createlink(b0,b1,chainends,hbonds,hbondpresent): # Input
     is both backbones, a list of endpoints of chains, a list
     of hydrogen bonds and a list that indicates wheter there
     is a hydrogenbond attached to each amino acid. Output is
     a list 'L' of chains that go forth along the second
    backbone and back along the first one and loops along 6
    points representing the hydrogen bonds, a list 'Lsimple'
    of backbones of chains and a list 'hbondindices' of
    hydrogen bonds, described as pairs of amino acids they
    connect. From every chain, the first part before the
    first hydrogen bond and the last part after the last
    hydrogen bond is deleted.
    chains=[[chainends[t]+1,chainends[t+1]] for t in range(-1,
     len(chainends)-1)]
    chains[0]=[0,chainends[0]]
    chainnumber=0
    backboneindices=[0 for j in range(chainends[-1]+1)]
    m=[max([b0[t][i] for t in range(len(b0))])-min([b0[t][i]
     for t in range(len(b0))]) for i in [0,1,2]]
    if m[0]<m[1] and m[0]<m[2]: # permute coordinates such
     that the projection onto the xy-plane is likely to have
     less crossings
      for a in range(len(b0)):
        b0[a]=vector([b0[a][1],b0[a][2],b0[a][0]])
        b1[a]=vector([b1[a][1],b1[a][2],b1[a][0]])
    elif m[1]<m[2]:
      for a in range(len(b0)):
        b0[a]=vector([b0[a][0],b0[a][2],b0[a][1]])
        b1[a]=vector([b1[a][0],b1[a][2],b1[a][1]])
    t=0
    i=0
    while t<len(chainends):
      if hbondpresent[i]:
        chains[t][0]=i
        for j in range(i,chainends[t]+1):
          backboneindices[j]=[chainnumber,j-i]
        chainnumber+=1
        i=chainends[t]+1
        t+=1
        if t>=len(chainends):
          break
      else:
        i+=1
        if i==chains[t][1]+1:
          chains[t]=0
          t+=1
```

```
    t=-1
    i=chainends[-1]
    while t>=-len(chainends):
      if chains[t]==0:
        t-=1
        if t>=-len(chainends):
          if chains[t]!=0:
            i=chains[t][1]
      elif hbondpresent[i]:
        chains[t][1]=i
        t-=1
        if t>=-len(chainends):
          if chains[t]!=0:
            i=chains[t][1]
      else:
        i-=1
    chains=[c for c in chains if c]
    L=[]
    Lsimple=[]
    for c in chains:
      L.append([b1[d] for d in range(c[0],c[1]+1)]+list(
      reversed([b0[d] for d in range(c[0],c[1]+1)])))
      Lsimple.append([b0[d] for d in range(c[0],c[1]+1)])
    hbondloops=[]
    hbondindices=[sorted([backboneindices[h[0]],
      backboneindices[h[1]]]) for h in hbonds]
    for h in hbonds:
      [c1,i]=backboneindices[h[0]]
      [c2,j]=backboneindices[h[1]]
      a2=L[c1][i]
      b2=L[c2][j]
      a1=L[c1][i-1]
      b1=L[c2][j-1]
      a3=L[c1][i+1-len(L[c1])]
      b3=L[c2][j+1-len(L[c2])]
      hbondloops.append(createhbond(a1,a2,a3,b2)+createhbond(
      b1,b2,b3,a2))
    L.extend(hbondloops)
    return [L,Lsimple,hbondindices]
```

C Sage code: link projection

```
def nearby(L,m=0): # Input is a list of lists of points 'L'
    and 'm'=sqrt(d), where d is the critical distance. Output
     is a list in which entry i,j is a list of the points
    that are closer than d to point i,j after projection to
    the xy-plane.
  if m==0:
    m=5/4*max([max([(L[i][j][0]-L[i][j+1-len(L[i])][0])^2+(L
    [i][j][1]-L[i][j+1-len(L[i])][1])^2 for j in range(len(L[
    i]))]) for i in range(len(L))])
  N=[[[[] for k in L] for j in i] for i in L]
  for i1 in range(len(L)):
    for j1 in range(len(L[i1])):
      for j2 in range(j1+2,len(L[i1])-(j1==0)):
        if (L[i1][j1][0]-L[i1][j2][0])^2+(L[i1][j1][1]-L[i1
    ][j2][1])^2<m:
          N[i1][j1][i1].append(j2)
          N[i1][j2][i1].append(j1)
```

```
11        for i2 in range(i1+1,len(L)):
12          for j2 in range(len(L[i2])):
13            if (L[i1][j1][0]-L[i2][j2][0])^2+(L[i1][j1][1]-L[
      i2][j2][1])^2<m:
14              N[i1][j1][i2].append(j2)
15              N[i2][j2][i1].append(j1)
16    return N
17
18  def makeregular(D): # Input and output is a list of lists of
       points. Each list represents a link component, given by
      drawing straight line segments between consequent points.
       This function adapts coordinates such that the
      projection to the xy-plane is injective up to a finite
      number of points.
19    directions=[]
20    for i in range(len(D)):
21      for j in range(len(D[i])):
22        d=vector([D[i][j+1-len(D[i])][t]-D[i][j][t] for t in
      [0,1]])
23        if d[0]!=0:
24          d=d/d[0]
25        elif d[1]!=0:
26          d=sign(d[1])*vector([0,1])
27        else:
28          D[i][j][0]+=0.0001
29          d=vector([D[i][j+1-len(D[i])][t]-D[i][j][t] for t in
       [0,1]])
30          d=d/d[0]
31          show('coordinates adapted due to possible non-
      regular projection (1)')
32        directions.append([d,i,j])
33    directions=sorted(directions)
34    for t in range(len(directions)-1):
35      if directions[t][0]==directions[t+1][0]:
36        v=[D[directions[t+1][1]][directions[t+1][2]][i]-D[
      directions[t][1]][directions[t][2]][i] for i in [0,1]]
37        dt=matrix([directions[t][0],v]).det() # zero if both
      edges are on the same line
38        if dt==0:
39          show('coordinates adapted due to possible non-
      regular projection (2)')
40          if directions[t][0]!=0:
41            D[directions[t][1]][directions[t][2]][0]+=0.0001
42          else:
43            D[directions[t][1]][directions[t][2]][1]+=0.0001
44
45  def crossings(D,nearby): # Input is a list of sequences of
      points 'D', each representing a link component, and a
      list of unconnected vertices close enough in the
      projection 'nearby'. Output is a list 'cr' with a list of
       crossings for each edge ordered by distance to point i
      and a list of signs of all crossings 'crlist'.
46    makeregular(D)
47    C=[[vector([c[0],c[1]]) for c in B] for B in D]
48    cr=[[[] for c in B] for B in C]
49    crlist=[]
50    i=0
51    t=0
52    for i in range(len(C)):
53      l=len(C[i])
54      for j in range(l):
```

```
55        pointsnearedge=[[p for p in set(nearby[i][j][h]).union
      (nearby[i][j+1-l][h]) if ([h,p]>[i,j+1] or (h==i and p==0
       and j!=0 and j+1<l))] for h in range(len(C))]
56        pointsnearedge=[[h,p] for h in range(len(
      pointsnearedge)) for p in pointsnearedge[h]]
57        for k in pointsnearedge:
58          if len(C[k[0]])==k[1]+1:
59            m=0
60          else:
61            m=k[1]+1
62          if [k[0],m] in pointsnearedge:
63            dt1=matrix([C[i][j+1-l]-C[i][j],C[k[0]][m]-C[k
      [0]][k[1]]]).det()
64            if not dt1==0:
65              dt2=matrix([C[k[0]][k[1]]-C[i][j],C[i][j+1-l]-C[
      i][j]]).det()
66              mu=dt2/dt1
67              if 0<=mu<=1:
68                s=(C[i][j+1-l][0]-C[i][j][0]==0) # to prevent
      dividing by 0 in the next line
69                labda=(mu*(C[k[0]][m][s]-C[k[0]][k[1]][s])+C[k
      [0]][k[1]][s]-C[i][j][s])/(C[i][j+1-l][s]-C[i][j][s])
70                if 0<labda<1:
71                  if 0<mu<1:
72                    c1=D[i][j][2]+labda*(D[i][j+1-l][2]-D[i][j
      ][2])
73                    c2=D[k[0]][k[1]][2]+mu*(D[k[0]][m][2]-D[k
      [0]][k[1]][2])
74                    overstr=(c1>c2) # true if ij is over-
      strand, false if k is
75                    crtype=(overstr*2-1==sign(dt1)) # true if
      the crossing is positive, false if it is negative, all
      directions following the sequences
76                    cr[i][j].append([labda,t,overstr])
77                    cr[k[0]][k[1]].append([mu,t,not(overstr)])
78                    crlist.append(crtype)
79                    t+=1
80                  elif mu==0: # minor shifts if the endpoint
      of an edge is projected unto another edge
81                    C[k[0]][k[1]]+=0.0001*(C[k[0]][m]-C[k[0]][
      k[1]])
82                    show('coordinates adapted due to non-
      regular projection (3)')
83                  elif mu==1:
84                    C[k[0]][m]+=0.0001*(C[k[0]][k[1]]-C[k[0]][
      m])
85                    show('coordinates adapted due to non-
      regular projection (3)')
86                elif labda==0:
87                  C[i][j]+=0.0001*(C[i][j+1-l]-C[i][j])
88                  show('coordinates adapted due to non-regular
       projection (3)')
89                elif labda==1:
90                  C[i][j+1-l]+=0.0001*(C[i][j]-C[i][j+1-l])
91                  show('coordinates adapted due to non-regular
       projection (3)')
92        cr[i][j]=sorted(cr[i][j])
93        for a in range(len(cr[i][j])-1):
94          if cr[i][j][a][0]==cr[i][j][a+1][0]:
95            show('double crossing')
96        for c in cr[i][j]:
```

```
97          del c[0]
98    return [cr,crlist]
```

D Sage code: compute alexander

```
1  prime=101
2  P.<t>=PolynomialRing(GF(prime),'t') # or use ZZ
3  R=FractionField(P)
4  pC=[1,matrix(R,2,2,[[1,1-t],[0,t]],sparse=true)]
5  nC=[1,matrix(R,2,2,[[1,1-t^-1],[0,t^-1]],sparse=true)]
6  npC=[nC,pC]
7
8  def disjointunion(l): # Input is a list of pairs X=[omega,A]
       and output their disjoint union.
9    n=len(l)
10   list=[]
11   omega=1
12   for i in range(n):
13     omega=omega*l[i][0]
14     list.append(l[i][1])
15   for i in range(n):
16     list[i]=list[i]*omega/l[i][0]
17   return [omega,block_diagonal_matrix(list, subdivide=false)
       ]
18
19 def rename(l,n,m): # Input is a list 'l' of row/column
       numbers corresponding to the original strand labels, and
       the numbers 'n' and 'm' such that n is deleted and will
       refer to m. Output is an updated 'l'.
20   newl=[x for x in l]
21   k=len(l)
22   for i in range(k):
23     if l[i]==l[n]:
24       newl[i]=l[m]
25     if newl[i]>l[n]:
26       newl[i]-=1
27   return newl
28
29 def stitch(X,h,t,a=0,b=1): # Input is a pair X=[omega,A],
       the row numbers corresponding to the strands (heads) 'h',
        the column numbers corresponding to the strands (tails)
       't' and the strand number 'a' that is stitched to strand
       number 'b'.
30   omega = X[0]; A=X[1]
31   mu = A[h[a],t[b]]
32   newOmega = omega-mu
33   if newOmega==0 or A.ncols()==1:
34     return [[0,omega,A],h,t]
35   v=matrix([[A[i,t[b]]] for i in range(A.nrows()) if i!=h[a
       ]])
36   w=matrix([[A[h[a],j] for j in range(A.ncols()) if j!=t[b
       ]]])
37   newA=(1-omega^(-1)*mu)*(A.delete_columns([t[b]]).
       delete_rows([h[a]]))+omega^(-1)*v*w
38   h=rename(h,a,b)
39   t=rename(t,b,a)
40   return [[newOmega,newA],h,t]
41
```

```
42  def alexlink(L,cr,crlist): # Input is a link 'L' represented
        by a list of lists of points, a list of crossings for
      each edge 'cr' and 'crlist', which lists the type of each
       crossing. Output is the single-variable Alexander
      polynomial of this link, denoted as a tuple of its
      coefficients.
43    F=[1,matrix(sparse=true)]
44    h=range(2*len(crlist))
45    t=range(2*len(crlist))
46    count=0
47    tt=0
48    newlabels=[-1 for c in crlist]
49    endstitches=[]
50    if len(crlist)==0:
51      if len(L)>1:
52        return vector([0])
53      else:
54        return vector([1])
55    for k in range(len(cr)):
56      i=0
57      loopfinished=false
58      while i<(len(cr[k]))/2:
59        for s in range(len(cr[k][i])):
60          c=cr[k][i][s]
61          if s==len(cr[k][i])-1:
62            j=i-len(cr[k])+1
63            while len(cr[k][j])==0:
64              j+=1
65            if j>=-i:
66              if k+1==len(cr) and len(endstitches)==0:
67                p=list(vector(F[0].numerator().list())/F[0].
      denominator().list()[-1])
68                #p=F[0].numerator().list()
69                if len(p)>1:
70                  while p[0]==0:
71                    del p[0]
72                  for v in range(len(p)):
73                    p[v]=int(p[v])
74                    if p[v]>=prime/2:
75                      p[v]-=prime
76                  p=vector(p)
77                  p=p*sign(p[-1])
78                else:
79                  p=vector([0])
80                return p
81              else:
82                loopfinished=true #stopping the while-loop
83                break
84            d=cr[k][j][0]
85          else:
86            d=cr[k][i][s+1]
87          if newlabels[c[0]]==-1:
88            F=disjointunion([F,npC[crlist[c[0]]]])
89            newlabels[c[0]]=count
90            count+=1
91          if newlabels[d[0]]==-1:
92            F=disjointunion([F,npC[crlist[d[0]]]])
93            newlabels[d[0]]=count
94            count+=1
95          [F,h,t]=stitch(F,h,t,2*newlabels[c[0]]-c[1]+1,2*
      newlabels[d[0]]-d[1]+1)
```

```
96          tt+=1
97          if F[0]==0:
98            endstitches.append([2*newlabels[c[0]]-c[1]+1,2*
      newlabels[d[0]]-d[1]+1])
99            del F[0]
100       if loopfinished:
101         break
102       for s in range(len(cr[k][-i-1])):
103         c=cr[k][-i-1][-s-1]
104         if s==len(cr[k][-i-1])-1:
105           j=i+2
106           while len(cr[k][-j])==0:
107             j+=1
108           if j>=len(cr[k])-i:
109             if k+1==len(cr) and len(endstitches)==0:
110               p=list(vector(F[0].numerator().list())/F[0].
      denominator().list()[-1])
111               #p=F[0].numerator().list()
112               if len(p)>1:
113                 while p[0]==0:
114                   del p[0]
115                 for v in range(len(p)):
116                   p[v]=int(p[v])
117                   if p[v]>=prime/2:
118                     p[v]-=prime
119                 p=vector(p)
120                 p=p*sign(p[-1])
121               else:
122                 p=vector([0])
123               return p
124             else:
125               i=len(cr[k]) #stopping the while-loop
126           d=cr[k][-j][-1]
127         else:
128           d=cr[k][-i-1][-s-2]
129         if newlabels[c[0]]==-1:
130           F=disjointunion([F,npC[crlist[c[0]]]])
131           newlabels[c[0]]=count
132           count+=1
133         if newlabels[d[0]]==-1:
134           F=disjointunion([F,npC[crlist[d[0]]]])
135           newlabels[d[0]]=count
136           count+=1
137         [F,h,t]=stitch(F,h,t,2*newlabels[d[0]]-d[1]+1,2*
      newlabels[c[0]]-c[1]+1)
138         tt+=1
139         if F[0]==0:
140           endstitches.append([2*newlabels[d[0]]-d[1]+1,2*
      newlabels[c[0]]-c[1]+1])
141           del F[0]
142       i+=1
143   if len(endstitches)>0:
144     del endstitches[-1]
145   i=0
146   j=0
147   while i in range(len(endstitches)): # last stitches to be
      made
148     e=endstitches[i]
149     [F,h,t]=stitch(F,h,t,e[0],e[1])
150     if F[0]==0: # go to the next stitching first Delta=0
151       del F[0]
```

```
152         endstitches.append(e)
153         del endstitches[i]
154         j+=1
155         if j+i>=len(endstitches): # if all remaining
          stitchings yield Delta=0
156           return vector([0])
157       else:
158         i+=1
159         j=0
160     #p=F[0].numerator().list()
161     p=list(vector(F[0].numerator.list())/F[0].denominator().
        list()[-1])
162     while p[0]==0:
163       del p[0]
164     for v in range(len(p)):
165           p[v]=int(p[v])
166           if p[v]>=prime/2:
167             p[v]-=prime
168     p=vector(p)
169     p=p*sign(p[-1])
170     return p
171
172   def alexsimple(chains,hbonds): # Input is a list of chains
        and a list of hydrogen bonds, just as the second and
        third output of the function 'createlink'. Output is the
        Alexander polynomials of all chains, glued together at
        the hydrogen bond that turns the chain into the longest
        loop, and the Alexander polynomial of the link formed by
        these loops.
173     polynomials=[]
174     chainslocal=[c for c in chains]
175     bonds=[[[h[0][1],h[1][1]] for h in hbonds if h[0][0]==h
        [1][0]==i] for i in range(len(chains))]
176     i=0
177     while i<len(bonds):
178       if len(bonds[i])==0:
179         del bonds[i]
180         del chainslocal[i]
181       else:
182         i+=1
183     lengths=[[[b[1]-b[0]] for b in r] for r in bonds]
184     maxbonds=[l.index(max(l)) for l in lengths]
185     newchains=[[chainslocal[i][j] for j in range(bonds[i][
        maxbonds[i]][0],bonds[i][maxbonds[i]][1]+1)] for i in
        range(len(chainslocal))]
186     if len(newchains)==0:
187       show("no hydrogen bonds to the same chain found")
188       return 0
189     else:
190       n=nearby(newchains)
191       [cr,crlist]=crossings(newchains,n)
192       polynomials.append(alexlink(newchains,cr,crlist))
193       if len(newchains)>1:
194         for c in newchains:
195           n=nearby([c])
196           [cr,crlist]=crossings([c],n)
197           polynomials.append(alexlink([c],cr,crlist))
198     return polynomials
```

E Sage code: run

```
import os
load('/vol/home/s1137859/sagefiles/read PDB file.sage','/vol
    /home/s1137859/sagefiles/create link.sage','/vol/home/
    s1137859/sagefiles/link projection.sage','/vol/home/
    s1137859/sagefiles/compute alexander2.sage')
source='/vol/home/s1137859/Proteins'
l=[]
for protein in os.listdir(source):
  fullpath = os.path.join(source, protein)
  try:
    R=read(fullpath)
    if R!=0:
      [L,chainends]=R
      E=edges(L)
      [b0,b1]=backbone(L,E)
      if b0!=[]:
        [hbonds,hbondpresent]=hydrogenbonds(L,E)
        c=createlink(b0,b1,chainends,hbonds,hbondpresent)
        if c[0]!=[]:
          polynomials[protein[0:4]]=alexsimple(c[1],c[2])
        else:
          polynomials[protein[0:4]]=0
  except:
    pass
```

References

[1] J. W. Alexander, *Topological Invariants of Knots and Links*. Trans. Amer. Math. Soc. 30, 275-306, 1928. http://www.maths.ed.ac.uk/~aar/papers/alex1.pdf

[2] K. Alexander, A.J. Taylor, M.R. Dennis, *Proteins analysed as virtual knots*. ArXiv article, 2016. https://arxiv.org/abs/1611.06185

[3] P. W. Atkins, *Chemical bonding*. 23 January 2017. https://www.britannica.com/science/chemical-bonding/Covalent-bonds

[4] P. W. Atkins, *Chemical bonding*. 23 January 2017. https://www.britannica.com/science/chemical-bonding/Intermolecular-forces

[5] P. W. Atkins, *Chemical bonding*. 23 January 2017. https://www.britannica.com/science/chemical-bonding/Molecular-shapes-and-VSEPR-theory

[6] D. Bar-Natan, *Cheat Sheet Meta-Calculi*, 2016. http://drorbn.net/AcademicPensieve/Classes/17-1350-AKT/nb/170207-GammaCalculus.pdf

[7] D. Bar-Natan, *Meta-Monoids, Meta-Bicrossed Products, and the Alexander Polynomial*. Talk at the 2012 CMS Summer Meeting, Regina, June 2012. http://www.math.toronto.edu/drorbn/Talks/Regina-1206/

[8] G. Burde, H. Zieschang, *Knots, Second Revised and Extended Edition*. de Gruyter Studies in Mathematics, Berlin, 2003.

[9] C. Chieh, *Bond Lengths and Energies*. http://www.science.uwaterloo.ca/~cchieh/cact/c120/bondel.html

[10] The Editors of Encyclopædia Britannica, *Hydrogen bonding*. 5 January 2017. https://www.britannica.com/science/hydrogen-bonding

[11] G.M. Fisher, *On the Group of all Homeomorphisms of a Manifold*. Trans. Amer. Math. Soc. 97 (1960), 193-212. http://www.ams.org/journals/tran/1960-097-02/S0002-9947-1960-0117712-9/S0002-9947-1960-0117712-9.pdf

[12] R.H. Fox, *Free Differential Calculus. I: Derivation in the Free Group Ring*. Annals of Mathematics, 57(3), second series, 547-560. https://www.jstor.org/stable/1969736

[13] I. Halacheva, *Alexander Type Invariants of Tangles, Skewe Howe Duality for Crystals and The Cactus Group*. PhD Thesis, University of Toronto, 2016. http://blog.math.toronto.edu/GraduateBlog/files/2016/03/Halacheva-thesis.pdf

[14] A. Hatcher, *Algebraic Topology*. Cambridge University Press, 2002. https://www.math.cornell.edu/~hatcher/AT/AT.pdf

[15] F. Haurowitz, D.E. Koshland, *Protein*. 26 May 2017. https://www.britannica.com/science/protein

[16] G.A. Jeffrey, *An introduction to hydrogen bonding*. Oxford University Press, 1997.

[17] W.B.R. Lickorish, *An Introduction to Knot Theory*. Springer Science+Business Media, New York, 1997.

[18] *RCSB Protein Data Bank*. https://www.rcsb.org/pdb/home/home.do

[19] D. Rolfsen, *Knots and Links*. AMS Chelsea Publishing, Providence, Rhode Island, 2003.

[20] L. Tubiana, E. Orlandini, C. Micheletti, *Probing the entanglement and locating knots in ring polymers: a comparative study of different arc closure schemes*. Progress of Theoretical Physics Supplements 191, 192204 (2011). https://doi.org/10.1143/PTPS.191.192

[21] V.G. Turaev, *Introduction to Combinatorial Torsions*. Birkhäuser Verlag, Basel, 2001.

[22] M. Winter, *Covalent Radius: periodicity*. https://www.webelements.com/periodicity/covalent_radius/